

VISUALDSP++™ 3.1

Linker and Utilities Manual

for Blackfin Processors

Revision 2.1, April 2003

Part Number
82-000410-05

Analog Devices, Inc.
Digital Signal Processor Division
One Technology Way
Norwood, Mass. 02062-9106



Copyright Information

© 2003 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, VisualDSP, the VisualDSP logo, Blackfin, the Blackfin logo, SHARC, the SHARC logo, TigerSHARC, and the TigerSHARC logo are registered trademarks of Analog Devices, Inc.

VisualDSP++, the VisualDSP++ logo, CROSSCORE, the CROSSCORE logo, and EZ-KIT Lite are trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual	xiii
Intended Audience	xiii
Manual Contents	xiv
What's New in This Manual	xv
Technical or Customer Support	xv
Supported Processors	xvi
Product Information	xvi
MyAnalog.com	xvii
DSP Product Information	xvii
Related Documents	xviii
Online Technical Documentation	xix
From VisualDSP++	xix
From Windows	xx
From the Web	xx
Printed Manuals	xxi
VisualDSP++ Documentation Set	xxi
Hardware Manuals	xxi
Data Sheets	xxi

CONTENTS

Contacting DSP Publications	xxii
Notation Conventions	xxii

LINKER

Software Development Flow	1-2
Compiling and Assembling	1-3
Inputs – C/C++ and Assembly Sources	1-3
Input Section Directives in Assembly Code	1-4
Input Section Directives in C/C++ Source Files	1-4
Linking	1-6
Loading and Splitting	1-7
Linking Overview	1-10
Linker Operation	1-10
Directing Linker Operation	1-12
Linking Process Rules	1-13
Linker Description File (.LDF)	1-14
Linking Environment	1-15
Project Builds	1-15
Expert Linker	1-20
Linker Warning and Error Messages	1-21
Link Target Description	1-22
Representing Memory Architecture	1-22
ADSP-BF535 Processor Memory Architecture Overview	1-23
Specifying the Memory Map	1-26
Placing Code on the Target	1-29

Passing Arguments for Simulation/Emulation	1-31
Linker Command-Line Reference	1-33
Linker Command Syntax	1-33
Command Line Object Files	1-34
Command-Line File Names	1-35
Objects	1-37
Linker Command-Line Switches	1-38
Switch Format	1-38
Switch Summary	1-39
@filename	1-41
-DprocessorID	1-41
-L path	1-42
-M	1-42
-MM	1-42
-Map file	1-42
-MDmacro[=def]	1-42
-S	1-43
-T filename	1-43
-e	1-43
-es sectionName	1-43
-ev	1-43
-h or -help	1-43
-i path	1-44
-jcs2l	1-44

CONTENTS

-jcs2l+	1-44
-keep symbolName	1-44
-o filename	1-44
-od filename	1-44
-pp	1-45
-proc ProcessorID	1-45
-s	1-45
-sp	1-45
-t	1-45
-v	1-45
-version	1-45
-warnonce	1-46
-xref filename	1-46
Memory Management Using Overlays	1-47
Memory Overlays	1-48
Overlay Managers	1-49
Memory Overlay Support	1-50
Example – Managing Two Overlays	1-53
Reducing Overlay Manager Overhead	1-60
Using PLIT{} and Overlay Manager	1-64
LINKER DESCRIPTION FILE	
LDF Guide	2-2
.LDF File Overview	2-3
Example – Basic .LDF File	2-4

Notes on Basic .LDF File Example	2-6
LDF Structure	2-10
Command Scoping	2-11
LDF Expressions	2-12
LDF Keywords, Commands, and Operators	2-13
Miscellaneous LDF Keywords	2-14
LDF Operators	2-15
ABSOLUTE() Operator	2-15
ADDR() Operator	2-16
DEFINED() Operator	2-17
MEMORY_SIZEOF() Operator	2-17
SIZEOF() Operator	2-18
Location Counter (.)	2-18
LDF Macros	2-19
Built-In LDF Macros	2-20
User-Declared Macros	2-21
LDF Macros and Command-Line Interaction	2-22
LDF Commands	2-22
ALIGN()	2-23
ARCHITECTURE()	2-24
ELIMINATE()	2-24
ELIMINATE_SECTIONS()	2-25
INCLUDE()	2-25
INPUT_SECTION_ALIGN()	2-25

CONTENTS

KEEP()	2-27
LINK_AGAINST()	2-27
MAP()	2-28
MEMORY{}	2-28
MPMEMORY{}	2-31
OVERLAY_GROUP{}	2-33
Ungrouped Overlay Execution	2-34
Grouped Overlay Execution	2-35
PLIT{}	2-37
PLIT Syntax	2-38
Allocating Space for PLITs	2-39
PLIT Examples	2-40
PLIT – Summary	2-41
PROCESSOR{}	2-42
RESOLVE()	2-44
SEARCH_DIR()	2-44
SECTIONS{}	2-45
SHARED_MEMORY{}	2-50
LDF Programming Examples	2-53
Linking for a Single-Processor System	2-54
Linking Large Uninitialized Variables	2-55
Linking for Assembly Source File	2-57
Linking for C Source File – Example 1	2-60
Linking for Complex C Source File – Example 2	2-63

Linking for Overlay Memory Example	2-68
--	------

EXPERT LINKER

Expert Linker Overview	3-2
Launching the Create LDF Wizard	3-4
Step 1: Specifying Project Information	3-5
Step 2: Specifying System Information	3-6
Step 3: Completing the LDF Wizard	3-8
Expert Linker Window Overview	3-9
Using the Input Sections Pane	3-10
Using the Input Sections Menu	3-10
Mapping an Input Section to an Output Section	3-12
Viewing Icons and Colors	3-13
Sorting Objects	3-15
Using the Memory Map Pane	3-16
Using the Context Menu	3-17
Tree View Memory Map Representation	3-21
Graphical View Memory Map Representation	3-22
Specifying Pre- and Post-Link Memory Map View	3-28
Zooming In and Out on the Memory Map	3-28
Inserting a Gap into a Memory Segment	3-32
Working With Overlays	3-33
Viewing Section Contents	3-35
Viewing Symbols	3-37
Managing Object Properties	3-40

CONTENTS

Managing Global Properties	3-41
Managing Processor Properties	3-42
Managing PLIT Properties for Overlays	3-44
Managing Elimination Properties	3-45
Managing Symbols Properties	3-47
Managing Memory Segment Properties	3-51
Managing Output Section Properties	3-53
Managing Packing Properties	3-55
Managing Alignment and Fill Properties	3-56
Managing Overlay Properties	3-58
Managing Stack and Heap in DSP Memory	3-60

ARCHIVER

Archiver Guide	4-2
Creating a Library From VisualDSP++	4-3
File Name Conventions	4-3
Making Archived Functions Usable	4-3
Writing Archive Routines: Creating Entry Points	4-4
Accessing Archived Functions From Your Code	4-5
Archiver File Searches	4-6
Archiver Command-Line Reference	4-7
elfar Command Syntax	4-7
Example elfar Command	4-8
Parameters and Switches	4-8
Constraints	4-9

Archiver Symbol Name Encryption Reference	4-11
---	------

LOADER

Boot Sequence	5-3
ADSP-BF535 Processor Booting	5-3
Restrictions Using the Second-Stage Loader	5-10
ADSP-BF531, ADSP-BF532, and ADSP-BF533 Processor Booting	5-12
Creating an .LDR File Using the Elfloader	5-15
Specifying Basic Loader Settings	5-16
Loader Command-Line Switches	5-18
Load Page Description	5-22
Boot Kernel Options for Specifying a Second-Stage Loader	5-24
Creating an .LDR using the ROM Splitter	5-28
File Formats	5-30
Boot Streams for the ADSP-BF535 Processor	5-31
Boot Streams for ADSP-BF531, ADSP-BF532, and ADSP-BF533	
Processors	5-37

FILE FORMATS

Source Files	A-2
C/C++ Source Files	A-2
Assembly Source Files (.ASM)	A-3
Assembly Initialization Data Files (.DAT)	A-3
Header Files (.H)	A-4
Linker Description Files (.LDF)	A-4

CONTENTS

Linker Command-Line Files (.TXT)	A-5
Build Files	A-5
Assembler Object Files (.DOJ)	A-5
Library Files (.DLB)	A-6
Linker Output Files (.DXE, .SM, and .OVL)	A-6
Memory Map Files (.MAP)	A-6
Loader Output Files in Intel Hex-32 Format (.LDR)	A-6
Splitter Output Files in ASCII Format (.LDR)	A-8
Debugger Files	A-9
Format References	A-10
 UTILITIES	
elfdump – ELF File Dumper	B-1
Disassembling a Library Member	B-3
Dumping Overlay Library Files	B-4
 INDEX	

PREFACE

Thank you for purchasing Analog Devices development software for digital signal processors (DSPs).

Purpose of This Manual

The *VisualDSP++ 3.1 Linker and Utilities Manual for Blackfin Processors* contains information about the linker and utilities programs for Blackfin[®] processors. The Blackfin processors are embedded processors that sport a Media Instruction Set Computing (MISC) architecture. This architecture is the natural merging of RISC, media functions, and digital signal processing (DSP) characteristics towards delivering signal processing performance in a microprocessor-like environment.

This manual provides information on the linking process and describes the syntax for the linker's command language—a scripting language that the linker reads from the linker description file. The manual leads you through using the linker, archiver, and loader to produce DSP programs and provides reference information on the file utility software.

Intended Audience

The manual is primarily intended for programmers who are familiar with Analog Devices DSPs. This manual assumes that the audience has a working knowledge of the appropriate device architecture and instruction set. Programmers who are unfamiliar with Analog Devices DSPs can use this

manual but should supplement it with other texts, such as *Hardware Reference* and *Instruction Set Reference* manuals, that describe your target architecture.

Manual Contents

The manual contains:

- Chapter 1, “[Linker](#)”

This chapter provides an overview of each utility.

- Chapter 2, “[Linker Description File](#)”

This chapter describes how to write an `.LDF` file to define the target.

- Chapter 3, “[Expert Linker](#)”

This chapter describes Expert Linker which is an interactive graphical tool to set up and map DSP memory.

- Chapter 4, “[Archiver](#)”

This chapter describes how to combine object files into reusable library files to link routines referenced by other object files.

- Chapter 5, “[Loader](#)”

This chapter describes loading/splitting for the processors. Topics include loader software and command-line switches; boot sequence, boot-kernels, and boot-loadable files.

- Appendix A, “[File Formats](#)”

This appendix lists and describes the file formats that the development tools use as inputs or produce as outputs

- Appendix B, “[Utilities](#)”

This appendix describes the file utilities that provide legacy and file conversion support

What's New in This Manual

This edition of the manual documents support for new Blackfin processors.

This manual has undergone extensive reorganization. A Preface has been added, and .LDF information has been placed in a separate chapter.

Chapter 5, “[Loader](#)” provides descriptions on new loader/splitter features.

Technical or Customer Support

You can reach DSP Tools Support in the following ways.

- Visit the DSP Development Tools Web site at
www.analog.com/technology/dsp/developmentTools/index.html
- Submit a DSP Tools Technical Support Form:
http://forms.analog.com/Form_Pages/DSP/tools/contactDSP.asp
- E-mail questions to dsptools.support@analog.com
- Phone questions to **1-800-ANALOGD**
- Contact your ADI local sales office or authorized distributor
- Send questions by mail to:

Analog Devices, Inc.

Supported Processors

DSP Division
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA

Supported Processors

The name “*Blackfin*” refers to a family of Analog Devices 16-bit, fixed-point processors. VisualDSP++ currently supports the following Blackfin processors:

- ADSP-BF532 (formerly ADSP-21532)
- ADSP-BF535 (formerly ADSP-21535)
- ADSP-BF531
- ADSP-BF533
- ADSP-DM102
- AD6532

The ADSP-BF531 and ADSP-BF533 are memory variants of the ADSP-BF532 DSP. The ADSP-DM102 dual-core processor is similar to the ADSP-BF532 DSP.

Product Information

You can obtain product information from the Analog Devices Web site, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at www.analog.com. Our Web site provides information about a broad range of products – analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notification containing updates to the Web pages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:

Visit www.myanalog.com to sign up. Click **Register** to use MyAnalog.com. Registration takes about five minutes and serves as means for you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your e-mail address.

DSP Product Information

For information on digital signal processors, visit our Web site at www.analog.com/dsp, which provides access to technical publications, data sheets, application notes, product overviews, and product announcements.

Product Information

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- E-mail questions or requests for information to
`dsp.support@analog.com`
- Fax questions or requests for information to
1-781-461-3010 (North America)
089/76 903-557 (Europe)
- Access the Digital Signal Processing Division's FTP Web site at
`ftp ftp.analog.com` or **ftp 137.71.23.21**
`ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications:

VisualDSP++ 3.1 Getting Started Guide for Blackfin Processors

VisualDSP++ 3.1 User's Guide for Blackfin Processors

VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.x Kernel (VDK) User's Guide

VisualDSP++ 3.x Component Software Engineering User's Guide

Quick Installation Reference Card

For hardware information, refer to the processor's *Hardware Reference Manual* and *Instruction Set Reference Manual*.

Online Technical Documentation

Online documentation comprises VisualDSP++ Help system and tools manuals, Dinkum Abridged C++ library and FlexLM network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary .PDF files for the tools manuals are also provided.

A description of each documentation file type is as follows.

File	Description
.CHM	Help system files and VisualDSP++ tools manuals.
.HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation. Viewing and printing the .HTML files require a browser, such as Internet Explorer 4.0 (or higher).
.PDF	VisualDSP++ tools manuals in Portable Documentation Format, one .PDF file for each manual. Viewing and printing the .PDF files require a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time by rerunning the Tools installation.

Access the online documentation from the VisualDSP++ environment, Windows Explorer, or Analog Devices Web site.

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's **C**ontents, **S**earch, and **I**ndex commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (.CHM files) are located in the `Help` folder, and .PDF files are located in the `Docs` folder of your VisualDSP++ installation. The `Docs` folder also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other .CHM files.

Using the Windows Start Button

- Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs, VisualDSP, and VisualDSP++ Documentation**.
- Access the .PDF files by clicking the **Start** button and choosing **Programs, VisualDSP, Documentation for Printing**, and the name of the book.

From the Web

To download the tools manuals, point your browser at http://www.analog.com/technology/dsp/developmentTools/gen_purpose.html.

Select a DSP family and book title. Download archive (.ZIP) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at 1-800-ANALOGD (1-800-262-5643) and follow the prompts.

VisualDSP++ Documentation Set

VisualDSP++ manuals may be purchased through Analog Devices Customer Service at 1-781-329-4700; ask for a Customer Service representative. The manuals can be purchased only as a kit. For additional information, call 1-603-883-2430.

If you do not have an account with Analog Devices, you will be referred to Analog Devices distributors. To get information on our distributors, log onto <http://www.analog.com/salesdir/continent.asp>.

Hardware Manuals

Hardware reference and instruction set reference manuals can be ordered through the Literature Center or downloaded from the Analog Devices Web site. The phone number is 1-800-ANALOGD (1-800-262-5643). The manuals can be ordered by a title or by product number located on the back cover of each manual.

Data Sheets

All data sheets can be downloaded from the Analog Devices Web site. As a general rule, any data sheet with a letter suffix (L, M, N) can be obtained from the Literature Center at 1-800-ANALOGD (1-800-262-5643) or downloaded from the Web site. Data sheets without the suffix can be downloaded from the Web site only—no hard copies are available. You can ask for the data sheet by a part name or by product number.

Notation Conventions

If you want to have a data sheet faxed to you, the phone number for that service is **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. Call the Literature Center first to find out if requested data sheets are available.


Contacting DSP Publications

Please send your comments and recommendation on how to improve our manuals and online Help. You can contact us by:



- E-mailing dsp.techpubs@analog.com
- Filling in and returning the attached Reader's Comments Card found in our manuals

Notation Conventions

The following table identifies and describes text conventions used in this manual.

 Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (File menu)	Text in bold style indicates the location of an item within the VisualDSP++ environment's menu system. For example, the Close command appears on the File menu.
{this that}	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> .
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .

Example	Description
[this,...]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with <i>letter gothic font</i> .
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	A note, providing information of special interest or identifying a related topic. In the online version of this book, the word Note appears instead of this symbol.
	A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word Caution appears instead of this symbol.

1 LINKER

This manual describes several program development tools, such as the linker, loader, splitter, archiver, and ELF file dumper.

This chapter includes:

- [“Software Development Flow” on page 1-2](#)
Shows how linking, loading, and splitting fit into the DSP project development process.
- [“Linking Overview” on page 1-10](#)
Describes the link target, linking environment, and other files used while linking.
- [“Linker Command-Line Reference” on page 1-33](#)
Describes linker command-line switches.
- [“Memory Management Using Overlays” on page 1-47](#)
Describes memory overlays as well as advanced LDF commands.

Chapter 2 focuses on the Linker Description File (`.LDF`), and Chapter 3 describes Expert Linker, an alternative tool that generates an `.LDF` file.

Software Development Flow

The majority of this manual describes linking, a critical stage in the program development process for embedded applications.

The linker utility (`linker.exe`) consumes object and library files to produce executable files, which can be loaded onto the simulator or target processor. The linker also produces map files and other output that contain information used by the debugger. Debug information is embedded in the executable file.

After running the linker, you test the output with the simulator or emulator. Refer to the *VisualDSP++ 3.1 User's Guide for Blackfin Processors* and online Help for information about debugging.

Finally, you process the debugged executable file(s) through the loader or splitter to create output for use on the actual processor. The output file may reside on another processor (host) or may be burned into a PROM. Chapter 5 describes options in generating these output files.

The processor software development flow can be split into three phases:

1. Compiling and Assembling – Input source files C (.C), C++ (.CPP), and assembly (.ASM) yield object files (.DOJ)
2. Linking – Under the direction of the Linker Description File (.LDF), a linker command line, and VisualDSP++ **Project Options** dialog box settings, the linker utility consumes object files (.DOJ) to yield an executable (.DXX) file. If specified, shared memory (.SM) and overlay (.OVL) files are also produced.
3. Loading or splitting – The executable (.DXX), as well as shared memory (.SM) and overlay (.OVL) files, are processed to yield output file(s). For Blackfin processors, these are bootloadable (.LDR) files or non-bootable PROM image files, which execute from processor external memory.

Compiling and Assembling

The process starts with source files written in C, C++, or assembly. The compiler (or developers that write assembly code) organizes each distinct sequence of instructions or data into named sections, which become the main components acted upon by the linker.

Inputs – C/C++ and Assembly Sources

The first step towards producing an executable is to compile or assemble C, C++, or assembly source files into *object files*. The VisualDSP++ development software assigns a `.DOJ` extension to object files ([Figure 1-1](#)).

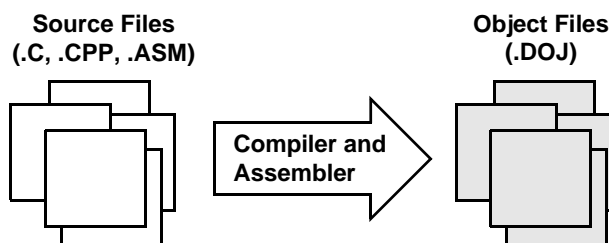


Figure 1-1. Compiling and Assembling

Object files produced by the compiler (via the assembler) and by the assembler itself consist of *input sections*. Each input section contains a particular type of compiled/assembled source code. For example, an input section may consist of program opcodes or data, such as variables of various widths.

Some input sections may contain information to enable source-level debugging and other VisualDSP++ features. The linker maps each input section (via a corresponding output section in the executable) to a *memory segment*, a contiguous range of memory addresses on the target system.

Each input section in the .LDF file requires a unique name, as specified in the source code. Depending on whether the source is C, C++, or assembly, different conventions are used to name an input section (see [“Linker Description File \(.LDF\)” on page 1-14](#)).

Input Section Directives in Assembly Code

A `.SECTION` directive defines a section in assembly source. This directive must precede its code or data.

Example

```
.SECTION Library_Code_Space;    /* Section Directive */
.global _abs;
_abs:
    R0 = ABS R0;                /* Take absolute value of input */
    RTS;
_abc.end
```

In this example, the assembler places the global symbol/label `_abs` and the code after the label into the input section `Library_Code_Space`, as it processes this file into object code.

Input Section Directives in C/C++ Source Files

Typically, C/C++ code does not specify an input section name, so the compiler uses a default name. By default, the input section names `program` (for code) and `data1` (for data) are used. Additional input section names are defined in .LDF files.

In C/C++ source files, you can use the optional `section("name")` C language extension to define sections.

Example

While processing the following code, the compiler stores the `temp` variable in an input section of the .DOJ file called `ext_data`, and also stores the code generated from `func1` in an input section named `extern`.

```
...
section ("ext_data") int temp;          /* Section directive */
section ("extern") void func1(void) { int x = 1; }
...
```

Example

In the following example, section ("extern") is optional. Note the new function (func2) does not require section ("extern"). For more information on LDF sections, refer to [“Specifying the Memory Map” on page 1-26](#).

```
section ("ext_data") int temp;
section ("extern") void func1(void) { int x = 1; }
                      int func2(void) { return 13; }      /* New */
```

For information on compiler default section names, refer to the *VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors* and column 1 of [Table 1-2 on page 1-28](#).



Identify the difference between input section names, output section names, and memory segment names, because these types of names appear in the .LDF file. Usually, default names are used. However, there may be situations when you may want to use non-default names. This happens when various functions or variables (in the same source file) are to be placed into different memory segments.

Linking

After you have (compiled and) assembled source files into object files, use the linker to combine the object files into an executable file. By default, the development software gives executable files a .DXE extension (Figure 1-2).

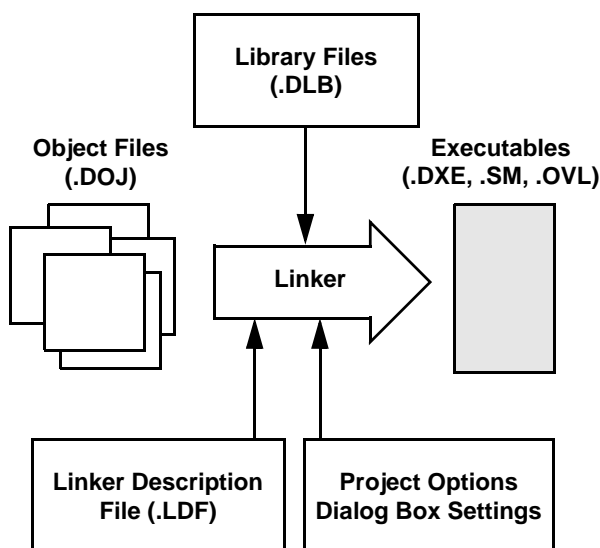


Figure 1-2. Linking Diagram

Linking is instrumental in enabling that your code runs efficiently in the target environment. Linking is described in detail in [“Linker Operation” on page 1-10](#).

i When developing a new project, use the Expert Linker to generate the project’s .LDF file. See [“Expert Linker” on page 3-1](#).

Loading and Splitting

After debugging the `.DXE` file, you process it through a loader or splitter to create output files used by the actual processor. This file may reside on another processor (host) or may be burned into a PROM. Chapter 5 describes options for generating bootloadable `.LDR` (loader) files.

Chapter 5 also describes splitting, which creates non-bootloadable files that execute from the processor's external memory.

Figure 1-3 shows a simple application of the loader. In this example, the loader's input is a single executable (`.DXE`). The loader can accommodate up to two `.DXE` files as input plus one boot-kernel file (`.DXE`).

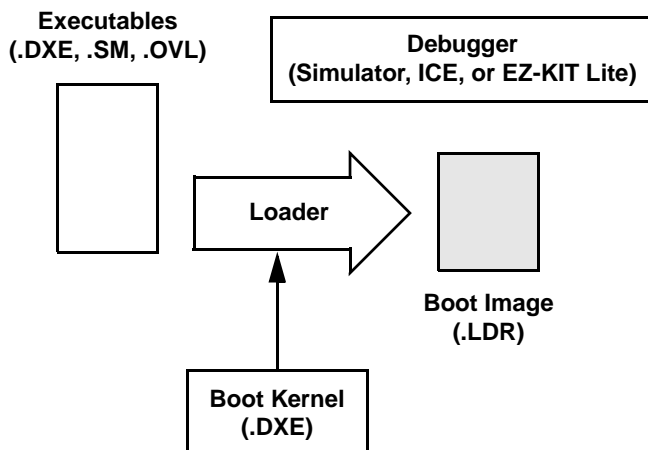


Figure 1-3. Loading Diagram

For Blackfin processors, use the loader (`elfloader.exe`) to yield a boot-loadable image (`.LDR`), which resides in memory external to the processor (PROM or host processor). When the processor is reset, the boot-kernel

portion of the image is transferred to the processor's core, and then the instruction and data portion of the image are loaded into the processor's internal RAM (see [Figure 1-4](#)) by the boot-kernel.

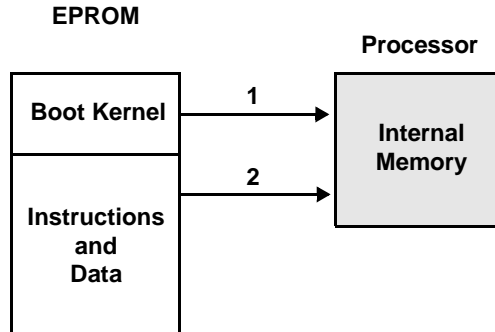


Figure 1-4. Booting from a Bootloadable (.LDR) File

VisualDSP++ includes boot-kernel files (.DxE) which are automatically used when you run the loader. You can also customize boot-kernel source files (also included with VisualDSP++) by modifying and rebuilding them.

[Figure 1-5](#) shows how multiple input files – in this case, two executable (.DxE) files, a shared memory (.SM) file, and overlay files (.OVL) – are consumed by the loader to create a single image file (.LDR). The example demonstrates the generation of a loader file for a multiprocessor architecture.

i The .SM and .OVL files must reside in the same directory as the input .DxE file(s) or in the current working directory. If your system does not use shared memory or overlays, .SM and .OVL files are not required.

This example has two executables that share memory. In addition, overlays are also included. The resulting output is a compilation of all the inputs.

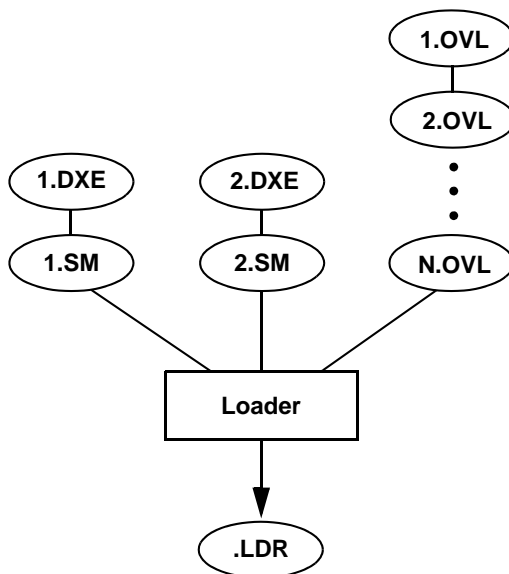


Figure 1-5. Input Files for a Multiprocessor System

Linking Overview

Linking assigns code and data to processor memory. For a simple single-processor architecture, a single `.DxE` file is generated. Repeat the same linking process to create multiple executable files (`.DxE`) for multiprocessor (MP) architectures. Linking can also produce a shared memory (`.SM`) file for an MP system. A large executable can be split into a smaller executable and overlays (`.OVL` files), which contain code that is called in (swapped into internal processor memory) as needed.

The linker (`linker.exe`) performs this task. The linker is run from a command line or from the VisualDSP++ Integrated Development and Debugging Environment (IDDE).

You can load the results of the link into the VisualDSP++ debugger for simulation, testing, and profiling.

Linker Operation

Figure 1-6 illustrates a basic linking operation. The figure shows several object files (`.DOJ`) being linked into a single executable file (`.DxE`). The Linker Description File (`.LDF`) directs the linking process.



When developing a new project, use the Expert Linker to generate the project's `.LDF` file. See [“Expert Linker” on page 3-1](#).

When an architecture is for a multiprocessor system, a `.DxE` file for each processor is generated. For example, for a multiprocessing application, you must generate two `.DxE` files. The processors in a multiprocessor architecture share memory. When directed by statements in the `.LDF` file, the linker produce a shared memory executable (`.SM`) file, whose code is used by multiple processors.

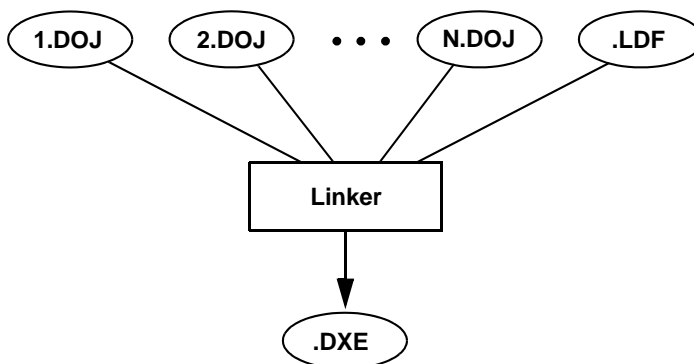


Figure 1-6. Linking Object Files to Produce an Executable File

Overlay files (`.OVL`), another linker output, support applications that require more program instructions and data than can fit in the processor's internal memory. Refer to [“Memory Management Using Overlays” on page 1-47](#) for more information.

Similar to object files, executable files are partitioned into *output sections* with their own names. Output sections are defined by the Executable and Linking Format (ELF) file standard to which VisualDSP++ conforms.

i The executable's input section names and output section names occupy different namespaces. Because the namespaces are independent, the same section names may be used. The linker uses input section names as labels to locate corresponding input sections within object files.

The executable file(s) and auxiliary files (`.SM` and `.OVL`) are not loaded into the processor or burned onto an EPROM. These files are used to debug the system.

Directing Linker Operation

Linker operations are directed by these options and commands:

- Linker (`linker.exe`) command-line switches (options)
- Settings (options) on the **Link** page of the **Project Options** dialog box (see [Figure 1-8 on page 1-17](#))
- LDF commands

Linker options control how the linker processes object files and library files. These options specify various criteria such as search directories, map file output, and dead code elimination. You select linker options via linker command-line switches (see [“Linker Command-Line Reference” on page 1-33](#)), or by settings on the **Link** page of the **Project Options** dialog box within the VisualDSP++ environment.

LDF commands in a Linker Description File (`.LDF`) define the target memory map and the placement of program sections within processor memory. The text of these commands provide the information needed to link your code. For a detailed description of the LDF commands, refer to [“LDF Guide” on page 2-2](#).



The VisualDSP++ **Project** window displays the `.LDF` file as a source file, though the file provides linker command input.

Using directives in the `.LDF` file, the linker:

- Reads input sections in the object files and maps them to output sections in the executable. More than one input section may be placed in an output section.
- Maps each output section in the executable to a *memory segment*, a contiguous range of memory addresses on the target processor. More than one output section may be placed in a single memory segment.


Linking Process Rules

The linking process observes these rules:

- Each source file produces one object file.
- Source files may specify one or more input sections as destinations for compiled/assembled object(s).
- The compiler and assembler produce object code with labels that direct various portion(s) to particular output sections.
- As directed by the `.LDF` file, the linker maps each input section in the object code to an output section in the `.DXE` file.
- As directed by the `.LDF` file, the linker maps each output section to a memory segment.
- Each input section may contain multiple code items, but a code item may appear in one input section only.
- More than one input section may be placed in an output section.
- Each memory segment must have a specified width.
- Contiguous addresses on different-width hardware must reside in different memory segments.
- More than one output section may map to a memory segment if the output sections fit completely within the memory segment.

Linker Description File (.LDF)

Whether you are linking C/C++ functions or assembly routines, the mechanism is the same. After converting the source files into object files, the linker uses directives in an .LDF file to combine the objects into an executable file (.DXE), which may be loaded into a simulator for testing.

 Executable file structure conforms to the Executable and Linkable Format (ELF) standard.

Each project must include one .LDF file that specifies the linking process by defining the target memory and mapping the code and data into that memory. You can write your own .LDF file, or you can modify an existing file; modification is often the easier alternative when there are few changes in your system's hardware or software. VisualDSP++ provides an .LDF file that supports the default mapping of each processor type.

 When developing a new project, use the Expert Linker to generate the project's .LDF file. See [“Expert Linker” on page 3-1](#).

Similar to an object (.DOJ) file, an executable (.DXE) file consists of different segments, called *output sections*. Input section names are independent of output section names. Because they exist in different namespaces, an input section name can be the same as an output section name.

Refer to [“Linker Description File” on page 2-1](#) for further information.

Linking Environment

The linking environment refers to Windows command prompt windows and the VisualDSP++ IDDE. At a minimum, run development tools (such as the linker) via a command line and view output in standard output.

VisualDSP++ provides an environment that simplifies the processor program build process. From VisualDSP++, you specify build options from the **Project Options** dialog box and modify files, including the Linker Description File (.LDF). The **Project Options** dialog box's **Type** option allows you to choose whether to build a library (.DLB) file, an executable (.EXE) file, or an image file. Error and warning messages display in the **Output** window.

Project Builds

The linker runs from an operating system command line, issued from the VisualDSP++ IDDE or a command prompt window. [Figure 1-7](#) shows the VisualDSP++ environment with the **Project** window and an .LDF file open in an editor window.

The IDDE provides an intuitive interface for processor programming. When opening VisualDSP++, a work area contains everything needed to build, manage, and debug a DSP project. You can easily create or edit an .LDF file, which maps code or data to specific memory segments on the target.



Refer to the *VisualDSP++ 3.1 User's Guide for Blackfin Processors* or online Help for information about the VisualDSP++ environment. Online Help provides powerful search capabilities. To obtain information on a code item, parameter, or error, select text in an VisualDSP++ IDDE editor window or **Output** window and press the keyboard's **F1** key.

Linking Overview

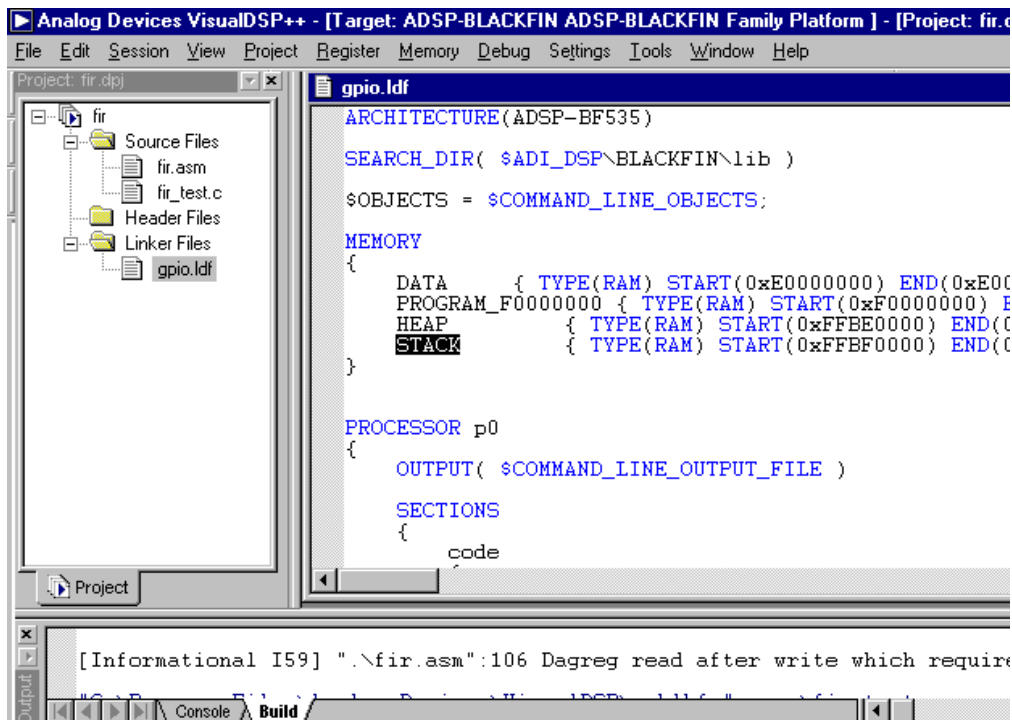


Figure 1-7. VisualDSP++ Environment

Within VisualDSP++, specify tool settings for project builds. Modify linker options via the **Link** page of the **Project Options** dialog box. Choosing a **Category** from the pull-down list at the top of the **Link** page presents different options.

The callouts in [Figure 1-8](#), [Figure 1-9](#), and [Figure 1-10](#) refer to corresponding linker command-line switches. In addition to selecting options on the **Link** page, use the page's **Additional options** field to specify switches and parameters that do not have corresponding **Link** page controls. Refer to "[Linker Command Syntax](#)" on [page 1-33](#) for more information.

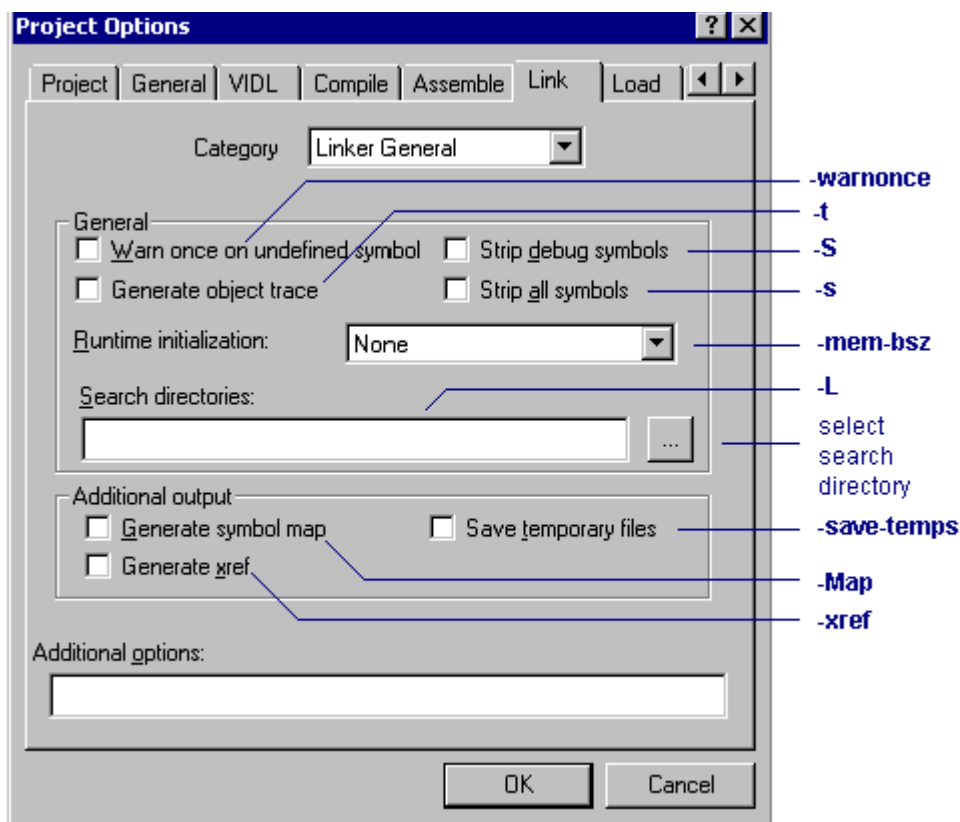


Figure 1-8. Link Page (1 or 3) of the Project Options Dialog Box

Linking Overview

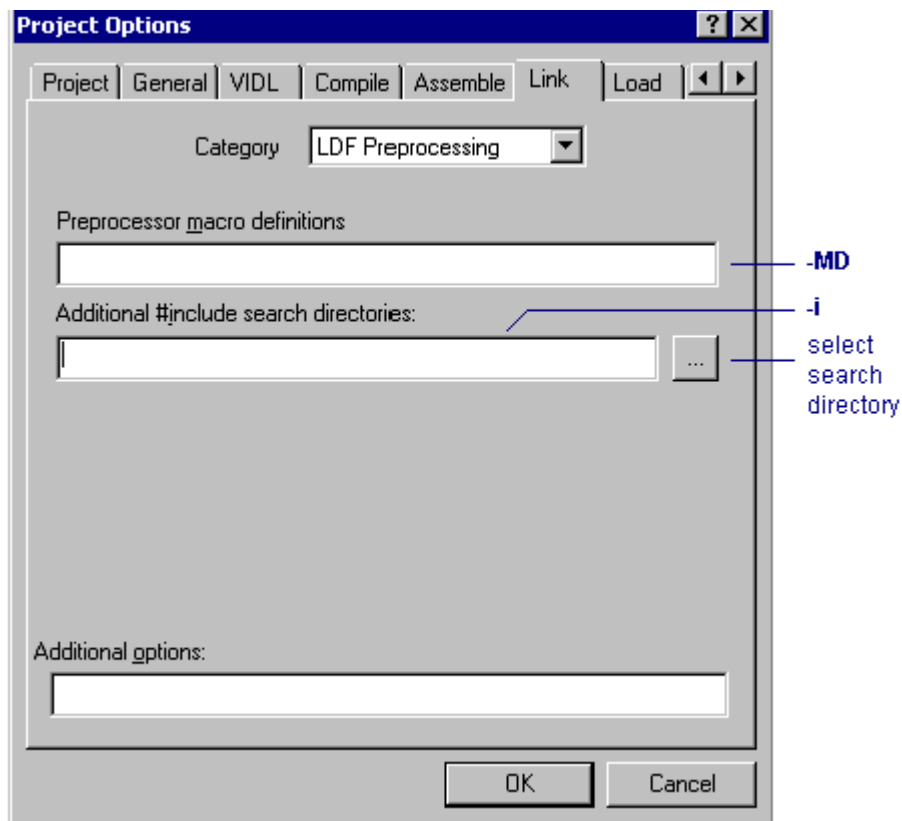


Figure 1-9. Link Page (2 of 3) of the Project Options Dialog Box

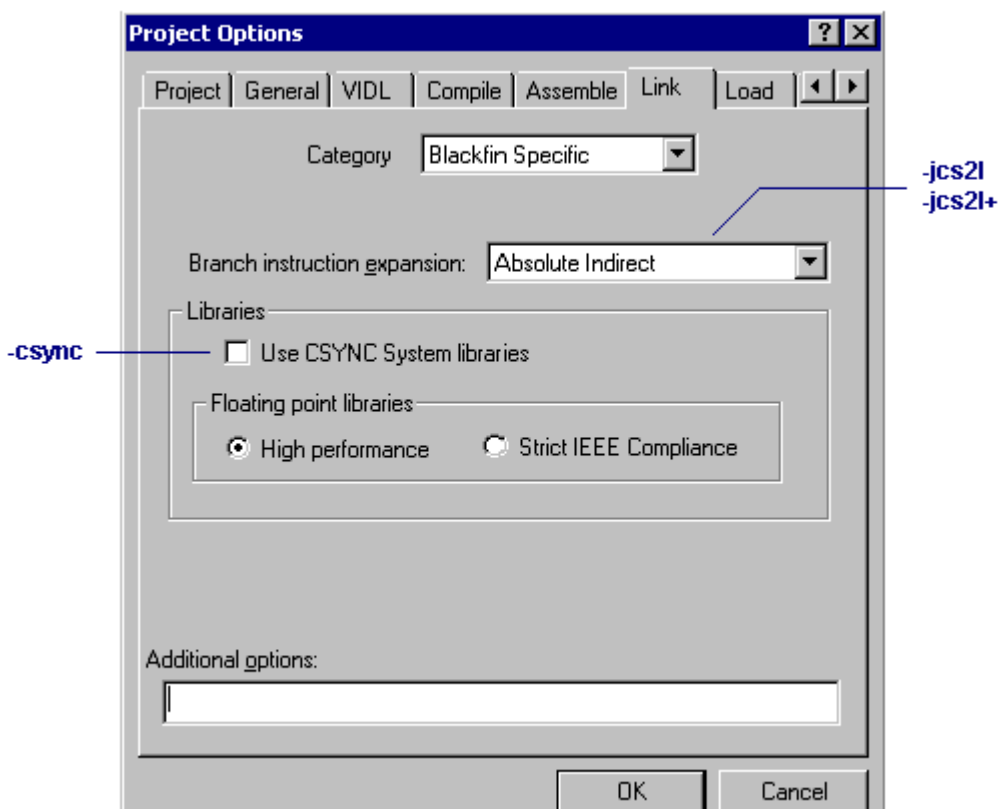


Figure 1-10. Link Page (3 of 3) of the Project Options Dialog Box

Linking Overview

Expert Linker

The VisualDSP++ IDDE features an interactive tool, *Expert Linker*, to map code or data to specific memory segments. When developing a new project, use the Expert Linker to generate the .LDF file.

Expert Linker graphically displays the .LDF information (object files, LDF macros, libraries, and a target memory description). With Expert Linker, use drag-and-drop operations to arrange the object files in a graphical memory mapping representation. When you are satisfied with the memory layout, generate the executable file (.DXX).

Figure 1-11 shows the Expert Linker window, which comprises two panes: **Input Sections** and **Memory Map** (output sections). Refer to “[Expert Linker](#)” on page 3-1 for additional information.

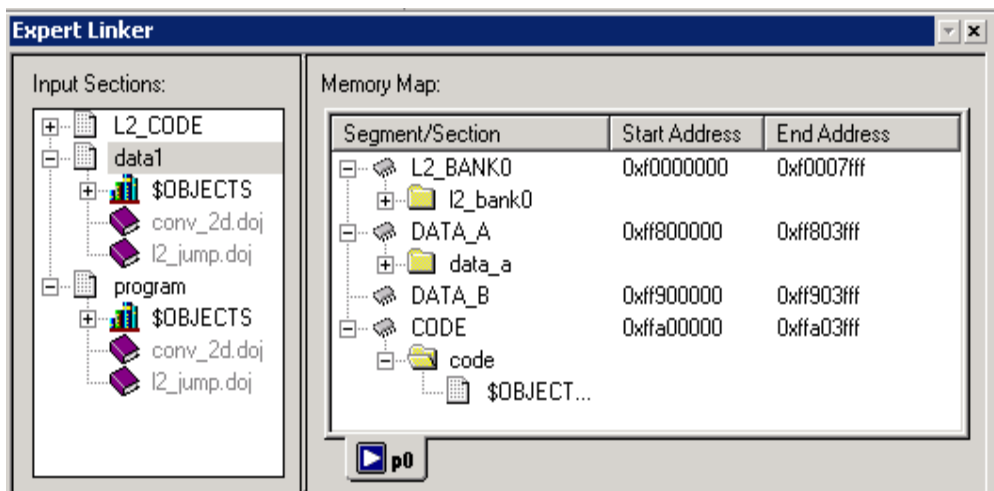


Figure 1-11. Expert Linker Window

Linker Warning and Error Messages

Linker messages are written to the VisualDSP++ **Output** window (standard output when the linker is run from a command line). Messages describe problems the linker encountered while processing the .LDF file. *Warnings* indicate processing errors that do not prevent the linker from producing a valid output file, such as an unused symbol in your code. *Errors* are issued when the linker encounters situations that prevent the production of a valid output file.

Typically, these messages include the name of the .LDF file, the line number containing the message, a six-character code, and a brief description of the condition.

Example

```
>linker -T nofile.ldf
[Error li1002] The linker description file 'NOFILE.LDF'
could not be found
Linker finished with 1 error(s) 0 warning(s)
```

Interpreting Linker Messages

Within VisualDSP++, the **Output** window's **Build** page displays project build status and error messages. In most cases, double-click on a message to view the line in the source file causing the problem. Descriptions of linker messages are accessible from VisualDSP++ online Help by selecting the six-character code (for example, li1002) and pressing the F1 key.

Some build errors, such as a bad or missing cross-reference to an object or executable file, do not correlate directly to source files. These errors often stem from omissions in the .LDF file.


Linking Overview

For example, if an input section from the object file is not placed by the LDF, a cross-reference error occurs at every object that refers to labels in the missing section. Fix this problem by reviewing the LDF and specifying all sections that need placement. For more information, refer to the *VisualDSP++ 3.1 User's Manual for Blackfin Processors* or online Help.

Link Target Description

Before defining the system's memory and program placement with linker commands, analyze the target system, so you can describe the target in terms the linker can process. Then, produce an .LDF file for your project to specify these system attributes:

- Physical memory map
- Program placement within the system's memory map

 If the project does not include an .LDF file, the linker uses a default LDF that matches the `-Darchitecture` switch on the linker's command line (or the **Processor** option specified on the **Project** page of the **Project Options** dialog box in the VisualDSP++ IDE). The examples in this manual are for ADSP-BF535 processors.

Be sure to understand the processor's memory architecture, which is described in the processor's *Hardware Reference* manual and in its data sheet.

Representing Memory Architecture

The .LDF file's `MEMORY{}` command is used to represent the memory architecture of your DSP system. The linker uses this information to place the executable file into the system's memory.

Perform the following tasks to write a `MEMORY{ }` command:

- **Memory Usage.** List the ways your program uses memory in your system. Typical uses for memory segments include interrupt tables, initialization data, program code, data, heap space, and stack space. Refer to [“Specifying the Memory Map” on page 1-26](#).
- **Memory Characteristics.** List the types of memory in your DSP system and the address ranges and word width associated with each memory type. Memory type is defined as `RAM` or `ROM`.
- **MEMORY{} Command** Construct a `MEMORY{ }` command to combine the information from the previous two lists and to declare your system’s memory segments.

For complete information, refer to [“MEMORY{}” on page 2-28](#).

ADSP-BF535 Processor Memory Architecture Overview

This section uses the Blackfin ADSP-BF535 as an example for to describe memory architecture and memory map organization. Other processors in the Blackfin family have different memory architectures.

The ADSP-BF535 processor includes the L1 memory subsystem with a 16 Kbyte instruction SRAM/cache, a dedicated 4 Kbyte data scratchpad, and a 32 Kbyte data SRAM/cache configured as two independent 16 Kbyte banks (memories). Each independent bank can be configured as SRAM or cache.

The ADSP-BF535 processor also has an L2 SRAM memory that provides 2 Mbits (256 Kbytes) of memory. The L2 memory is unified; that is, it is directly accessible by the instruction and data ports of the ADSP-BF535 processor. The L2 memory is organized as a multi-bank architecture of single-ported SRAMs (there are eight sub-banks in L2), such that simultaneous accesses by the core and the DMA controller to different banks can occur in parallel.

Linking Overview

The device has two ports into the L2 memory: one dedicated to core requests, and the other dedicated to system DMA and PCI requests. The processor units can process 8-, 16-, 32-, or 40-bit data, depending on the type of function being performed.

[Figure 1-12](#) shows the ADSP-BF535 system block diagram.

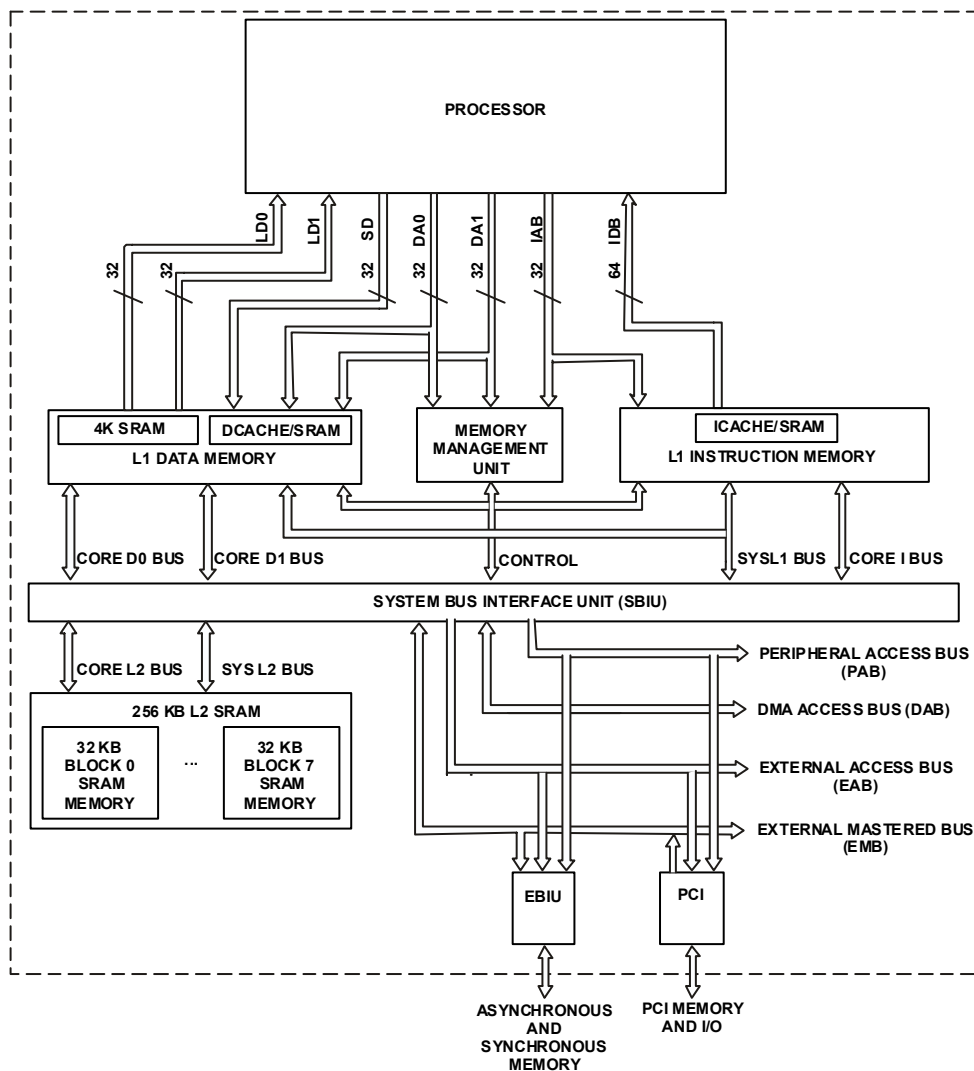


Figure 1-12. ADSP-BF535 System Block Diagram

Linking Overview

Memory ranges are listed in [Table 1-1](#). Address ranges that are not listed are reserved.

Table 1-1. ADSP-BF535 Processor Memory Map Addresses

Memory Range	Range Description
0xFFE00000 - 0xFFFFFFFF	Core MMR registers (2MB)
0xFFC00000 - 0xFFDFFFFFFF	System MMR registers (2MB)
0xFFB00000 - 0xFFB00FFF	Scratchpad SRAM (4K)
0xFFA00000 - 0xFFA03FFF	Instruction SRAM (16K)
0xFF900000 - 0xFF903FFF	Data Memory Bank 2 SRAM (16K)
0xFF800000 - 0xFF803FFF	Data Memory Bank 1 SRAM (16K)
0xFF0040000 - 0xFF7FFFFFFF	Reserved
0xF0000000 - 0xF003FFFF	L2 Memory Bank SRAM (256K)
0xEF000400 - 0xEFFFFFFF	Reserved
0xEF000000 - 0xEF0003FF	Boot ROM (1K)
0x00000000 - 0xEEFFFFFFF	Unpopulated

The MEMORY section in [Listing 1-1 on page 1-29](#) assume that only L1 and L2 SRAM are available and that L1 is unused. Refer to the *C/C++ Compiler and Library Manual for Blackfin Processors* and the appropriate *Hardware Reference* for information about cache configuration.

Specifying the Memory Map

A DSP program must conform to the constraints imposed by the processor's data path (bus) widths and addressing capabilities. The following steps show an .LDF file for a hypothetical project. This file specifies several memory segments that support the SECTIONS{} command. (See [“SECTIONS{}” on page 2-45](#).)

The three steps involved in allocating memory are demonstrated below.

1. **Memory usage** – Input section names are generated automatically by the compiler or are specified in the assembly source code. The LDF defines memory segment names and output section names.

The default `.LDF` file handles all compiler-generated input sections (refer to the “Input Section” column in [Table 1-2](#)). The produced `.DXX` file has a corresponding output section for each input section. Although programmers typically do not use output section labels, the labels are used by downstream tools.


Use the ELF file dumper utility (`elfdump.exe`) to dump contents of an output section (for example, `data1`) of an executable file. See [“elfdump – ELF File Dumper” on page B-1](#) for information about this utility.

Linking Overview

Table 1-2 shows the correspondence between input, output, and memory used in the default .LDF file for the ADSP-BF535 processor.

Table 1-2. Memory vs. Sections Usage for ADSP-BF535 LDF

Input Section	Output Section	Memory Section
program	dxe_program	MEM_PROGRAM
data1	dxe_program	MEM_PROGRAM
constdata	dxe_program	MEM_PROGRAM
heap	dxe_heap	MEM_HEAP
stack	dxe_stack	MEM_STACK
sysstack	dxe_sysstack	MEM_SYSSTACK
bootup	dxe_bootup	MEM_BOOTUP
ctor	dxe_program	MEM_PROGRAM
argv	dxe_argv	MEM_ARGV

 You can modify your .LDF file to place objects into L1 memories when they are configured as SRAM.

2. **Memory characteristics** – Blackfin processors have a 32-bit addressing range to support memory addresses from 0x0 to 0xFFFFFFFF.

Note: Some portions of the DSP memory are reserved. Refer to your processor's *Hardware Reference* manual.

3. **Linker MEMORY{} Command** – Referring to steps 1 and 2, specify the target's memory with the MEMORY{} command as shown in Listing 1-1.

Listing 1-1. MEMORY{} Command Code in an .LDF File

```
MEMORY          /* Define/label system memory      */
{               /* List of global Memory Segments */
    MEM_L2
        { TYPE(RAM) START(0xF0000000) END(0xF002FFFF) WIDTH(8) }
    MEM_HEAP
        { TYPE(RAM) START(0xF0030000) END(0xF0037FFF) WIDTH(8) }
    MEM_STACK
        { TYPE(RAM) START(0xF0038000) END(0xF003DFFF) WIDTH(8) }
    MEM_SYSSTACK
        { TYPE(RAM) START(0xF003E000) END(0xF003FDFF) WIDTH(8) }
    MEM_ARGV
        { TYPE(RAM) START(0xF003FE00) END(0xF003FFFF) WIDTH(8) }
}
```

Notes on the Above Example

The above example applies to the preceding discussion of how to write a `MEMORY{}` command and to the following discussion of the `SECTIONS{}` command. The `SECTIONS{}` command is not atomic; it can be interspersed with other directives, including location counter information. You can define new symbols within the .LDF file.

This example defines the starting stack address, the highest possible stack address, and the heap's starting location and size. These newly created symbols are entered in the executable's symbol table.

Placing Code on the Target

Use the `SECTIONS{}` command to map code and data to the physical memory of a processor in a DSP system.

Linking Overview

To write a `SECTIONS{}` command:

1. List all input sections defined in the source files.

Assembly files. List each assembly code `.SECTION` directive, identifying its memory type (PM or CODE, or DM or DATA) and noting when location is critical to its operation. These `.SECTIONS` portions include interrupt tables, data buffers, and on-chip code or data.

C/C++ source files. The compiler generates sections with the name “program” for code, and the names “data1” and “data2” for data. These sections correspond to your source when you do not specify a section by means of the optional `section()` extension.

2. Compare the input sections list to the memory segments specified in the `MEMORY{}` command. Identify the memory segment into which each `.SECTION` must be placed.
3. Combine the information from these two lists to write one or more `SECTIONS{}` commands in the `.LDF` file.



`SECTIONS{}` commands must appear within the context of the `PROCESSOR{}` or `SHARED_MEMORY()` command.

Listing 1-2. `SECTIONS{}` Command in the `.LDF` File

```
SECTIONS
{
    /* List of sections for processor P0 */

    dx_e_L2
    {
        INPUT_SECTION_ALIGN(2)
        /* Align all code sections on 2 byte boundary */
        INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS( $OBJECTS(constdata)
                        $LIBRARIES(constdata))
    }
}
```

```

        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
    } >MEM_L2

stack
{
    ldf_stack_space = .;
    ldf_stack_end =
        ldf_stack_space + MEMORY_SIZEOF(MEM_STACK) - 4;
} >MEM_STACK

sysstack
{
    ldf_sysstack_space = .;
    ldf_stack_end =
        ldf_stack_space + MEMORY_SIZEOF(MEM_STACK) - 4;
} >MEM_SYSSTACK

heap
{
    /* Allocate a heap for the application */
    ldf_heap_space = .;
    ldf_heap_end =
        ldf_heap_space + MEMORY_SIZEOF(MEM_HEAP) - 1;
    ldf_heap_length = ldf_heap_end - ldf_heap_space;
} >MEM_HEAP

argv
{
    /* Allocate argv space for the application */
    ldf_argv_space = .;
    ldf_argv_end =
        ldf_argv_space + MEMORY_SIZEOF(MEM_ARGV) - 1;
    ldf_argv_length =
        ldf_argv_end - ldf_argv_space;
} >MEM_ARGV

} /* end SECTIONS */

```

Passing Arguments for Simulation/Emulation

Linking Overview

To support simulation and emulation, the linker should obtain the start address and buffer length of the argument list from the ARGV memory segment of the .LDF file (refer to [“Example – Basic .LDF File” on page 2-4](#)).


To set the address:

1. In the MEMORY{ } section, add a line to define the MEM_ARGV section.
2. Add a command to define the ARGV section and the values for `ldf_argv_space`, `ldf_argv_length`, and `ldf_argv_end`.

If you modify the .LDF file and change the start or end of the MEM_ARGV section, you must pass the data to the VisualDSP++ IDDE as well. You do this via the following VisualDSP++ menu command from:

Settings->Simulator->Command Line Arguments

Refer to the *VisualDSP++ 3.1 User's Manual for Blackfin Processors* or online Help for information about the simulator and command-line arguments.

 Do not use command-line arguments for linked programs without first modifying the .LDF file to allocate a buffer suitable for your application.

Linker Command-Line Reference

This section provides reference information, including:

- “Linker Command Syntax”
- “[Linker Command-Line Switches](#)” on page 1-38



When using the linker via the VisualDSP++ IDDE, the settings on the **Link** page of the **Project Options** dialog box correspond to linker command-line switches. VisualDSP++ calls the linker with these switches when linking your code. For more information, refer to the *VisualDSP++ 3.1 User's Manual for Blackfin Processors* and VisualDSP++ online Help.

Linker Command Syntax

Run the linker using one of the following normalized formats of the linker command line.

```
linker -proc processorID -switch [-switch ...] object [object ...]
linker -T target.ldf -switch [-switch ...] object [object ...]
```

The linker command requires `-proc processorID` or a `-T <ldf name>` for the link to proceed. If the command line does not include `-proc processorID`, the `.LDF` file following the `-T` switch must contain a `-proc processorID` command.



The command line must have at least one object (an object file name). Other switches are optional, and some commands are mutually exclusive.

Example

The following is an example linker command.

Linker Command-Line Reference

```
linker -proc ADSP-BF535 p0.doj -T target.ldf -t -o program.dxe
```

-  Use `-proc processorID` instead of `-Darchitecture` on the command line to select the target processor. See [Table 1-4 on page 1-39](#) for more information.
-  The linker command line (except for file names) is case sensitive. For example, `linker -t` differs from `linker -T`.

When using the linker's command line, you should be familiar with the following topics:

- [“Command Line Object Files”](#)
- [“Switch Format” on page 1-38](#)
- [“Command-Line File Names” on page 1-35](#)

Command Line Object Files

The command line must list at least one (typically more) object file(s) to be linked together. These files may be of several different types.

- Standard object files (`.DOJ`) produced by the assembler.
- One or more libraries (archives), each with a `.DLB` extension. Examples include the C run-time libraries and math libraries included with VisualDSP++. You may create libraries of common or specialized objects. Special libraries are available from DSP algorithm vendors. For more information, see [“Archiver” on page 4-1](#)
- An executable (`.DXE`) file to be linked against. Refer to `$COMMAND_LINE_LINK_AGAINST` in [“Built-In LDF Macros” on page 2-20](#).

Object File Names

Object file names are not case sensitive. An object file name may include:

- The drive, directory path, file name, and file extension
- An absolute or relative path to the directory where the linker is invoked
- Long file names enclosed within straight quotes

If the file exists before the link begins, the linker opens the file to verify its type before processing the file. [Table 1-3](#) lists valid file extensions used by the linker.

Command-Line File Names

Some linker switches take a file name as an optional parameter. [Table 1-3](#) lists the types of files, names, and extensions that the linker expects on file name arguments. The linker follows the conventions for file extensions in [Table 1-3](#).

Table 1-3. File Extension Conventions

Extension	File Description
.DLB	Library (archive) file
.DOJ	Object file
.DXE	Executable file
.LDF	Linker Description File
.OV	Overlay file
.SM	Shared memory file

Linker Command-Line Reference

The linker supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur in the following order.

1. Specified path – If the command line includes relative or absolute path information on the command line, the linker searches that location for the file.
2. Specified directories – If you do not include path information on the command line and the file is not in the default directory, the linker searches for the file in the search directories specified with the `-L (path)` command-line switch, and then searches directories specified by `SEARCH_DIR` commands in the `.LDF` file. Directories are searched in order of appearance on the command line or in the LDF.
3. Default directory – If you do not include path information in the `.LDF` file named by the `-T` switch, the linker searches for the `.LDF` file in the current working directory. If you use a default `.LDF` file (by omitting LDF information in the command line and instead specify `-Darchitecture`), the linker searches in the processor-specific LDF directory; for example, `...\$ADI_DSP\Blackfin\ldf`.

For more information on file searches, see [“Built-In LDF Macros” on page 2-20](#).

When providing input or output file names as command-line parameters:

- Use a space to delimit file names in a list of input files.
- Enclose long file names within straight quotes; for example, “long file name”.
- Include the appropriate extension to each file. The linker opens existing files and verifies their type before processing. When the linker creates a file, it uses the file extension to determine the type of file to create.

Objects

The linker handles an object (file) by its file type. File type is determined by the following rules.

- Existing files are opened and examined to determine their type. Their names can be anything.
- Files created during the link are named with an appropriate extension and are formatted accordingly. A map file is formatted as text and given a `.MAP` extension. An executable is written in the ELF format and given a `.DXE` extension.

The linker treats object (`.DOJ`) and library (`.DLB`) files that appear on the command line as object files to be linked. The linker treats executable (`.DXE`) and shared memory (`.SM`) files on the command line as executables to be linked against.

For more information on objects, see the `$COMMAND_LINE_OBJECTS` macro. For information on executables, see the `$COMMAND_LINE_LINK_AGAINST` macro. Both are described in [“Built-In LDF Macros” on page 2-20](#).

If link objects are not specified on the command line or in the `.LDF` file, the linker generates appropriate informational/error messages.

Linker Command-Line Switches

This section describes the linker's command-line switches. [Table 1-4](#) briefly describes each switch with regard to case sensitivity, equivalent switches, switches overridden/contradicted by the one described, and naming and spacing constraints for parameters.

Switch Format

The linker provides switches to select operations and modes. The standard switch syntax is:

```
-switch [argument]
```

Rules

- Switches may be used in any order on the command line. Items in brackets [] are optional. Items in *italics* are user-definable and are described with each switch.
- Path names may be relative or absolute.
- File names containing white space or colons must be enclosed by double quotation marks, though relative path names such as `..\..\test.dxe` do not require double quotation marks.



Different switches require (or prohibit) white space between the switch and its parameter.

Example

```
linker p0.doj p1.doj p2.doj -T target.ldf -t -o program.dxe
```

Note the difference between the `-T` and the `-t` switches. The command calls the linker as follows:

- `p0.doj`, `p1.doj`, and `p2.doj` – Links three object files into an executable.
- `-T target.ldf` – Uses a secondary `.LDF` to specify executable program placement.
- `-t` – Turns on trace information, echoing each link object's name to stdout as it is processed.
- `-o program.dxe` – Specifies a name of the linked executable.

Typing `linker` without any switches displays a summary of command-line options. This is the same as typing `linker -help`.

Switch Summary

[Table 1-4](#) briefly describes each switch. Each individual switch is described in detail following this table.

Table 1-4. Linker Command-Line Switches – Summary

Switch	Description	More Info
<code>@file</code>	Uses the specified file as input on the command line.	on page 1-41
<code>-DprocessorID</code>	Specifies the target processor ID. The use of <code>-proc processorID</code> is recommended.	on page 1-41
<code>-L path</code>	Adds the path name to search libraries for objects.	on page 1-42
<code>-M</code>	Produces dependencies.	on page 1-42
<code>-MM</code>	Builds and produces dependencies.	on page 1-42
<code>-Map file</code>	Outputs a map of link symbol information to a file.	on page 1-42

Linker Command-Line Reference

Table 1-4. Linker Command-Line Switches – Summary (Cont'd)

Switch	Description	More Info
-D <i>macro</i> [= <i>def</i>]	Defines and assigns value <i>def</i> to a preprocessor macro.	on page 1-42
-S	Omits debugging symbols from the output file.	on page 1-43
-T <i>filename</i>	Names the LDF.	on page 1-43
-e	Eliminates unused symbols from the executable.	on page 1-43
-es <i>secName</i>	Names input sections (<i>secName</i> list) to which elimination algorithm is being applied.	on page 1-43
-ev	Eliminates unused symbols verbosely.	on page 1-43
-h -help	Outputs the list of command-line switches and exits.	on page 1-43
-i <i>path</i>	Includes search directory for preprocessor include files.	on page 1-44
-ip	Not currently available. Fills fragmented memory with individual data objects that fit. This option requires that objects have been assembled with the assembler's -ip switch.	
-jcs21	Converts out-of-range short calls and jumps to the longer form	on page 1-44
-jcs21+	Enables -jcs21 and allows the linker to convert out-of-range branches to indirect calls and jumps sequences	on page 1-44
-keep <i>symName</i>	Retains unused symbols.	on page 1-44
-o <i>filename</i>	Outputs the named executable file.	on page 1-44
-od <i>filename</i>	Specifies the output directory.	on page 1-44
-pp	Stops after preprocessing.	on page 1-45
-proc <i>processorID</i>	Selects a target processor. .	on page 1-45
-s	Strips symbol information from the output file.	on page 1-45
-sp	Skips preprocessing.	on page 1-45

Table 1-4. Linker Command-Line Switches – Summary (Cont'd)

Switch	Description	More Info
-t	Outputs the names of link objects.	on page 1-45
-v -verbose	Verbose. Outputs status information.	on page 1-45
-version	Outputs version information and exits.	on page 1-45
-warnonce	Warns only once for each undefined symbol.	on page 1-46
-xref <i>filename</i>	Outputs a list of all cross-referenced symbols.	on page 1-46

@filename

Uses *filename* as input to the linker command line. The @ switch circumvents environmental command-line length restrictions. *filename* may not start with “linker” (that is, it cannot be a linker command line). White space (including “newline”) in *filename* serves to separate tokens.

-DprocessorID

Specifies target processor (architecture); for example, -DADSP-BF535 or -DADSP-BF532. Valid *processorIDs* include ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF535, ADSP-DM102, and AD6532.



The -proc *processorID* command is recommended as a replacement for the -D*processorID* command line to specify target processor.

White space is not permitted between -D and *processorID*. The architecture entry is case sensitive and must be available in your VisualDSP++ installation. This switch must be used if no .LDF file is specified on the command line (see -T). This switch must be used if the specified .LDF file does not specify ARCHITECTURE(). Architectural inconsistency between this switch and the .LDF file causes an error.

Linker Command-Line Reference

-L *path*

Adds *path* name to search libraries and objects. This switch is case sensitive and spacing is unimportant. The *path* parameter enables searching for any file, including the LDF itself. Repeat this switch to add multiple search paths. The paths named with this switch are searched before arguments in the LDF's `SEARCH_DIR{}` command.

-M

Directs the linker to check a dependency and to output the result to `stdout`.

-MM

Directs the linker to check a dependency and to output the result to `stdout` and to perform the build.

-Map *file*

Outputs a map of link symbol information to *file*, which can have any name. The *file* is obligatory and has a `.MAP` extension, provided by the linker. White space is obligatory before *file*; otherwise the link fails.

-MD*macro*[=*def*]

Declares and assigns value *def* to the preprocessor macro named *macro*. For example, `-MDTEST=BAR` executes code following `#ifdef TEST==BAR` in the `.LDF` file (but not code following `#ifdef TEST==XXX`).

If `=def` is not included, *macro* is declared and set to “1”, so code following `#ifdef TEST` is executed. This switch may be repeated.

-S

Omits debugging symbol information (*not* all symbol information) from the output file. Compare with `-s` switch [on page 1-45](#).

-T filename

Uses *filename* to name an `.LDF` file. The `.LDF` file specified following the `-T` switch must contain an `ARCHITECTURE()` command if the command line does not have `-Darchitecture`. The linker requires the `-T` switch when linking for a processor for which no IDDE support has been installed. For example, the processor ID does not appear in the **Target processor** field of the **Project Options** dialog box.

The *filename* must exist and be found (for example, via the `-L` option). There must be white space before *filename*. A file's name is unconstrained, but must be valid. For example, *a.b* works if it is a valid `.LDF` file, where `.LDF` is a valid extension but not a requirement.

-e

Eliminates unused symbols from the executable.

-es sectionName

Names sections (*sectionName* list) to which the elimination algorithm is to be applied. This restricts elimination to the named input sections.

-ev

Eliminates unused symbols and verbose – reports on each eliminated symbol.

-h or -help

Displays a summary of command-line options and exits.

Linker Command-Line Reference

-i *path*

Includes a search directory; directs the preprocessor to append the directory to the search path for include files.

-jcs2l

Directs the linker to convert out-of-range short calls and jumps to the longer form. Refer to **Branch expansion instruction** on the **Link** page.

-jcs2l+

Enables the `-jcs2l` switch and allows the linker to convert out-of-range branches (`-0x800000` to `0x7FFFFFFF`) to indirect calls/jumps sequences using the `p1` register. This is used, for example, when a call from a function in L2 memory is made to a function in L1 memory.

-keep *symbolName*

Retains unused symbols. Directs the linker (when `-e` or `-ev` is enabled) to retain listed symbols in the executable even if they are unused.

-o *filename*

Outputs the executable file with the specified name. If *filename* is not specified, the linker outputs a `.DXE` file in the project's current directory. Alternatively, use the `LDF OUTPUT()` command in the `.LDF` file to name the output file.

-od *filename*

Specifies the value of the `$COMMAND_LINE_OUTPUT_DIRECTORY` LDF macro. This allows you to make a command-line change that propagates to many places without changing the `.LDF` file. Refer to [“Built-In LDF Macros” on page 2-20](#)

-pp

Ends after preprocessing. Stops after the preprocessor runs without linking. The output (preprocessed source code) prints to `stdout`.

-proc *ProcessorID*

Specifies the target processor; for example `-proc ADSP-BF535`
or `-proc ADSP-BF533`.

-s

Strips all symbols. Omits all symbol information from the output file.



Some debugger functionality (including “run to main”) all `stdio` functions, and the ability to stop at the end of program execution rely on the debugger’s ability to locate certain symbols in the executable. This switch removes these symbols.

-sp

Skips preprocessing. Links without preprocessing the `.LDF` file.

-t

Outputs the names of link objects to standard output as the linker processes them.

-v

Verbose. Outputs status information while linking.

-version

Directs the linker to output its version to `stderr` and exit.

Linker Command-Line Reference

-warnonce

Warns only once for each undefined symbol, rather than once for each reference to that symbol.

-xref *filename*

Outputs a list of all cross-referenced symbols (and where they are used) in the link to the named file.

Memory Management Using Overlays

To reduce DSP system costs, many applications employ processors with small amounts of on-chip memory and place much of the program code and data off chip.

ADSP-BF53x processors have an integrated instruction and data cache which can be used to reduce the manual process of moving instructions and data in and out of core. Memory overlays can also be used to run applications efficiently, but require a user-managed set of DMAs.

The linker supports the linking of executables for systems with overlay memory. Applications notes on the Analog Devices Web site provide detailed descriptions of this technique. Since Blackfin processors do not have a separate overlay manager built in, the DMA controller must be used to move code in and out of internal L1 memory. For this reason, it is almost always better to use instruction cache.

This section describes the use of memory overlays with DSPs. The following sections are included:

- [“Memory Overlays” on page 1-48](#)
- [“Overlay Managers” on page 1-49](#)
- [“Memory Overlay Support” on page 1-50](#)
- [“Example – Managing Two Overlays” on page 1-53](#)
- [“Reducing Overlay Manager Overhead” on page 1-60](#)
- [“Using PLIT{} and Overlay Manager” on page 1-64](#)

The following LDF commands facilitate advanced linker features.

- [“OVERLAY_GROUP{}” on page 2-33](#)
- [“PLIT{}” on page 2-37](#)

- [“MEMORY{}” on page 2-28](#)

Memory Overlays

When the built-in caching mechanisms are not used, *memory overlays* support applications that cannot fit the program instructions into the processor’s internal memory. In such cases, program instructions are partitioned and stored in external memory until they are required for program execution. These partitions are memory overlays, and the routines that call and execute them are called *overlay managers*.

Overlays are a “many to one” memory mapping system. Several overlays may “live” (stored) in unique locations in external memory, but “run” (execute) in a common location in internal memory. Throughout the following description, the overlay storage location is referred to as the “live” location, and the internal location where instructions are executed is referred to as the “run” (run-time) space.

Overlay functions are written to *overlay files* (.OVL), which may be specified as one type of linker executable output file. The loader can read .OVL files to generate an .LDR file. On Blackfin processors, this must be done using the memory DMA controller.

[Figure 1-13](#) demonstrates the concept of memory overlays. There are two memory spaces: internal and external. The external memory is partitioned into five overlays. The internal memory contains the main program, an overlay manager function, and two memory segments reserved for execution of overlay program instructions.

In this example, overlays 1 and 2 share the same run-time location within internal memory, and overlays 3, 4, and 5 also share a common run-time memory. When `FUNC_B` is required, the overlay manager loads overlay 2 to the location in internal memory where overlay 2 is designated to run. When `FUNC_D` is required, the overlay manager loads overlay 3 into its designated run-time memory.

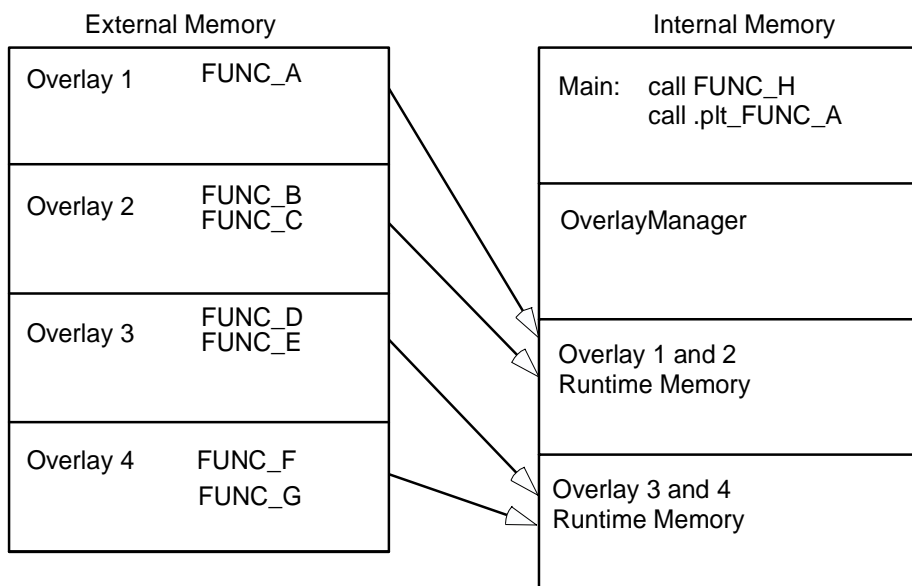


Figure 1-13. Memory Overlays

The transfer is typically implemented with the processor's Direct Memory Access (DMA) capability. The overlay manager can also handle advanced functionality, such as checking whether the requested overlay is already in run-time memory, executing another function while loading an overlay, and tracking recursive overlay function calls.

Overlay Managers

An overlay manager is a user-definable routine responsible for loading a referenced overlay function or data buffer into internal memory (run-time space). This is accomplished with linker-generated constants and `PLIT{}` commands.

Memory Management Using Overlays

Linker-generated constants inform the overlay manager of the overlay's live address, where the overlay resides for execution, and the number of words in the overlay. `PLIT{}` commands inform the overlay manager of the requested overlay and the run-time address of the referenced symbol.

An overlay manager's main objective is to transfer overlays to a run-time location when required. Overlay managers may also:

- Set up a stack to store register values
- Check whether a referenced symbol has already been transferred into its run-time space as a result of a previous reference

If the overlay is already in internal memory, the overlay transfer is bypassed and execution of the overlay routine begins immediately.

- Load an overlay while executing a function from a second overlay (or a non-overlay function)

You may require an overlay manager to perform other specialized tasks to satisfy the special needs of a given application. Overlay managers for Blackfin processors must be developed by the user.

Memory Overlay Support

The overlay support provided by the DSP tools includes:

- Specification of the live and run locations of each overlay
- Generation of constants
- Redirection of overlay function calls to a jump table

Overlay support is partially user-designed in the `.LDF` file (LDF). You specify which overlays share run-time memory and which memory segments establish the live and run space.

[Listing 1-3](#) shows the portion of an .LDF file that defines two overlays. This overlay declaration configures the two overlays to share a common run-time memory space. The syntax for the `OVERLAY_INPUT{}` command is described in [“OVERLAY_GROUP{ }” on page 2-33](#).

`OVLY_one` contains `FUNC_A` and lives in memory segment `M0_ovly`.
`OVLY_two` contains functions `FUNC_B` and `FUNC_C` and also lives in memory segment `M0_ovly`.

Listing 1-3. Overlay Declaration in an .LDF File

```
.dxcode
{ OVERLAY_INPUT {
    OVERLAY_OUTPUT (OVLY_one.ovl)
    INPUT_SECTIONS (FUNC_A.doj(sec_code))
} >ovl_code

OVERLAY_INPUT {
    OVERLAY_OUTPUT (OVLY_two.ovl)
    INPUT_SECTIONS (FUNC_B.doj(sec_code) FUNC_C.doj(sec_code))
} >ovl_code
} >sec_code
```

The common run-time location shared by overlays `OVLY_one` and `OVLY_two` is within the `sec_code` memory segment.

The .LDF file configures the overlays and provides the information necessary for the overlay manager to load the overlays. The information includes the following linker-generated overlay constants (where `#` is the overlay ID):


```
_ov_startaddress_#
_ov_endaddress_#
_ov_size_#
_ov_word_size_run_#
```

Memory Management Using Overlays

```
_ov_word_size_live_#  
_ov_runtimestartaddress_#
```

Each overlay has a word size and an address, which is used by the overlay manager to determine where the overlay resides and where it is executed. One exception, `_ov_size_#`, specifies the total size in bytes.

Overlay live and run word sizes differ when internal memory and external memory widths differ. A system containing 16-bit-wide external memory requires data packing to store an overlay containing instructions.

 The Blackfin processor architecture supports byte addressing using 16-bit, 32-bit, or 64-bit opcodes. Thus, no data packing is required.

Redirection. In addition to providing constants, the linker replaces overlay symbol references to the overlay manager within your code. Redirection is accomplished using a *procedure linkage table* (PLIT). A PLIT is essentially a jump table that executes user-defined code and then jumps to the overlay manager. The linker replaces an overlay symbol reference (function call) with a jump to a location in the PLIT.

You must define PLIT code within the `.LDF` file. This code prepares the overlay manager to handle the overlay that contains the referenced symbol. The code initializes registers to contain the overlay ID and the referenced symbol's run-time address.

[Listing 1-4](#) is an example PLIT definition from an `.LDF` file, where register `R0` is set to the value of the overlay ID that contains the referenced symbol and register `R1` is set to the run-time address of the referenced symbol. The last instruction branches to the overlay manager that uses the initialized registers to determine which overlay to load (and where to jump to execute the called overlay function).

Listing 1-4. PLIT Definition in LDF

```

PLIT
{
    R0.h = PLIT_SYMBOL_OVERLAYID;
    R0.l = PLIT_SYMBOL_OVERLAYID;
    R1.h = PLIT_SYMBOL_ADDRESS;
    R1.l = PLIT_SYMBOL_ADDRESS;
    JUMP OverlayManager;
}

```

The linker expands the PLIT definition into individual entries in a table. An entry is created for each overlay symbol as shown in [Listing 1-5 on page 1-55](#). The redirection function calls the PLIT table for overlays 1 and 2. For each entry, the linker replaces the generic assembly instructions with specific instructions (where applicable).

For example, the first PLIT entry in [Listing 1-3 on page 1-51](#) is for the overlay symbol `FUNC_A`. The linker replaces the constant name `PLIT_SYMBOL_OVERLAYID` with the ID of the overlay containing `FUNC_A`. The linker also replaces the constant name `PLIT_SYMBOL_ADDRESS` with the run-time address of `FUNC_A`.

When the overlay manager is called via the jump instruction of the PLIT table, `R0` contains the referenced function's overlay ID and `R1` contains the referenced function's run-time address. The overlay manager uses the overlay ID and run-time address to load and execute the referenced function.

Example – Managing Two Overlays

The following example has two overlays – each contains two functions. Overlay 1 contains the functions `fft_first_two_stages` and `fft_last_stage`. Overlay 2 contains functions `fft_middle_stages` and `fft_next_to_last`.

Memory Management Using Overlays

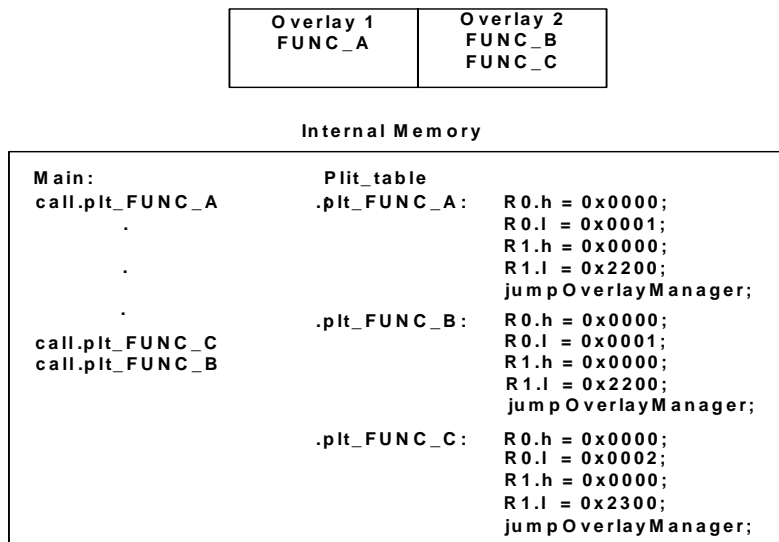


Figure 1-14. Expanded PLIT Table

For examples of overlay manager source code, refer to the example programs shipped with the development software.

The overlay manager:

- Creates and maintains a stack for the registers it uses
- Determines whether the referenced function is in internal memory
- Sets up a DMA transfer
- Executes the referenced function

Several code segments for the LDF and the overlay manager are displayed and explained next.

Listing 1-5. FFT Overlay Example 1

```

OVERLAY_INPUT
{
    ALGORITHM (ALL_FIT)
    OVERLAY_OUTPUT (fft_one.ovl)
    INPUT_SECTIONS ( Fft_1st_last.doj(program) )
} > ovl_code    // Overlay to live in section ovl_code
OVERLAY_INPUT
{
    ALGORITHM (ALL_FIT)
    OVERLAY_OUTPUT (fft_two.ovl)
    INPUT_SECTIONS ( Fft_mid.doj(program) )
} > ovl_code    // Overlay to live in section ovl_c

```

The two defined overlays (`fft_one.ovl` and `fft_two.ovl`) live in memory segment `ovl_code` (defined by the `MEMORY{}` command), and run in section `program`. All instruction and data defined in the `program` memory segment within the `Fft_1st_last.doj` file are part of the `fft_one.ovl` overlay. All instructions and data defined in `program` within the file `Fft_mid.doj` are part of overlay `fft_two.ovl`. The result is two functions within each overlay.

The first and the last called functions are in overlay `fft_one`. The two middle functions are in overlay `fft_two`. When the first function (`fft_one`) is referenced during code execution, overlay `id=1` is transferred to internal memory. When the second function (`fft_two`) is referenced, overlay `id=2` is transferred to internal memory. When the third function (in overlay `fft_two`) is referenced, the overlay manager recognizes that it is already in internal memory and an overlay transfer does not occur.

Memory Management Using Overlays

To verify whether an overlay is in internal memory, place the overlay ID of this overlay into a register (for example, P0) and compare this value to the overlay ID of each overlay already loaded by loading these overlay values into a register (for example, R1).

```
                /* Is overlay already in internal memory? */  
CC = p0 == p1;  
                /* If so, do not transfer it in. */  
if CC jump skipped_DMA_setup;
```

Finally, when the last function (fft_one) is referenced, overlay id=1 is again transferred to internal memory for execution.

The following code segment calls the four FFT functions.

```
fftrad2:  
    call fft_first_2_stages;  
    call fft_middle_stages;  
    call fft_next_to_last;  
    call fft_last_stage;  
wait:  
    NOP;  
    jump wait;
```

The linker replaces each overlay function call with a call to the appropriate entry in the PLIT. For this example, only three instructions are placed in each entry of the PLIT, as follows.

```
PLIT  
{  
    R0.h = PLIT_SYMBOL_OVERLAYID;  
    R0.l = PLIT_SYMBOL_OVERLAYID;  
    R1.h = PLIT_SYMBOL_ADDRESS;  
    R1.l = PLIT_SYMBOL_ADDRESS;  
    JUMP OverlayManager;  
}
```


Register `R0` contains the overlay ID that contains the referenced symbol, and register `R1` contains the run-time address of the referenced symbol. The final instruction jumps the program counter (PC) to the starting address of the overlay manager. The overlay manager uses the overlay ID in conjunction with the overlay constants generated by the linker to transfer the proper overlay into internal memory. Once the transfer is complete, the overlay manager sends the PC to the address of the referenced symbol stored in `R1`.

Linker-Generated Constants

The following constants, generated by the linker, are used by the overlay manager.

```
.EXTERN _ov_startaddress_1;
.EXTERN _ov_startaddress_2;
.EXTERN _ov_endaddress_1;
.EXTERN _ov_endaddress_2;
.EXTERN _ov_size_1;
.EXTERN _ov_size_2;
.EXTERN _ov_word_size_run_1;
.EXTERN _ov_word_size_run_2;
.EXTERN _ov_word_size_live_1;
.EXTERN _ov_word_size_live_2;
.EXTERN _ov_runtimestartaddress_1;
.EXTERN _ov_runtimestartaddress_2;
```

The constants provide the following information to the overlay manager.

- Overlay sizes (both run-time word sizes and live word sizes)
- Starting address of the live space
- Starting address of the run space

Overlay Word Sizes

Memory Management Using Overlays

Each overlay has a word size and an address, which the overlay manager uses to determine where the overlay resides and where it is executed.

These are the linker-generated constants.

```
_ov_startaddress_1      = 0x00000000
_ov_startaddress_2      = 0x00000010
_ov_endaddress_1        = 0x0000000F
_ov_endaddress_2        = 0x0000001F
_ov_word_size_run_1     = 0x00000010
_ov_word_size_run_2     = 0x00000010
_ov_word_size_live_1    = 0x00000010
_ov_word_size_live_2    = 0x00000010
_ov_runtimestartaddress_1 = 0xF0001000
_ov_runtimestartaddress_2 = 0xF0001000
```

The overlay manager places the constants in arrays as shown below. The arrays are referenced by using the overlay ID as the index to the array. The index or ID is stored in a modify (M#) register, and the beginning address of the array is stored in the index (I#) register.

```
.VAR liveAddresses[2] = _ov_startaddress_1,
                        _ov_startaddress_2;
.VAR runAddresses[2]  = _ov_runtimestartaddress_1,
                        _ov_runtimestartaddress_2;
.VAR runWordSize[2]   = _ov_word_size_run_1,
                        _ov_word_size_run_2;
.VAR liveWordSize[2]  = _ov_word_size_live_1,
                        _ov_word_size_live_2;
```

Storing Overlay ID

The overlay manager also stores the ID of an overlay that is currently in internal memory. When an overlay is transferred to internal memory, the overlay manager stores the overlay ID in internal memory in the buffer labeled `ov_id_loaded`. Before another overlay is transferred, the overlay manager compares the required overlay ID with that stored in the

`ov_id_loaded` buffer. If they are equal, the required overlay is already in internal memory and a transfer is not required. The PC is sent to the proper location to execute the referenced function. If they are not equal, the value in `ov_id_loaded` is updated and the overlay is transferred into its internal run space via DMA.

On completion of the transfer, the overlay manager restores register values from the run-time stack, flushes the cache, and then jumps the PC to the run-time location of the referenced function. It is very important to flush the cache before jumping to the referenced function because when code is replaced or modified, incorrect code execution may occur if the cache is not flushed. If the program sequencer searches the cache for an instruction and an instruction from the previous overlay is in the cache, the cached instruction may be executed rather than receiving the expected cache miss.

Summary

In summary, the overlay manager routine does the following.

- Maintains a run-time stack for registers being used by the overlay manager
- Compares the requested overlay's ID with that of the previously loaded overlay (stored in the `ov_id_loaded` buffer)
- Sets up the DMA transfer of the overlay (if it is not already in internal memory)
- Jumps the PC to the run-time location of the referenced function

These are the basic tasks that are performed by an overlay manager. More sophisticated overlay managers may be required for individual applications.

Reducing Overlay Manager Overhead

The example in this section incorporates the ability to transfer one overlay to internal memory while the core executes a function from another overlay. Instead of the core sitting idle while the overlay DMA transfer occurs, the core enables the DMA, and then begins executing another function.

This example uses the concept of overlay function loading and executing. A function `load` is a request to load the overlay function into internal memory but not execute the function. A function `execution` is a request to execute an overlay function that may or may not be in internal memory at the time of the execution request. If the function is not in internal memory, a transfer must occur before execution.

There are several circumstances under which an overlay transfer can be in progress while the core is executing another task. Each circumstance can be labeled as *deterministic* or *non-deterministic*. A deterministic circumstance is one where you know exactly when an overlay function is required for execution. A non-deterministic circumstance is one where you cannot predict when an overlay function is required for execution. For example, a deterministic application may consist of linear flow code except for function calls. A non-deterministic example is an application with calls to overlay functions within an interrupt service routine where the interrupt occurs randomly.

The example provided by the software contains deterministic overlay function calls. The time of overlay function execution requests are known as the number of cycles required to transfer an overlay. Therefore, an overlay function load request can be placed such that the transfer is complete by the time the execution request is made. The next overlay transfer (from a load request) can be enabled by the core and the core can execute the instructions leading up to the function execution request.

Since the linker handles all overlay symbol references in the same way (jump to PLIT table then overlay manager), it is up to the overlay manager to distinguish between a symbol reference requesting the load of an

overlay function and a symbol reference requesting the execution of an overlay function. In the example, the overlay manager uses a buffer in memory as a flag to indicate whether the function call (symbol reference) is a load or an execute request.

The overlay manager first determines whether the referenced symbol is in internal memory. If not, it sets up the DMA transfer. If the symbol is not in internal memory and the flag is set for execution, the core waits for the transfer to complete (if necessary) and then executes the overlay function. If the symbol is set for load, the core returns to the instructions immediately following the location of the function load reference.

Every overlay function call requires initializing the load/execute flag buffer. Here, the function calls are delayed branch calls. The two slots in the delayed branch contain instructions to initialize the flag buffer. Register P0 is set to the value that is placed in the flag buffer, and the value in P0 is stored in memory; 1 indicates a load, and 0 indicates an execution call. At each overlay function call, the load buffer **must** be updated.

The following code is from the main FFT subroutine. Each of the four function calls are execution calls so the pre-fetch (load) buffer is set to zero. The flag buffer in memory is read by the overlay manager to determine if the function call is a load or an execute.

```

    R0 = 0 (Z);
    p0.h = prefetch;
    p0.l = prefetch;
    [P0] = R0;
call  fft_first_2_stages;
    R0 = 0 (Z);
    p0.h = prefetch;
    p0.l = prefetch;
    [P0] = R0;
call  fft_middle_stages;
    R0 = 0 (Z);
    p0.h = prefetch;
```

Memory Management Using Overlays

```
p0.l = prefetch;  
[P0] = R0;  
call fft_next_to_last;  
R0 = 0 (Z);  
p0.h = prefetch;  
p0.l = prefetch;  
[P0] = R0;  
call fft_last_stage;
```

The next set of instructions represents a load function call.

```
R0 = 1 (Z);  
p0.h = prefetch;  
p0.l = prefetch;  
[P0] = R0;  
    /* Set prefetch flag to 1 to indicate a load */  
call fft_middle_stages;  
    /* Pre-loads the function into the      */  
    /* overlay run memory.    */
```

The code executes the first function and transfers the second function and so on. In this implementation, each function resides in a unique overlay and requires two run-time locations. While one overlay loads into one run-time location, a second overlay function executes in another run-time location.

The following code segment allocates the functions to overlays and forces two run-time locations.

```
OVERLAY_GROUP1 {  
    OVERLAY_INPUT  
    {  
        ALGORITHM(ALL_FIT)  
        OVERLAY_OUTPUT(fft_one.ovl)  
        INPUT_SECTIONS( Fft_ovl.doj (program) )  
    } >ovl_code // Overlay to live in section ovl_code
```

```

OVERLAY_INPUT
{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS( Fft_ovl.doj (program) )
} >ovl_code // Overlay to live in section ovl_code
} > mem_code

OVERLAY_MGR {
    INPUT_SECTIONS(ovly_mgr.doj(pm_code))
} > mem_code

OVERLAY_GROUP2 {
    OVERLAY_INPUT
    {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_two.ovl)
        INPUT_SECTIONS( Fft_ovl.doj(program) )
    } >ovl_code // Overlay to live in section ovl_code
    OVERLAY_INPUT
    {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_last.ovl)
        INPUT_SECTIONS( Fft_ovl.doj(program) )
    } >ovl_code // Overlay to live in section ovl_code
    } > mem_code

```

The first and third overlays share one run-time location and the second and fourth overlays share the second run-time location.


Additional instructions are included to determine whether function call is a load or an execution call. If the function call is a load, the overlay manager initiates the DMA transfer and then jumps the PC back to the location where the call was made. If the call is an execution call, the overlay manager determines whether the overlay is currently in internal

Memory Management Using Overlays

memory. If so, the PC jumps to the run-time location of the called function. If the overlay is not in the internal memory, a DMA transfer is initiated and the core waits for the transfer to complete.

The overlay manager pushes the appropriate registers on the run-time stack. It checks whether the requested overlay is currently in internal memory. If not, it sets up the DMA transfer. It then checks whether the function call is a load or an execution call.

If it is a load, it begins the transfer and returns the PC back to the instruction following the call. If it is an execution call, the core is idle until the transfer completes (if the transfer was necessary) and then jumps the PC to the run-time location of the function.

 The overlay managers in these examples are used universally. Specific applications may require some modifications which may allow for the elimination of some instructions. For instance, if your application allows the free use of registers, you may not need a run-time stack.

Using PLIT{} and Overlay Manager

The `PLIT{}` command inserts assembly instructions that handle calls to functions in overlays. The instructions are specific to an overlay and are executed each time a call to a function in that overlay is detected.

Refer to [“PLIT{ }” on page 2-37](#) for basic syntax information. Refer to [“Memory Management Using Overlays” on page 1-47](#) for detailed information on overlays.

[Figure 1-15](#) shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker generates special symbols for overlays. These overlay symbols are:

- `_ov_startaddress_#`
- `_ov_endaddress_#`

- `_ov_size_#`
- `_ov_word_size_run_#`
- `_ov_word_size_live_#`
- `_ov_runtimestartaddress_#`

The `#` indicates the overlay number.



Overlay numbers start at 1 (not 0) to avoid confusion when placing these elements into an array or buffer used by an overlay manager.

The two functions in [Figure 1-15](#) are on different overlays. By default, the linker generates PLIT code only when an unresolved function reference is resolved to a function definition in overlay memory.

The `main` function calls functions `X()` and `Y()`, which are defined in overlay memory. Because the linker cannot resolve these functions locally, the linker replaces the symbols `X` and `Y` with `.plt_X` and `.plt_Y`. Unresolved references to `X` and `Y` are resolved to `.plt_X` and `.plt_Y`.

When the reference and the definition reside in the same executable, the linker does not generate PLIT code. However, you can force the linker to output a PLIT, even when all references can be resolved locally.

The `.plt` command sets up data for the overlay manager, which will first load the overlay that defines the desired symbol, and then branch to that symbol.

Inter-Overlay Calls

PLITs allow you to resolve inter-overlay calls, as shown in [Figure 1-16 on page 1-67](#). Structure the `.LDF` file in such a way that the PLIT code generated for inter-overlay function references is part of the `.plt` section for `main()`, which is stored in non-overlay memory.



Always store the `.plt` section in non-overlay memory.

Memory Management Using Overlays

Non-Overlay Memory

```
main()
{
    int (*pf)() = X;
    Y();
}

/* PLIT & overlay manager handle calls,
   using the PLIT to resolve calls
   and load overlays as needed */

.plt_X: call OM
.plt_Y: call OM

Overlay 1 Storage  → X() {...}      // function X defined
Overlay 2 Storage  → Y() {...}      // function Y defined

Run-time Overlay Memory           // currently loaded overlay
```

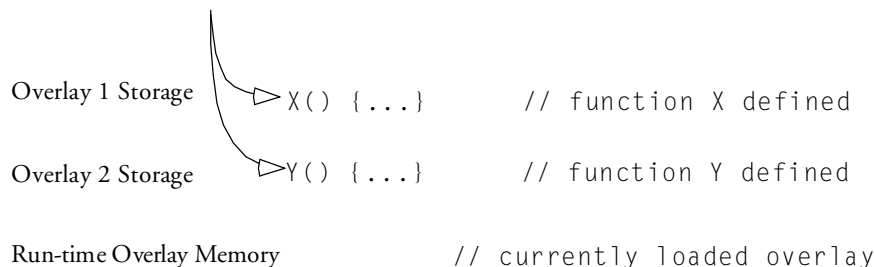


Figure 1-15. PLITs & Overlay Memory; main() Calls to Overlays

The linker resolves all references to variables in overlays, and the PLIT allows an overlay manager to handle the overhead of loading and unloading overlays. Optimize overlays by not placing global variables in overlays. This avoids the difficulty of ensuring the proper overlay is loaded before a global is called.

Inter-Processor Calls

Code in Non-Overlay Memory

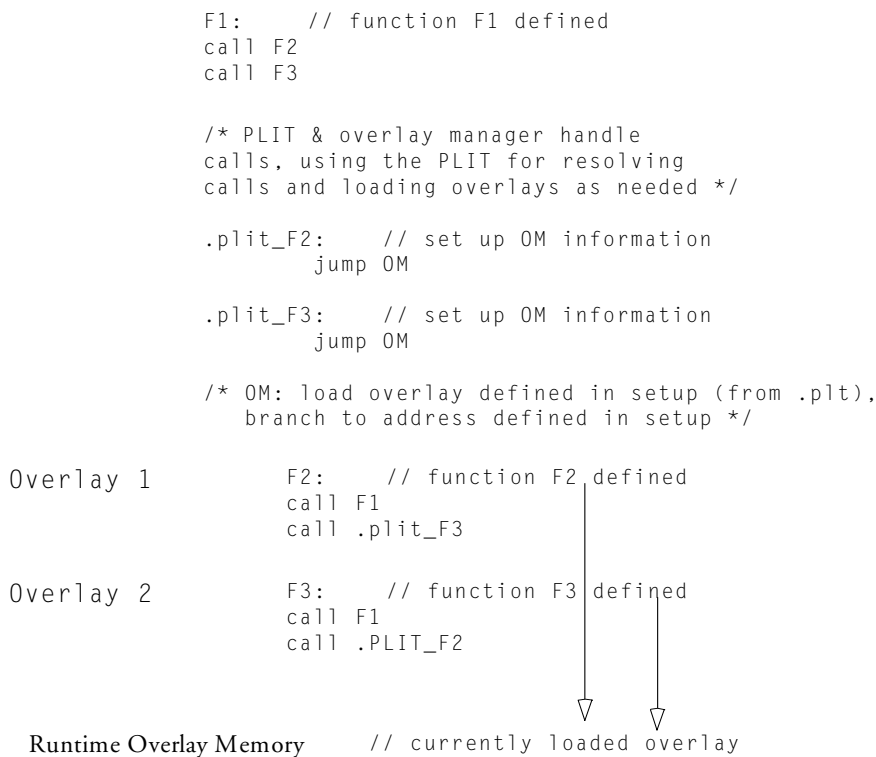



Figure 1-16. PLITs and Overlay Memory – Inter-Overlay Calls

PLITs resolve inter-processor overlay calls, as shown in [Figure 1-17 on page 1-69](#), for systems that permit one processor to access the memory of another processor. When one processor calls into another processor's overlay, the call increases the size of the .plit section in the executable that manages the overlay.

Memory Management Using Overlays

The linker resolves all references to variables in overlays, and the PLIT lets an overlay manager handle the overhead of loading and unloading overlays.

-  Optimize overlays by not putting global variables in overlays. This avoids the difficulty of having to ensure the proper overlay is loaded before a global is referenced.

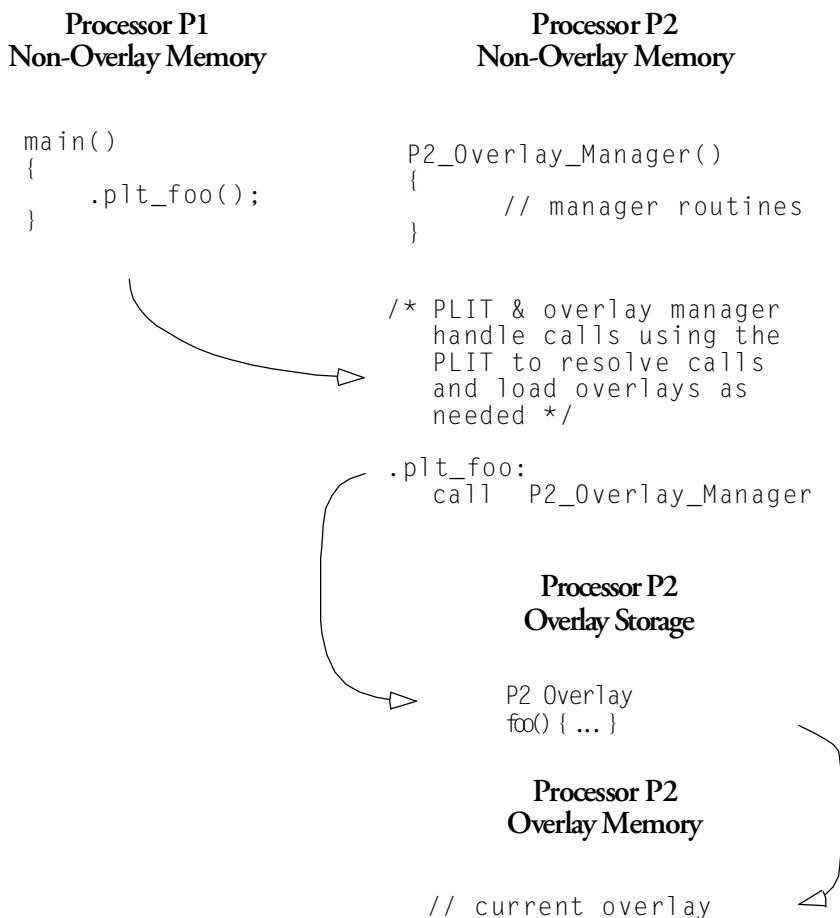


Figure 1-17. PLITs and Overlay Memory – Inter-Processor Calls

2 LINKER DESCRIPTION FILE

Every DSP project requires one Linker Description File (.LDF). The .LDF file specifies precisely how to link projects.



When generating a new .LDF file, use the Expert Linker to generate an .LDF file. Refer to [“Expert Linker” on page 3-1](#) for details.

This chapter is an .LDF file reference and includes:

- [“LDF Guide” on page 2-2](#) – Describes Linker Description File syntax, commands, macros, operators, and programming techniques.
- [“LDF Programming Examples” on page 2-53](#) – Provides example .LDF files for various system architectures.

Chapter 1 describes the linking process and how the .LDF file ties into the linking process.

LDF Guide

The .LDF file allows development of code for any DSP system. It defines your system to the linker and specifies how the linker creates executable code for your system. This section describes .LDF file syntax and provides examples for typical systems.

This section contains:

- [“.LDF File Overview” on page 2-3](#)
- [“LDF Structure” on page 2-10](#)
- [“LDF Expressions” on page 2-12](#)
- [“LDF Keywords, Commands, and Operators” on page 2-13](#)
- [“LDF Operators” on page 2-15](#)
- [“LDF Macros” on page 2-19](#)
- [“LDF Commands” on page 2-22](#)



The linker runs the preprocessor on the LDF, so you can use preprocessor commands (such as `#defines`) within the LDF. For information about preprocessor commands, see the *VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors*.

Assembler section declarations in this document correspond to the Blackfin processor family assembler's `.SECTION` directive.

Refer to [“LDF Programming Examples” on page 2-53](#) and to example DSP programs shipped with VisualDSP++ for example .LDF files supporting typical system models.

.LDF File Overview

The .LDF file directs the linker by mapping code or data to specific memory segments. The linker maps program code (and data) within the system memory and processor(s), assigning an address to every symbol, where:

```
symbol = label  
symbol = function_name  
symbol = variable_name
```

If you neither write an LDF nor import an LDF into your project, VisualDSP++ links the code using a default LDF. The default LDF is based on the processor specified in the VisualDSP++ environment's **Project Options** dialog box. Default .LDF files are packaged with your DSP tool distribution kit in a subdirectory specific to your target processor's family. One default LDF is provided for each DSP supported by your VisualDSP++ installation.

You can use an .LDF file written from scratch. However, modifying an existing LDF (or a default LDF) is often the easier alternative when there are no large changes in your system's hardware or software.

The LDF combines information, directing the linker to place input sections in an executable according to the memory available in the DSP system.



The linker may output warning messages and error messages. You must resolve these messages to enable the linker to produce valid output. See [“Linker Warning and Error Messages” on page 1-21](#) for more information.

Example – Basic .LDF File

[Listing 2-1](#) is an example of a basic .LDF file (formatted for readability). Note the `MEMORY{}` and `SECTIONS{}` commands and refer to [“Notes on Basic .LDF File Example”](#). Other .LDF file examples are provided in [“LDF Programming Examples”](#) on [page 2-53](#).

Listing 2-1. Example .LDF File

```
ARCHITECTURE(ADSP-BF535)
SEARCH_DIR($ADI_DSP\Blackfin\lib)
$OBJECTS = CRT, $COMMAND_LINE_OBJECTS ENDCRT;

MEMORY          /* Define/label system memory      */
{               /* List of global Memory Segments */
    MEM_L2
        { TYPE(RAM) START(0xF0000000) END(0xF002FFFF) WIDTH(8) }
    MEM_HEAP
        { TYPE(RAM) START(0xF0030000) END(0xF0037FFF) WIDTH(8) }
    MEM_STACK
        { TYPE(RAM) START(0xF0038000) END(0xF003DFFF) WIDTH(8) }
    MEM_SYSTACK
        { TYPE(RAM) START(0xF003E000) END(0xF003FDFF) WIDTH(8) }
    MEM_ARGV
        { TYPE(RAM) START(0xF003FE00) END(0xF003FFFF) WIDTH(8) }
}
SECTIONS
{   /* List of sections for processor P0 */

    dx_e_L2
    {
        INPUT_SECTION_ALIGN(2)
        /* Align all code sections on 2 byte boundary */
        INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS( $OBJECTS(constdata))
    }
}
```

```

        $LIBRARIES(constdata))
    INPUT_SECTION_ALIGN(1)
    INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
} >MEM_L2

stack
{
    ldf_stack_space = .;
    ldf_stack_end =
        ldf_stack_space + MEMORY_SIZEOF(MEM_STACK) - 4;
} >MEM_STACK

sysstack
{
    ldf_sysstack_space = .;
    ldf_sysstack_end =
        ldf_sysstack_space + MEMORY_SIZEOF(MEM_SYSSTACK) - 4;
} >MEM_SYSSTACK

heap
{
    /* Allocate a heap for the application */
    ldf_heap_space = .;
    ldf_heap_end =
        ldf_heap_space + MEMORY_SIZEOF(MEM_HEAP) - 1;
    ldf_heap_length = ldf_heap_end - ldf_heap_space;
} >MEM_HEAP

argv
{
    /* Allocate argv space for the application */
    ldf_argv_space = .;
    ldf_argv_end =
        ldf_argv_space + MEMORY_SIZEOF(MEM_ARGV) - 1;
    ldf_argv_length =
        ldf_argv_end - ldf_argv_space;
} >MEM_ARGV

} /* end SECTIONS */

} /* end PROCESSOR p0 */

```

Notes on Basic .LDF File Example

In the following description, the `MEMORY{}` and `SECTIONS{}` commands connect the program to the target DSP. For syntax information on LDF commands, see [“LDF Guide” on page 2-2](#).

These notes describe features of the .LDF file presented in [Listing 2-1](#).

- `ARCHITECTURE(ADSP-BF535)` specifies the target architecture (processorID). This dictates possible memory widths and address ranges, the register set, and other structural information for use by the debugger, linker, and loader. The target architecture must be installed in VisualDSP++.
- `SEARCH_DIR()` specifies directory paths to be searched for libraries and object files. This example’s argument (`$ADI_DSP\Blackfin\lib`) specifies one search directory. For more information, see [“SEARCH_DIR\(\)” on page 2-44](#).

The linker supports a sequence of search directories presented as an argument list (`directory1, directory2, ...`). The linker follows this sequence, stopping at the first match.

- `$OBJECTS` is an example of a user-definable *macro*, which expands to a comma-delimited list. Macros improve readability by replacing long strings of text. Conceptually similar to preprocessor macro support (`#defines`) also available in the .LDF file, string macros are independent. In this example, `$OBJECTS` is synonymous with `$COMMAND_LINE_OBJECTS`, and expands to a comma-delimited list of the input files to be linked.

Note: In this example and in the default .LDF files that accompany VisualDSP++, `$OBJECTS` in the `SECTIONS()` command specifies the object files to be searched for specific input sections.

As another example, `$ADI_DSP` expands to the VisualDSP++ home directory.

- `$COMMAND_LINE_OBJECTS` ([on page 2-20](#)) is an LDF *command-line macro*, which expands at the linker command line into the list of input files. Each linker invocation from the VisualDSP++ IDDE has a command-line equivalent. In the VisualDSP++ IDDE, `$COMMAND_LINE_OBJECTS` represents the `.DOJ` file of every source file in the VisualDSP++ **Project** window.

Note: The order in which the linker processes object files (which affects the order in which addresses in memory segments are assigned to input sections and symbols) is determined by the listed order in the `SECTIONS{}` command. As noted above, this order is typically the order listed in `$OBJECTS ($COMMAND_LINE_OBJECTS)`.

VisualDSP++ uses a linker command line that lists objects in alphabetical order. This order carries through to the `$OBJECTS` macro. You may customize the `.LDF` file to link objects in any desired order. Instead of using default macros such as `$OBJECTS`, each `INPUT_SECTION` command can have one or more explicit object names.

The following examples are functionally identical.

```
dx_program { INPUT_SECTIONS ( main.doj(program)
                             fft.doj(program) ) } > mem_program
```

```
$DOJS = main.doj, fft.doj;
dx_program {
    INPUT_SECTIONS ($DOJS(program))
} >mem_program;
```

- The `MEMORY{}` command ([on page 2-28](#)) defines the target system's physical memory and connects the program to the target system. Its arguments partition the memory into memory segments. Each memory segment is assigned a distinct name, memory type, a start and end address (or segment length), and a memory width. These names occupy different namespaces from input section names and

output section names. Thus, a memory segment and an output section may have the same name. In this example, the memory segment and output section are named as `MEM_L2` and `DXE_L2` because the memory holds both program (program) and data (data1) information.

- Each `PROCESSOR{}` command ([on page 2-42](#)) generates a single executable.
- The `OUTPUT()` command ([on page 2-43](#)) produces an executable (.DXE) file and specifies its file name.

In this example, the argument to the `OUTPUT()` command is the `$COMMAND_LINE_OUTPUT_FILE` macro ([on page 2-21](#)). The linker names the executable according to the text following the `-o` switch (which corresponds to the name specified in the **Project Options** dialog box when the linker is invoked via the VisualDSP++ IDE).

```
>linker ... -o outputfilename
```

`SECTIONS{}` ([on page 2-45](#)) specifies the placement of code and data in physical memory. The linker maps input sections (in object files) to output sections (in executables), and maps the output sections to memory segments specified by the `MEMORY{}` command.

The `INPUT_SECTIONS()` statement specifies the object file the linker uses as an input to resolve the mapping to the appropriate memory segment declared in the `.LDF` file.

The `INPUT_SECTIONS` statement specifies the object file that the linker uses as an input to resolve the mapping to the appropriate `MEMORY` segment declared in the `LDF`. For example, in [Listing 2-1](#), two input sections (`program` and `data1`) are mapped into one memory segment (`L2`), as shown below.

```
dx_e_L2
1  INPUT_SECTIONS_ALIGN (2)
2  INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
3  INPUT_SECTIONS_ALIGN (1)
4  INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
  >MEM_L2
```

- The second line directs the linker to place the object code assembled from the source file's "program" input section (via the ".section program" directive in the assembly source file), place the output object into the "DXE_L2" output section, and map it to the "MEM_L2" memory segment. The fourth line does the same for the input section "data1" and output section "DXE_L2", mapping them to the memory segment "MEM_L2".

The two pieces of code follow each other in the `program` memory segment. The `INPUT_SECTIONS()` commands are processed in order, so the `program` sections appear first, followed by the `data1` sections. The `program` sections appear in the same order as object files appear in the `$OBJECTS` macro.

You may intersperse `INPUT_SECTIONS()` statements within an output section with other directives, including location counter information.

LDF Structure

One way to produce a simple and maintainable .LDF file is to parallel the structure of your DSP system. Using your system as a model, follow these guidelines.

- Split the file into a set of `PROCESSOR{ }` commands, one for each DSP in your system.
- Place a `MEMORY{ }` command in the scope that matches your system, defining memory unique to a processor within the scope of the corresponding `PROCESSOR{ }` command.
- If applicable, place a `SHARED_MEMORY{ }` command in the .LDF file's global scope. This represents system resources available as shared resources.

Declare common (shared) memory definitions in the global scope before the `PROCESSOR{ }` commands. See [“Command Scoping” on page 2-11](#) for more information.

Comments in the .LDF File

C style comments may cross newline boundaries until a `*/` is encountered.

A `//` string precedes a single-line C++ style comment.

For more information on LDF structure, see [“Link Target Description” on page 1-22](#), [“Placing Code on the Target” on page 1-29](#), and [“LDF Programming Examples” on page 2-53](#).

Command Scoping

There are two LDF scopes – global and command.

A *global scope* occurs outside commands. Commands and expressions that appear in the global scope are always available and are visible in all subsequent scopes. LDF macros are available globally, regardless of the scope in which the macro is defined (see “[LDF Macros](#)” on page 2-19).

A *command scope* applies to all commands that appear between the braces ({ }) of another command, such as a `PROCESSOR{}` or `PLIT{}` command. Commands and expressions that appear in the command scopes are limited to those scopes.

[Figure 2-1](#) illustrates some scoping issues. For example, the `MEMORY{}` command that appears in the LDF’s global scope is available in all command scopes, but the `MEMORY{}` command that appear in command scopes are restricted to those scopes.

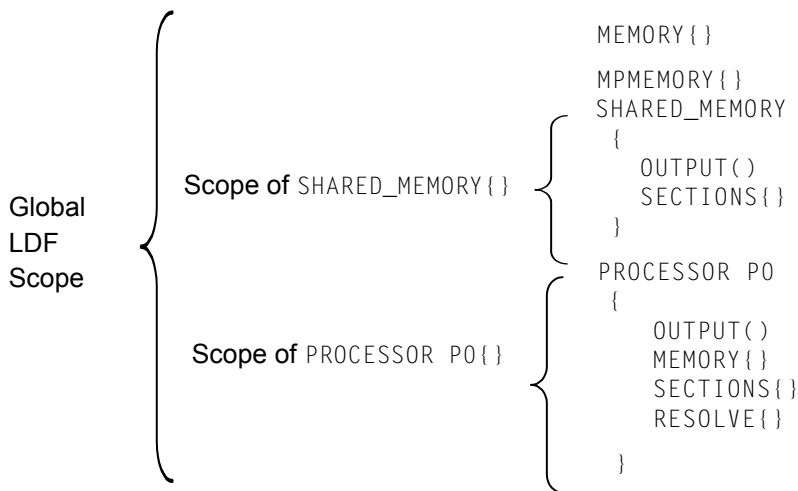


Figure 2-1. LDF Command Scoping Example

LDF Expressions

LDF commands may contain arithmetic expressions that follow the same syntax rules as C/C++ language expressions. The linker:

- Evaluates all expressions as type `unsigned long` and treats constants as type `unsigned long`
- Supports all C/C++ language arithmetic operators
- Allows definitions and references to symbolic constants in the LDF
- Allows reference to global variables in the program being linked
- Recognizes labels that conform to these constraints:
 - Must start with a letter, underscore, or point
 - May contain any letters, underscores, digits, and points
 - Are white space delimited
 - Do not conflict with any keywords
 - Are unique

Table 2-1. Valid Items in Expressions

Convention	Description
<code>.</code>	Current location counter (a period character in an address expression). See “Location Counter (.)” on page 2-18.
<code>0xnumber</code>	Hexadecimal number (a 0x prefix)
<code>number</code>	Decimal number (a number without a prefix)

Table 2-1. Valid Items in Expressions (Cont'd)

Convention	Description
<i>number</i> ^k or <i>number</i> ^K	A decimal number multiplied by 1024
B/#number or b/#number	A binary number

LDF Keywords, Commands, and Operators

Table 2-2 lists .LDF file keywords. Descriptions of LDF keywords, operators, macros, and commands are provided in the following sections.

- [“Miscellaneous LDF Keywords” on page 2-14](#)
- [“LDF Operators” on page 2-15](#)
- [“LDF Macros” on page 2-19](#)
- [“LDF Commands” on page 2-22](#)



Keywords are case sensitive; the linker recognizes a keyword only when the *entire* word is UPPERCASE.

Table 2-2. LDF File Keywords Summary

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BM	BOOT
DEFINED	DM	
ELIMINATE	ELIMINATE_SECTIONS	END
FALSE	FILL	FIRST_FIT
INCLUDE	INPUT_SECTION_ALIGN	INPUT_SECTIONS

Table 2-2. LDF File Keywords Summary (Cont'd)

KEEP	LENGTH	LINK_AGAINST
MAP	MEMORY	MEMORY_SIZEOF
MPMEMORY	NUMBER_OF_OVERLAYS	OUTPUT
OVERLAY_GROUP	OVERLAY_ID	OVERLAY_INPUT (legacy)
OVERLAY_OUTPUT	PACKING	PLIT
	PLIT_SYMBOL_ADDRESS	PLIT_SYMBOL_OVERLAYID
PROCESSOR	RAM	
RESOLVE	RESOLVE_LOCALLY	ROM
SEARCH_DIR	SECTIONS	SHARED_MEMORY
SHT_NOBITS	SIZE	SIZEOF
SROM	START	TYPE
VERBOSE	WIDTH	XREF

Miscellaneous LDF Keywords

The following linker keywords are not operators, macros, or commands.

Table 2-3. Miscellaneous LDF File Keywords

Keyword	Description
FALSE	A constant with a value of 0
TRUE	A constant with a value of 1
XREF	A cross-reference option setting. See “-xref filename” on page 1-46 .

For more information about other .LDF file keywords, see [“LDF Operators” on page 2-15](#), [“LDF Macros” on page 2-19](#) and [“LDF Commands” on page 2-22](#).

LDF Operators

LDF operators in expressions support memory address operations. Expressions that contain these operators terminate with a semicolon, except when the operator serves as a variable for an address. The linker responds to several LDF operators including the location counter.

Each LDF operator is described next.

ABSOLUTE() Operator

Syntax:

`ABSOLUTE(expression)`

The linker returns the value *expression*. Use this operator to assign an absolute address to a symbol. The *expression* can be:

- A symbolic expression in parentheses; for example:

```
ldf_start_expr = ABSOLUTE(start + 8);
```

This example assigns `ldf_start_expr` the value corresponding to the address of the symbol `start`, plus 8, as in:

```
ldf_start_expr = start + 8;
```

- A integer constant in one of these forms: hexadecimal, decimal, or decimal optionally followed by “K” (kilo [x1024]) or “M” (Mega [x1024x1024]).
- A period, indicating the current location (see [“Location Counter \(.\)” on page 2-18](#)).

The following statement, which defines the bottom of stack space in the LDF

```
ldf_stack_space = .;
```

can also be written as:

```
ldf_stack_space = ABSOLUTE(.);
```

- A symbol name

ADDR() Operator

Syntax:

```
ADDR(section_name)
```

This operator returns the start address of the named output section defined in the LDF. Use this operator to assign a section's absolute address to a symbol.

Example

If an .LDF file defines output sections as,

```
dx_L2
{
    INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
}> mem_L2

dx_L2
{
    INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
}> mem_L2
```

the .LDF file may contain the command:

```
ldf_start_L2 = ADDR(dx_L2)
```

The linker generates the constant `ldf_start_L2` and assigns it the start address of the `dx_L2` output section.

DEFINED() Operator

Syntax:

DEFINED(*symbol*)

The linker returns a 1 when the symbol appears in the global symbol table, and returns 0 when the symbol is not defined. Use this operator to assign default values to symbols.

Example

If an assembly object linked by the .LDF file defines the global symbol `test`, the following statement sets the `test_present` constant to 1. Otherwise, the constant has the value 0.

MEMORY_SIZEOF() Operator

Syntax:

MEMORY_SIZEOF(*segment_name*)

This operator returns the size (in words) of the named memory segment. Use this operator when a segment's size is required in order to move the current location counter to an appropriate location.

Example

This example (from a default .LDF file) sets a linker-generated constant based on the location counter plus the MEMORY_SIZEOF operator.

```
sec_stack {
    ldf_stack_limit = .;
    ldf_stack_base = . + MEMORY_SIZEOF(mem_stack) - 1;
} > mem_stack
```

The `sec_stack` section is defined to consume the entire `mem_stack` memory segment.

SIZEOF() Operator

Syntax:

`SIZEOF(section_name)`

This operator returns the size (in bytes) of the named output section. Use this operator when a section's size is required in order to move the current location counter to an appropriate memory location.

Example

The following LDF fragment defines the `_sizeofdata1` constant to the size of the `data1` section.

```
data1
{
    INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
    _sizeofdata1 = SIZEOF(data1);
} > MEM_DATA1
```

Location Counter (.)

The linker treats a “.” (period surrounded by spaces) as the symbol for the current location counter. The *location counter* is a pointer to the memory location at the end of the output section. Because the period refers to a location in an output section, this operator may appear only within an output section in a `SECTIONS{ }` command.

Observe these rules.

- Use a period anywhere a symbol is allowed in an expression.
- Assigning a value to the period operator moves the location counter, leaving voids or gaps in memory.
- The location counter may not be decremented.

LDF Macros

LDF macros (or *linker macros*) are built-in macros. They have predefined system-specific procedures or values. Other macros, called *user macros*, are user-definable.

LDF macros are identified by a leading dollar sign (\$) character. Each LDF macro is a name for a text string. You may assign LDF macros with textual or procedural values, or they may simply be declared to exist.

The linker:

- Substitutes the string value for the name. Normally the string value is longer than the name, so the macro expands to its textual length.
- Performs actions conditional on the existence of (or value of) the macro.
- Assigns a value to the macro, possibly as the result of a procedure, and uses that value in further processing.

LDF macros funnel input from the linker command line into predefined macros and provide support for user-defined macro substitutions. Linker macros are available globally in the LDF, regardless of where they are defined. For more information, see [“Command Scoping” on page 2-11](#) and [“LDF Macros and Command-Line Interaction” on page 2-22](#).



LDF macros are independent of preprocessor macro support, which is also available in the LDF. The preprocessor places preprocessor macros (or other preprocessor commands) into source files. Preprocessor macros repeat instruction sequences in your source code or define symbolic constants. These macros facilitate text

replacement, file inclusion, and conditional assembly and compilation. For example, the assembler's preprocessor uses the `#define` command to define macros and symbolic constants.

Refer to the *VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors* and the *VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors* for more information.

Built-In LDF Macros

The linker provides the following LDF macros.

- `$COMMAND_LINE_OBJECTS`

This macro expands into the list of object (`.DOJ`) and library (`.DLB`) files that are input on the linker's command line. Use this macro within the `INPUT_SECTIONS()` syntax of the linker's `SECTIONS{}` command. This macro provides a comprehensive list of object file input that the linker searches for input sections.

- `$COMMAND_LINE_LINK_AGAINST`

This macro expands into the list of executable (`.DXE` or `.SM`) files input on the linker's command line. For multiprocessor links, this macro is useful within the `RESOLVE()` and `PLIT{}` syntax of the

linker's `SECTIONS{ }` command. This macro provides a comprehensive list of executable file input that the linker searches to resolve external symbols.

`$COMMAND_LINE_OUTPUT_FILE`

This macro expands into the output executable file name, which is set with the linker's `-o` switch. This file name corresponds to the `<projectname.dxe>` set via the VisualDSP++ IDDE's **Project Options** dialog box. Use this macro only once in your LDF for file name substitution within an `OUTPUT()` command.

- `$COMMAND_LINE_OUTPUT_DIRECTORY`

This macro expands into the path of the output directory, which is set with the linker's `-od` switch (or `-o` switch when `-od` is not specified). For example, the following statement permits a configuration change (Release vs. Debug) without modifying the `.LDF` file.

`OVERLAY_OUTPUT ($COMMAND_LINE_OUTPUT_DIRECTORY\OVL1.OVL)`

- `$ADI_DSP`

This macro expands into the path of the VisualDSP++ installation directory. Use this macro to control how the linker searches for files.

User-Declared Macros

The linker supports user-declared macros for file lists. The following syntax declares `$macroname` as a comma-delimited list of files.

`$macroname = file1, file2, file3, ... ;`

After `$macroname` has been declared, the linker substitutes the file list when `$macroname` appears in the `.LDF` file. Terminate a `$macroname` declaration with a semicolon. The linker processes the files in the listed order.

LDF Macros and Command-Line Interaction

The linker receives commands through a command-line interface regardless whether the linker runs automatically from the VisualDSP++ IDDE or explicitly from a command window.

Many linker operations, such as input and output, are controlled through the command line entries. Use LDF macros to apply command-line inputs within an `.LDF` file.

Base your decision whether to use command-line inputs in the `.LDF` file or to control the linker with LDF code on the following considerations.

- An `.LDF` file that uses command-line inputs produces a more generic LDF that can be used in multiple projects. Because the command line can specify only one output, an `.LDF` file that relies on command-line input is best suited for single-processor systems.
- An `.LDF` file that does not use command-line inputs produces a more specific LDF that can control complex linker features.

LDF Commands

Commands in the `.LDF` file (called LDF commands) define the target system and specify the order in which the linker processes output for that system. LDF commands operate within a scope, influencing the operation of other commands that appear within the range of that scope. [For more information, see “Command Scoping” on page 2-11.](#)

The linker supports these LDF commands:

- [“ALIGN\(\)” on page 2-23](#)
- [“ARCHITECTURE\(\)” on page 2-24](#)
- [“ELIMINATE\(\)” on page 2-24](#)
- [“ELIMINATE_SECTIONS\(\)” on page 2-25](#)

- “INCLUDE()” on page 2-25
- “INPUT_SECTION_ALIGN()” on page 2-25
- “KEEP()” on page 2-27
- “LINK_AGAINST()” on page 2-27
- “MAP()” on page 2-28
- “MEMORY{}” on page 2-28
- “MPMEMORY{}” on page 2-31
- “OVERLAY_GROUP{}” on page 2-33
- “PLIT{}” on page 2-37
- “PROCESSOR{}” on page 2-42
- “RESOLVE()” on page 2-44
- “SEARCH_DIR()” on page 2-44
- “SECTIONS{}” on page 2-45
- “SHARED_MEMORY{}” on page 2-50

ALIGN()

The LDF `ALIGN(number)` command aligns the address of the current location counter to the next address that is a multiple of *number*, where *number* is a power of 2.

number is a word boundary (address) that depends on the word size of the memory segment in which the `ALIGN()` takes place.

ARCHITECTURE()

The `ARCHITECTURE()` command specifies the target system's processor. An `.LDF` file may contain one `ARCHITECTURE()` command only. The `ARCHITECTURE` command must appear with global LDF scope, applying to the entire `.LDF` file.

The command's syntax is:

```
ARCHITECTURE(processorID)
```

The `ARCHITECTURE()` command is case sensitive. Valid entries include `ADSP-BF535`, `ADSP-BF531`, `ADSP-BF532`, `ADSP-BF533`, `ADSP-DM102`, and `AD6532`. Thus, `ADSP-BF535` is valid, but `adsp-BF535` is not valid.

If the `ARCHITECTURE()` command does not specify the target processor, you must identify the target processor via the linker command line (`linker -proc processorID ...`). Otherwise, the linker cannot link the program. If processor-specific `MEMORY{}` commands in the `.LDF` file conflict with the processor type, the linker issues an error message and halts.



Test whether your VisualDSP++ installation accommodates a particular processor by typing the following linker command.

```
linker -proc processorID
```

If the architecture is not installed, the linker prints a message to that effect.

ELIMINATE()

The LDF `ELIMINATE()` command enables object elimination, which removes symbols from the executable if they are not called. Adding the `VERBOSE` keyword, `ELIMINATE(VERBOSE)`, reports on objects as they are eliminated. This command performs the same function as the linker's `-e` command-line switch.

ELIMINATE_SECTIONS()

The LDF `ELIMINATE_SECTIONS(sectionList)` command enables section elimination, which removes symbols from listed sections only, if they are not called. `sectionList` is a comma-delimited list of sections. Verbose elimination is obtained by specifying `ELIMINATE_SECTIONS(VERBOSE)`. This command performs the same function as the linker's `-es` command-line switch.

INCLUDE()

The LDF `INCLUDE()` command specifies an additional `.LDF` files which the linker processes before processing the remainder of the current LDF. Specify any number of additional `.LDF` files. Supply one file name per `INCLUDE()` command.

Each `.LDF` file must specify the same `ARCHITECTURE()`, though only one `.LDF` file is obligated to specify an architecture. Normally it is the top-level `.LDF` file which includes the other `.LDF` files.

INPUT_SECTION_ALIGN()

The LDF `INPUT_SECTION_ALIGN(number)` command aligns each input section (data or instruction) in an output section to an address satisfying *number*. *number*, which must be a power of 2, is a word boundary (address). Valid values for *number* depend on the word size of the memory segment receiving the output section being aligned.

The linker fills holes created by `INPUT_SECTION_ALIGN()` commands with zeros (by default), or with the value specified with the preceding `FILL` command valid for the current scope. See `FILL` under [“SECTIONS{}” on page 2-45](#)

The `INPUT_SECTION_ALIGN()` command is valid only within the scope of an output section. For more information, see [“Command Scoping” on page 2-11](#). For more information on output sections, see the syntax description for [“SECTIONS{” on page 2-45](#).

Example

In this example, input sections from `a.doj`, `b.doj`, and `c.doj` are aligned on even addresses. Input sections from `d.doj` and `e.doj` are *not* quad-word aligned because `INPUT_SECTION_ALIGN(1)` indicates subsequent sections are not subject to input section alignment.

```
SECTIONS
{
    program
    {
        INPUT_SECTION_ALIGN(2)

        INPUT_SECTIONS ( a.doj(program))
        INPUT_SECTIONS ( b.doj(program))
        INPUT_SECTIONS ( c.doj(program))

        // end of alignment directive for input sections
        INPUT_SECTION_ALIGN(1)

        // The following sections will not be aligned.
        INPUT_SECTIONS ( d.doj(data1))
        INPUT_SECTIONS ( e.doj(data1))

    } >MEM_PROGRAM
}
```


KEEP()

The linker uses the LDF `KEEP(keepList)` command when section elimination is enabled, retaining the listed objects in the executable even when they are not called. *keepList* is a comma-delimited list of objects to be retained.

LINK_AGAINST()

The LDF `LINK_AGAINST()` command checks specific executables to resolve variables and labels that have not been resolved locally.



To link programs for multiprocessor systems, you must use the `LINK_AGAINST()` command in the `.LDF` file.

This command is an optional part of `PROCESSOR{}`, `SHARE_MEMORY{}`, and `OVERLAY_INPUT{}` commands. The syntax of the `LINK_AGAINST()` command (as part of a `PROCESSOR{}` command) is:

```
PROCESSOR Pn
{
    ...
    LINK_AGAINST (executable_file_names)
    ...
}
```

where:

- *Pn* is the processor name; for example, `P0` or `P1`.
- *executable_file_names* is a list of one or more executable (`.DxE`) or shared memory (`.SM`) files. Separate multiple file names with white space.

The linker searches the executable files in the order specified in the `LINK_AGAINST()` command. When a symbol's definition is found, the linker stops searching.

Override the search order for a specific variable or label by using the `RESOLVE()` command (see [“RESOLVE\(\)” on page 2-44](#)), which directs the linker to ignore `LINK_AGAINST()` for a specific symbol. `LINK_AGAINST()` for other symbols still applies. Example LDFs that contain the `LINK_AGAINST()` and `RESOLVE()` commands are available in [Listing 2-6 on page 2-56](#).

MAP()

The LDF `MAP(filename)` command outputs a map file (`.MAP`) with the specified name. You must supply a file name. Place this command anywhere in the LDF.

The `MAP(filename)` command corresponds to and is overridden by the linker’s `-Map <filename>` command-line switch. In VisualDSP++, if a project’s options (**Link** page of **Project Options** dialog box) specify the generation of a symbol map, the linker runs with `-Map <projectname>.map` asserted, and the LDF `MAP()` command generates a warning.

MEMORY{}

The LDF `MEMORY{ }` command specifies the memory map for the target system. After declaring memory segment names with this command, use the memory segment names to place program sections via the `SECTIONS{ }` command.

The LDF must contain a `MEMORY{ }` command for global memory on the target system and may contain a `MEMORY{ }` command that applies to each processor’s scope. There is no limit to the number of memory segments you can declare within each `MEMORY{ }` command. [For more information, see “Command Scoping” on page 2-11.](#)

In each scope scenario, follow the `MEMORY{ }` command with a `SECTIONS{ }` command. Use the memory segment names to place program sections. Only memory segment declarations may appear within the `MEMORY{ }` command. There is no limit to section name lengths.

If you do not specify the target processor's memory map with the `MEMORY{}` command, the linker cannot link your program. If the combined sections directed to a memory segment require more space than exists in the segment, the linker issues an error message and halts the link.

The syntax for the `MEMORY{}` command appears in [Figure 2-2](#), followed by a description of each part of a *segment_declaration*.

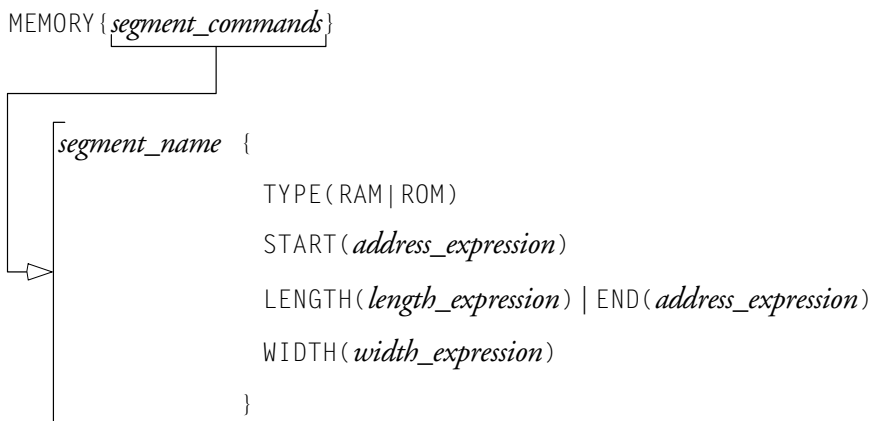


Figure 2-2. Syntax Tree of the `MEMORY{}` Command

Segment Declarations

segment_declaration declares a memory segment on the target processor. Although an LDF may contain only one `MEMORY{}` command that applies to all scopes, there is no limit to the number of memory segments declared within a `MEMORY{}` command.

Each *segment_declaration* must contain a *segment_name*, `TYPE()`, `START()`, `LENGTH()` or `END()`, and a `WIDTH()`. [Table 2-4](#) describes the make-up of a segment declaration.

Table 2-4. Parts of a Segment Declaration

Item	Description
<i>segment_name</i>	Identifies the memory region. <i>segment_name</i> must start with a letter, underscore, or point, and may include any letters, underscores, digits, and points, and must not conflict with LDF keywords.
TYPE()	Identifies the architecture-specific type of memory within the memory segment. (Note: not all target processors support all types of memory.) The linker stores this information in the executable for use by other development tools. Specify the functional or hardware locus (RAM or ROM). The RAM declarator specifies segments that need to be booted. ROM segments are not booted; they are executed/loaded directly from off-chip PROM space.
START(<i>address_number</i>)	Specifies the memory segment's start address. The <i>address_number</i> must be an absolute address.
LENGTH(<i>length_number</i>) or END(<i>address_number</i>)	Identifies the length of the memory segment (in words) or specifies the segment's end address. When stating the length, <i>length_number</i> is the number of addressable words within the region or an expression that evaluates to the number of words. When stating the end address, <i>address_number</i> is an absolute address.
WIDTH(<i>width_number</i>)	Identifies the physical width (number of bits) of the on-chip or off-chip memory interface. <i>width_number</i> must be a number. For Blackfin processors, width must be 16.

MPMEMORY{}

The LDF `MPMEMORY{}` command specifies the offset of each processor's physical memory in a multiprocessor target system. After declaring the processor names and memory segment offsets with the `MPMEMORY{}` command, the linker uses the offsets during multiprocessor linking. Refer to [“Memory Overlay Support” on page 1-50](#) for a detailed description of overlay functionality.

Your .LDF file (and other .LDF files that it includes), may contain one `MPMEMORY{}` command only. The maximum number of processors that can be declared is architecture-specific. Follow the `MPMEMORY{}` command with `PROCESSOR processor_name{}` commands, which contain each processor's `MEMORY{}` and `SECTIONS{}` commands.

[Figure 2-3](#) shows `MPMEMORY{}` command syntax.

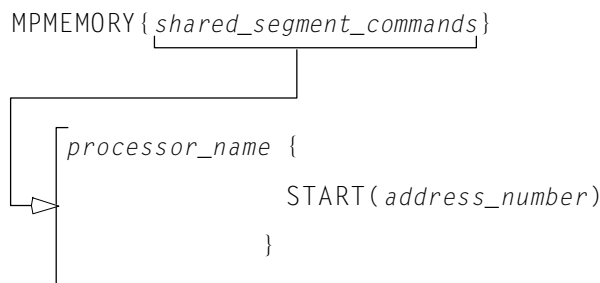


Figure 2-3. `MPMEMORY{}` Command Syntax Tree

Definitions for the parts of the `MPMEMORY{ }` command's syntax are:

- *shared_segment_commands*. Contains *processor_name* declarations with a `START{ }` address for each processor's offset in multiprocessor memory. Processor names follow the same rules as any linker label. For more information, refer to [“LDF Expressions” on page 2-12](#).
- *processor_name{placement_commands}*. Applies the *processor_name* offset for multiprocessor linking. Refer to [“PROCESSOR{ }” on page 2-42](#) for more information.

OVERLAY_GROUP{}

The OVERLAY_GROUP command provides legacy support. When running the linker, the following warning occurs.

```
[Warning 1i2534] More than one overlay group or explicit
OVERLAY_GROUP command is detected in the output section
'seg_pmda'. Create a separate output section for each group
of overlays. Expert Linker makes the change automatically
upon reading the .LDF file.
```

Memory overlays support applications whose program instructions and data do not fit in the internal memory of the processor.

Overlays may be *grouped* or *ungrouped*. Use the OVERLAY_INPUT{} command to support ungrouped overlays. Refer to [“Memory Overlay Support” on page 1-50](#) for a detailed description of overlay functionality.

The LDF OVERLAY_GROUP{} command groups overlays, so each group is brought into run-time memory, running the overlay for each group from a different starting address in run-time memory.

Overlay declarations syntactically resemble SECTIONS{} commands. They are portions of SECTIONS{} commands. The OVERLAY_GROUP{} command syntax is:

```
OVERLAY_GROUP
{
    OVERLAY_INPUT
    {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT()
        INPUT_SECTIONS()
    }
}
```

Figure 2-4 on page 2-34 demonstrates grouped overlays.

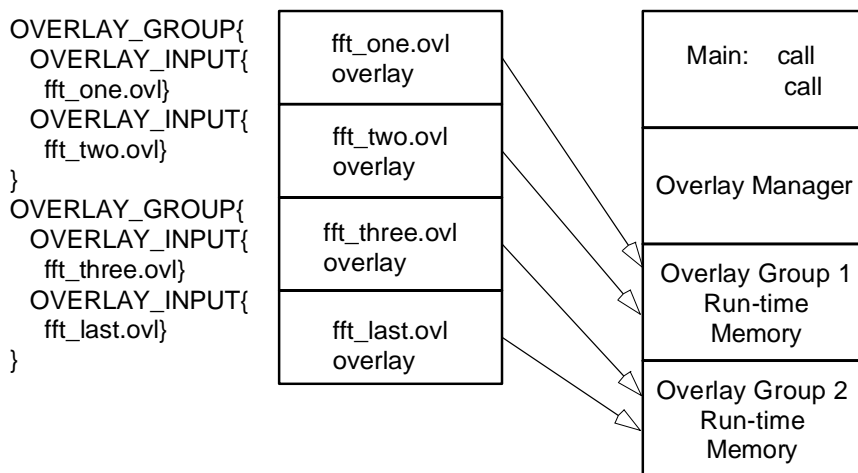


Figure 2-4. Example of Overlays – Grouped

In the simplified examples in [Listing 2-2](#) and [Listing 2-3](#), the functions are written to *overlay files* (.OVL). Whether functions are disk files or memory segments does not matter (except to the DMA transfer that brings them in). Overlays are active only while executing in run-time memory, which is located in the `program` memory segment.

Ungrouped Overlay Execution

In [Listing 2-2](#), as the FFT progresses, and overlay functions are called in turn, they are brought into run-time memory in sequence – four function transfers. [Figure 2-5](#) shows the ungrouped overlays.



“Live” locations reside in several different memory segments. The linker outputs the executable overlay (.OVL) files while allocating destinations for them in `program`.

Listing 2-2. LDF Overlays – Not Grouped

```
// This is part of the SECTIONS{} command for processor P0.
// Declare which functions reside in which overlay.
// The overlays have been split into different segments
// in one file, or into different files.
// The overlays declared in this section (seg_pmco)
// will run in segment seg_pmco.

OVERLAY_INPUT { // Overlays to live in section ovl_code
    ALGORITHM      ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_one.ovl)
    INPUT_SECTIONS ( Fft_1st.doj(program) ) } >ovl_code

OVERLAY_INPUT {
    ALGORITHM      ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_two.ovl)
    INPUT_SECTIONS ( Fft_2nd.doj(program) ) } >ovl_code

OVERLAY_INPUT {
    ALGORITHM      ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_three.ovl)
    INPUT_SECTIONS ( Fft_3rd.doj(program) ) } >ovl_code

OVERLAY_INPUT {
    ALGORITHM      ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_last.ovl)
    INPUT_SECTIONS ( Fft_last.doj(program) ) } >ovl_code
```

Grouped Overlay Execution

[Listing 2-3](#) shows a different implementation of the same algorithm. The overlaid functions are grouped in pairs. Since all four pairs of routines are resident simultaneously, the processor executes both routines before paging.

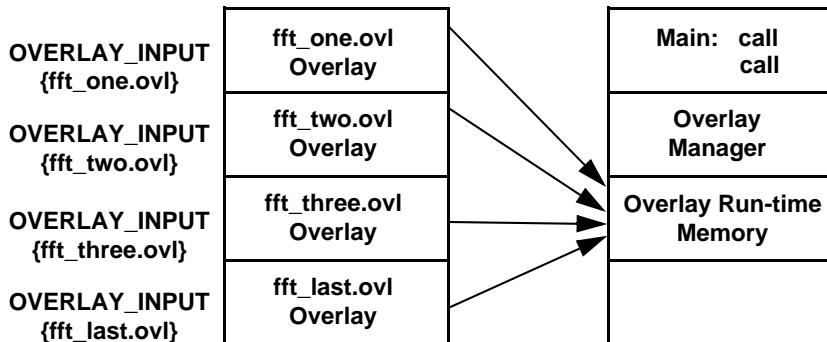


Figure 2-5. Example of Overlays – Not Grouped

Listing 2-3. LDF Overlays – Grouped

```
OVERLAY_GROUP {           // Declare first overlay group
    OVERLAY_INPUT {       // Overlays to live in section ovl_code
        ALGORITHM         ( ALL_FIT )
        OVERLAY_OUTPUT ( fft_one.ovl)
        INPUT_SECTIONS ( Fft_1st.doj(program) )
    } >ovl_code
    OVERLAY_INPUT {
        ALGORITHM         ( ALL_FIT )
        OVERLAY_OUTPUT ( fft_two.ovl)
        INPUT_SECTIONS ( Fft_mid.doj(program) )
    } >ovl_code
}
OVERLAY_GROUP {           // Declare second overlay group
```

```

OVERLAY_INPUT {    // Overlays to live in section ovl_code
    ALGORITHM      ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_three.ovl)
    INPUT_SECTIONS ( Fft_last.doj(program) )
} >ovl_code
OVERLAY_INPUT {
    ALGORITHM      ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_last.ovl)
    INPUT_SECTIONS ( Fft_last.doj(program) )
} >ovl_code
}

```

PLIT{}

The linker resolves function calls and variable accesses (both direct and indirect) across overlays. This requires the linker to generate extra code to transfer control to a user-defined routine (an overlay manager) that handles the loading of overlays. Linker-generated code goes in a special section of the executable, which has the section name `.PLIT`.

The LDF `PLIT{}` (*procedure linkage table*) command in an `.LDF` file inserts assembly instructions that handle calls to functions in overlays. The assembly instructions are specific to an overlay and execute each time a call to a function in that overlay is detected.

`PLIT{}` commands provide a template from which the linker generates assembly code when a symbol resolves to a function in overlay memory. The code typically handles a call to a function in overlay memory by calling an overlay memory manager. Refer to [“Memory Overlay Support” on page 1-50](#) for a detailed description of overlay and PLIT functionality.

A `PLIT{}` command may appear in the global LDF scope, within a `PROCESSOR{}` command, or within a `SECTIONS{}` command. For an example of using a PLIT, see [“Using PLIT{} and Overlay Manager” on page 1-64](#).

When you write the `PLIT{}` command in the LDF, the linker generates an instance of the PLIT, with appropriate values for the parameters involved, for each symbol defined in overlay code.

PLIT Syntax

Figure 2-6 shows the general syntax of the `PLIT{}` command and indicates how the linker handles a symbol (*symbol*) local to an overlay function. Table 2-5 describes parts of the command.

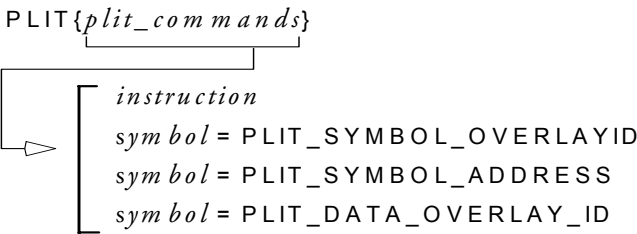


Figure 2-6. Syntax Tree of the `PLIT{}` Command

Table 2-5. Parts of the PLIT Command

Item	Description
<i>instruction</i>	None, one, or multiple assembly instructions. The instructions may occur in any reasonable order in the command structure and may precede or follow symbols. The following two constants contain information about <i>symbol</i> and the overlay in which it occurs. You must supply instructions to handle that information.
PLIT_SYMBOL_OVERLAYID	Returns the overlay ID.
PLIT_SYMBOL_ADDRESS	Returns the absolute address of the resolved symbol in run-time memory.

Command Evaluation and Setup

The linker first evaluates each *plit_command*, a sequence of assembly code. Each line is passed to a processor-specific assembler, which supplies values for the symbols and expressions.

After evaluation, the linker places the returned bytes into the `.PLIT` output section and manages addressing in that output section.

To help you write an overlay manager, the linker generates PLIT constants for each symbol in an overlay. Data can be overlaid, just like code.

If an overlay-resident function calls for additional data overlays, include an instruction for finding them.

After the setup and variable identification are completed, the overlay itself is brought (via DMA transfer) into run-time memory. This takes place under the control of assembly code called an overlay manager.



The branch instruction, such as `jump OverlayManager`, is normally the last instruction in the `PLIT{}` command.

Allocating Space for PLITs

The `.LDF` file must allocate space in memory to hold PLITs built by the linker. Typically, that memory resides in the program code memory segment. A typical LDF declaration for that purpose is:

```
// ... [In the SECTIONS command for Processor P0]
// Plit code is to reside and run in mem_program segment
.plit {} > mem_program
```

A `PLIT{}` command may appear in the global LDF scope, within a `PROCESSOR{}` command, or within a `SECTIONS{}` command.

- There is no input section associated with the `.PLIT` output section. The LDF allocates space for linker-generated routines, which do not contain (input) data objects.
- This segment allocation does not take any parameters. You write the structure of this command per `PLIT` syntax. The linker creates an instance of the command for each symbol that resolves to an overlay. The linker stores each instance in the `.PLIT` output section, which becomes part of the program code's memory segment.

PLIT Examples

The following are two examples of `PLIT{}` command implementation.

Simple PLIT – States are Not Saved

A simple `PLIT` merely copies the symbol's address and overlay ID into registers, and jumps to the overlay manager. The following fragment was extracted from the global scope (just after the `MEMORY{}` command) of `sample_fft_group.ldf`. Verify that the contents of `AX0` and `AX1` are either safe or irrelevant.

```
/* The global PLIT to be used whenever a PROCESSOR or OVERLAY
specific PLIT description is not provided. The plit initializes a
register to the overlay ID and the overlay run-time address of
the symbol called. Ensure the registers used in the plit do not
contain values that cannot be overwritten. */
```

```
PLIT
{
    P0 = PLIT_SYMBOL_OVERLAYID;
    P1.L = PLIT_SYMBOL_ADDRESS;
    P1.H = PLIT_SYMBOL_ADDRESS;
```

```
JUMP _OverlayManager;  
}
```

As a general case, minimize overlay transfer traffic. Improve performance by designing code so overlay functions are imported and use minimal (or no) reloading.

PLIT – Summary

A PLIT is a template of instructions for loading an overlay. For each overlay routine in the program, the linker builds and stores a list of PLIT instances according to that template, as it builds its executable. The linker may also save registers or stack context information. The linker does not accept a PLIT without arguments. If you do not want the linker to redirect function calls in overlays, omit the `PLIT{}` commands entirely.

To help you write an overlay manager, the linker generates `PLIT_SYMBOL` constants for each symbol in an overlay.

The overlay manager can also:

- Be helped by manual intervention. Save the target’s state on the stack or in memory before loading and executing an overlay function, so it continues correctly on return. However, you can implement this feature within the PLIT section of your LDF.

Note: Your program may not need to save this information.

- Initiate (jump to) the routine that transfers the overlay code to internal memory, given the previous information about its identity, size and location: `_OverlayManager`. “Smart” overlay managers first check whether an overlay function is already in internal memory to avoid reloading the function.

PROCESSOR{}

The LDF `PROCESSOR{}` command declares a processor and its related link information. A `PROCESSOR{}` command contains the `MEMORY{}`, `SECTIONS{}`, `RESOLVE{}`, and other linker commands that apply only to that processor.

The linker produces one executable file from each `PROCESSOR{}` command. If you do not specify the type of link with a `PROCESSOR{}` command, the linker cannot link your program.

The syntax for the `PROCESSOR{}` command appears in [Figure 2-7](#).

```
PROCESSOR processor_name
{
    OUTPUT(file_name.DXE)
    [MEMORY{segment_commands}]
    [PLIT{plit_commands}]
    SECTIONS{section_commands}
    RESOLVE(symbol, resolver)
}
```

Figure 2-7. `PROCESSOR{}` Command Syntax

The `PROCESSOR{}` command syntax is defined as.

- `processor_name`

Assigns a name to the processor. Processor names follow the same rules as linker labels. [For more information, see “LDF Expressions” on page 2-12.](#)

- `OUTPUT(file_name.DXE)`

Specifies the output file name for the executable (.DXE). An `OUTPUT()` command in a scope must appear before the `SECTIONS{}` command in that scope.

- `MEMORY{segment_commands}`

Defines memory segments that apply only to this processor. Use command scoping to define these memory segments outside the `PROCESSOR{}` command. [For more information, see “Command Scoping” on page 2-11 and “MEMORY{” on page 2-28.](#)

- `PLIT{plit_commands}`

Defines procedure linkage table (PLIT) commands that apply only to this processor. [For more information, see “PLIT{” on page 2-37.](#)

- `SECTIONS{section_commands}`

Defines sections for placement within the executable (.DXE). [For more information, see “SECTIONS{” on page 2-45.](#)

- `RESOLVE(symbol, resolver)`

Ignores any `LINK_AGAINST()` command. For details, see [“RESOLVE{” on page 2-44.](#)

RESOLVE()

Use the LDF `RESOLVE(symbol_name, resolver)` command to ignore a `LINK_AGAINST()` command for a specific symbol. This overrides the search order for a specific variable or label. Refer to the “[LINK_AGAINST\(\)](#)” on [page 2-27](#) for more information.

The `RESOLVE(symbol_name, resolver)` command uses the resolver to resolve a particular symbol (variable or label) to an address. The resolver is an absolute address or a file (.DxE or .SM) that contains the definition of the symbol. If the symbol is not located in the designated file, an error is issued.



Resolve a C/C++ variable by prefixing the variable with an underscore in the `RESOLVE()` command (for example, `_symbol_name`).

SEARCH_DIR()

The LDF `SEARCH_DIR()` command specifies one or more directories that the linker searches for input files. Specify multiple directories within a `SEARCH_DIR` command by delimiting each path with a semicolon (;) and enclosing long directory names within straight quotes.

The search order follows the order of the listed directories. This command appends search directories to the directory selected with the linker's `-L` command-line switch. Place this command at the beginning of the LDF, so the linker applies the command to all file searches.

Example

```
ARCHITECTURE (ADSP-BF535)
MAP (SINGLE-PROCESSOR.MAP)           // Generate a MAP file

SEARCH_DIR( $ADI_DSP\Blackfin\lib; ABC\XYZ )
// $ADI_DSP is a predefined linker macro that expands
// to the VDSP install directory. Search for objects
```

```
// in directory Blackfin/lib relative to the install directory
// and to the ABC\XYZ directory.
```

SECTIONS{}

The LDF `SECTIONS{}` command uses memory segments (defined by `MEMORY{}` commands) to specify the placement of output sections into memory. [Figure 2-8](#) shows syntax for the `SECTIONS{}` command.



Figure 2-8. Syntax Tree of the `SECTIONS{}` Command

An `.LDF` file may contain one `SECTIONS{}` command within each `PROCESSOR{}` command. The `SECTIONS{}` command must be preceded by a `MEMORY{}` command, which defines the memory segments in which the

linker places the output sections. Though an LDF may contain only one `SECTIONS{}` command within each command scope, multiple output sections may be declared within each `SECTIONS{}` command.

The `SECTIONS{}` command's syntax includes several arguments.

expressions

or

section_declarations

Use *expressions* to manipulate symbols or to position the current location counter. Refer to [“LDF Expressions” on page 2-12](#).

Use a *section_declaration* to declare an output section. Each *section_declaration* has a *section_name*, and optional *section_type*, *section_commands* and a *memory_segment*.

Figure 2-9. Parts of a Section Declaration

Item	Description
<i>section_name</i>	Must start with a letter, underscore, or period and may include any letters, underscores, digits, and points. A <i>section_name</i> must not conflict with any LDF keywords. The special <i>section_name</i> , <code>.PLIT</code> , indicates the procedure linkage table (PLIT) section that the linker generates when resolving symbols in overlay memory. Place this section in non-overlay memory to manage references to items in overlay memory.
<i>section_type</i>	(optional) Assigns an ELF section type. The only valid section type keyword is <code>SHT_NOBITS</code> (section header type no bits). This section type contains uninitialized data, so even if it is large, it can download quickly. Space is allocated but not written. For an example of <code>SHT_NOBITS</code> , see Listing 2-6 on page 2-56 .

Figure 2-9. Parts of a Section Declaration

Item	Description
<i>section_commands</i>	May consist of any combination of <code>INPUT_SECTIONS()</code> , <code>FILL()</code> , <code>PLIT()</code> , or <code>OVERLAY_INPUT()</code> commands and/or expressions.
<i>memory_segment</i>	Declares that the output section is placed in the specified memory segment. The <i>memory_segment</i> is optional. Some sections, such as those for debugging, need not be included in the memory image of the executable, but are needed for other development tools that read the executable file. By omitting a memory segment assignment for a section, you direct the linker to generate the section in the executable, but prevent section content from appearing in the memory image of the executable.

INPUT_SECTIONS()

The LDF `INPUT_SECTIONS()` portion of a *section_command* identifies the parts of the program to place in the executable. When placing an input section, you must specify the *file_source*. When *file_source* is a library, specify the input section's *archive_member* and *input_labels*.


- *file_source* may be a list of files or an LDF macro that expands into a file list, such as `$COMMAND_LINE_OBJECTS`. Delimit the list of object files or library files with commas.
- *archive_member* names the source-object file within a library. The *archive_member* parameter and the left/right brackets, (`[]`), are required when the *file_source* of the *input_label* is a library.
- *input_labels* are derived from run-time `.SECTION` names in assembly programs. Delimit the list of names with commas.

expression

In a *section_command*, an *expression* manipulates symbols or positions the current location counter. See [“LDF Expressions” on page 2-12](#) for details.

FILL(hex number)

In a *section_command*, the LDF `FILL()` command fills gaps (created by aligning or advancing the current location counter) with hexadecimal numbers.

 `FILL()` can be used only within a section declaration.

By default, the linker fills gaps with zeros. Specify only one `FILL()` command per output section. For example,

```
FILL (0x0)
or
FILL (0xFFFF)
```

PLIT{*plit_commands*}

In a *section_command*, a `PLIT{}` command declares a locally scoped procedure linkage table (PLIT). It contains its own labels and expressions. [For more information, see “PLIT{ }” on page 2-37.](#)

OVERLAY_INPUT{*overlay_commands*}

In a *section_command*, `OVERLAY_INPUT{}` identifies the parts of the program to place in an overlay executable (`.OVL` file). For more information on overlays, see [“Linking for Overlay Memory Example” on page 2-68.](#)

overlay_commands consist of at least one of the following commands:

`INPUT_SECTIONS()`, `OVERLAY_ID()`, `NUMBER_OF_OVERLAYS()`,
`OVERLAY_OUTPUT()`, `ALGORITHM()`, `RESOLVE_LOCALLY()`, or `SIZE()`.

overlay_memory_segment (optional) determines whether the overlay section is placed in an overlay memory segment. Some overlay sections, such as those loaded from a host, need not be included in the overlay memory image of the executable, but are required for other tools that read the executable file.

Omitting an overlay memory segment assignment from a section retains the section in the executable, but marks the section for exclusion from the overlay memory image of the executable.

The *overlay_commands* portion of an `OVERLAY_INPUT{ }` command follows these rules.

- `OVERLAY_OUTPUT()` outputs an overlay file (`.OVL`) for the overlay with the specified name. The `OVERLAY_OUTPUT()` in an `OVERLAY_INPUT{ }` command must appear before any `INPUT_SECTIONS()` for that overlay.
- `INPUT_SECTIONS()` has the same syntax within an `OVERLAY_INPUT{ }` command as when it appears within an *output_section_command*, except that a `.PLIT` section may not be placed in overlay memory. [For more information, see “INPUT_SECTIONS\(\)” on page 2-47.](#)
- `OVERLAY_ID()` returns the overlay ID.
- (not currently available) `NUMBER_OF_OVERLAYS()` returns the number of overlays that the current link generates when using the `FIRST_FIT` or `BEST_FIT` overlay placement `ALGORITHM()`.
- `ALGORITHM()` directs the linker to use the specified overlay linking algorithm. The only currently available linking algorithm is `ALL_FIT`.

For `ALL_FIT`, the linker tries to fit all the `OVERLAY_INPUT{ }` into a single overlay that can overlay into the output section’s run-time memory segment.

(not currently available) For `FIRST_FIT`, the linker splits the input sections listed in `OVERLAY_INPUT{ }` into a set of overlays that can each overlay the output section’s run-time memory segment, according to First-In-First-Out (FIFO) order.

(not currently available) For `BEST_FIT`, the linker splits the input sections listed in `OVERLAY_INPUT{ }` into a set of overlays that can each overlay the output section’s run-time memory segment, but splits these overlays to optimize memory usage.

- `RESOLVE_LOCALLY()`, when applied to an overlay, controls whether the linker generates PLIT entries for function calls that are resolved within the overlay.

`RESOLVE_LOCALLY(TRUE)`, the default, does not generate PLIT entries for locally resolved functions within the overlay.

`RESOLVE_LOCALLY(FALSE)` generates PLIT entries for all functions, regardless of whether they are locally resolved within the overlay.

- `SIZE()` sets an upper limit to the size of the memory that may be occupied by an overlay.

SHARED_MEMORY{ }

The linker can produce two types of executable output — .DXE files and .SM files. A .DXE file runs in a single-processor system's address space. Shared memory executable (.SM) files reside in the shared memory of a multiprocessor system. `SHARED_MEMORY{ }` produces .SM files.

If you do not specify the type of link with a `PROCESSOR{ }` or `SHARED_MEMORY{ }` command, the linker cannot link your program.

Your LDF may contain multiple `SHARED_MEMORY{ }` commands, but the maximum number of processors that can access a shared memory is processor-specific.

The LDF `SHARED_MEMORY{ }` command must appear within the scope of a `MEMORY{ }` command. `PROCESSOR{ }` commands declaring the processors that share this memory must also appear within this scope.

The syntax for the `SHARED_MEMORY{ }` command appears in [Figure 2-10](#) followed by definitions of its components in [Table 2-6](#).

An example of this command scoping appears in [Table 2-11](#).


```

    SHARED_MEMORY
    {
        OUTPUT(file_name.SM)
        SECTIONS{section_commands}
    }

```

Figure 2-10. SHARED_MEMORY{} Command Syntax

Table 2-6. Parts of the SHARED_MEMORY{} Command

Item	Description
OUTPUT()	Specifies the output file name (<i>file_name</i> .SM) of the shared memory executable (.SM) file. An OUTPUT() command in a SHARED_MEMORY{} command must appear before the SECTIONS{} command in that scope.
SECTIONS()	Defines sections for placement within the shared memory executable (.SM)

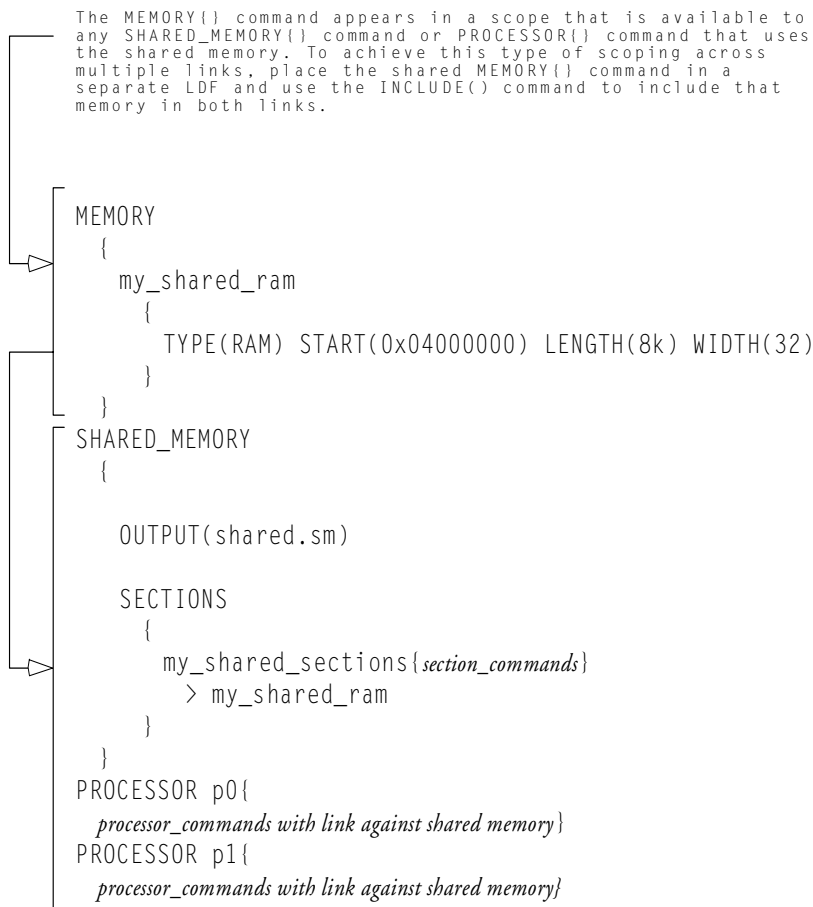


Figure 2-11. LDF Scopes for SHARED_MEMORY{}

LDF Programming Examples

This section shows several example .LDF files. As you modify these examples, refer to the syntax descriptions in “[LDF Commands](#)” on page 2-22.

This section provides the following examples:

- “[Linking for a Single-Processor System](#)” on page 2-54
- “[Linking Large Uninitialized Variables](#)” on page 2-55
- “[Linking for Assembly Source File](#)” on page 2-57
- “[Linking for C Source File – Example 1](#)” on page 2-60
- “[Linking for Complex C Source File – Example 2](#)” on page 2-63
- “[Linking for Overlay Memory Example](#)” on page 2-68



The source code for several programs is bundled with the development software. Each program includes an .LDF file. For working examples of the linking process, examine the .LDF files that come with the examples. These examples are in the directory:

```
VisualDSP++ InstallPath>\Blackfin\examples
```



The development software includes a variety of preprocessor default .LDF files. These files provide an example .LDF for each processor’s internal memory architecture. The default .LDF files are in the directory:

```
VisualDSP++ InstallPath>\Blackfin\ldf
```

Linking for a Single-Processor System

When linking an executable for a single-processor system, the .LDF file describes the processor's memory and places code for that processor. The .LDF file in [Listing 2-4](#) is for a single-processor system. Note the following commands in this example .LDF file:

- `ARCHITECTURE()` defines the processor type
- `SEARCH_DIR()` commands add the `lib` and current working directory to the search path
- `$OBJ`s and `$LIBS` macros get object (`.DOJ`) and library (`.DLB`) file input
- `MAP()` outputs a map file
- `MEMORY{}` defines memory for the processor
- `PROCESSOR{}` and `SECTIONS{}` commands define a processor and place program sections for that processor's output file, using the memory definitions

Listing 2-4. Example .LDF File for a Single-Processor System

```
ARCHITECTURE(ADSP-BF535)

SEARCH_DIR( $ADI_DSP\Blackfin\lib )

MAP(SINGLE-PROCESSOR.MAP) // Generate a MAP file

// $ADI_DSP is a predefined linker macro that expands
// to the VDSP install directory. Search for objects in
// directory Blackfin/lib relative to the install directory

LIBS libc.dlb, libevent.dlb, libsftflt.dlb, libcpp_blkfn.dlb,
libcppprt_blkfn.dlb, libdsp.dlb
$LIBRARIES = LIBS, librt.dlb;
```

```
// single.doj is a user generated file. The linker will be
// invoked as follows
//    linker -T single-processor.ldf single.doj.
// $COMMAND_LINE_OBJECTS is a predefined linker macro
// The linker expands this macro into the name(s) of the
// the object(s) (.doj files) and archives (.dlb files)
// that appear on the command line. In this example,
// $COMMAND_LINE_OBJECTS = single.doj

$OBJECTS = $COMMAND_LINE_OBJECTS;

//    A linker project to generate a DXE file

PROCESSOR P0
{
    OUTPUT( SINGLE.DXE ) // The name of the output file

    MEMORY                // Processor specific memory command
    { INCLUDE( "BF535_memory.ldf" ) }


    SECTIONS               // Specify the Output Sections
    { INCLUDE( "BF535_sections.ldf" ) }
    // end P0 sections
}                          // end P0 processor
```

Linking Large Uninitialized Variables

When linking an executable file that contains large uninitialized variables, you can reduce the size of the file by using the `SHT_NOBITS` section qualifier (Section Header Type No Bits).

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers this can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur large download times when using with the emulator. This also may occur when booting from a loader file (because of the increased file size). [Listing 2-6](#) shows an example using the `SHT_NOBITS` section to avoid initialization of a segment.

The .LDF file can specify that an output section be omitted from the output file. The SHT_NOBITS output section qualifier directs the linker to omit data for that section from the output file.

 The SHT_NOBITS technique corresponds to using the /UNINIT segment qualifier in previous (.ACH) development tools. Even if you do not use the SHT_NOBITS technique, the boot loader removes variables initialized to zeros from the .LDR file and replaces them with instructions for the loader kernel to zero out the variable. This reduces the loader's output file size, but still requires execution time for the processor to initialize the memory with zeros.

Listing 2-5. Large Uninitialized Variables: Assembly Source

```
.SECTION extram_area;    /* 1Mx8 EXTRAM */  
.BYTE    huge_buffer[0x0060000];
```

Listing 2-6. Large Uninitialized Variables: .LDF File Source

```
ARCHITECTURE(ADSP-BF535)  
$OBJECTS = $COMMAND_LINE_OBJECTS; // Libraries & objects from  
                                     // the command line  
MEMORY {  
    mem_extram {  
        TYPE(RAM) START(0x10000) END(0x15fff) WIDTH(8)  
    } // end segment  
} // end memory  
  
PROCESSOR P0 {  
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )  
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )  
    // SHT_NOBITS section isn't written to the output file  
    SECTION {  
        extram_output SHT_NOBITS {  
            INPUT_SECTIONS( $OBJECTS ( extram_area ) ) } >mem_extram;  
        } //end section  
    } // end processor P0
```

Linking for Assembly Source File

[Listing 2-8](#) shows an example .LDF file for an ADSP-BF535 DSP that describes a simple memory placement of an assembly source file ([Listing 2-7](#)) which contains code and data that is to reside in, and execute from, L2 SRAM. This example assumes that the code and data declared in L2 memory is cacheable within L1 code and data memories. The LDF file includes two commands, MEMORY and SECTIONS, which are used to describe specific memory and system information. Refer to Notes for [Listing 2-1 on page 2-4](#) for information on items in this basic example.

Listing 2-7. MyFile.ASM

```
.SECTION program;
.GLOBAL main;
main:

    p0.l = myArray;
    p0.h = myArray;
    r0 = [p0++];
    ...

.SECTION data1;
.GLOBAL myArray;
.VAR myArray[256] = "myArray.dat";
```

Listing 2-8. Simple .LDF File Based on Assembly Source File Only

```
#define L2_START 0xf0000000
#define L2_END   0xf003ffff

// Declare specific DSP Architecture here (for linker)
ARCHITECTURE(ADSP-BF535)
// LDF macro equals all object files in project command line
$OBJECTS = $COMMAND_LINE_OBJECTS;

// Describe the physical system memory below
```

LDF Programming Examples

```
MEMORY{
    // 256KB L2 SRAM memory segment for user code
    // and data L2SRAM
    {TYPE(RAM) START(L2_START) END(L2_END) WIDTH(8)}
}

PROCESSOR p0{
    OUTPUT($COMMAND_LINE_OUTPUT_FILE)

    SECTIONS{
        DXE_L2SRAM{
            // Align L2 instruction segments on a 2-byte boundaries
            INPUT_SECTION_ALIGN(2)
            INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
            // Align L2 data segments on 1-byte boundary
            INPUT_SECTION_ALIGN(INPUT_SECTIONS
                                ($OBJECTS(data1) $LIBRARIES(data1)))
        }
    }
}
// end SECTIONS
// end PROCESSOR p0
```

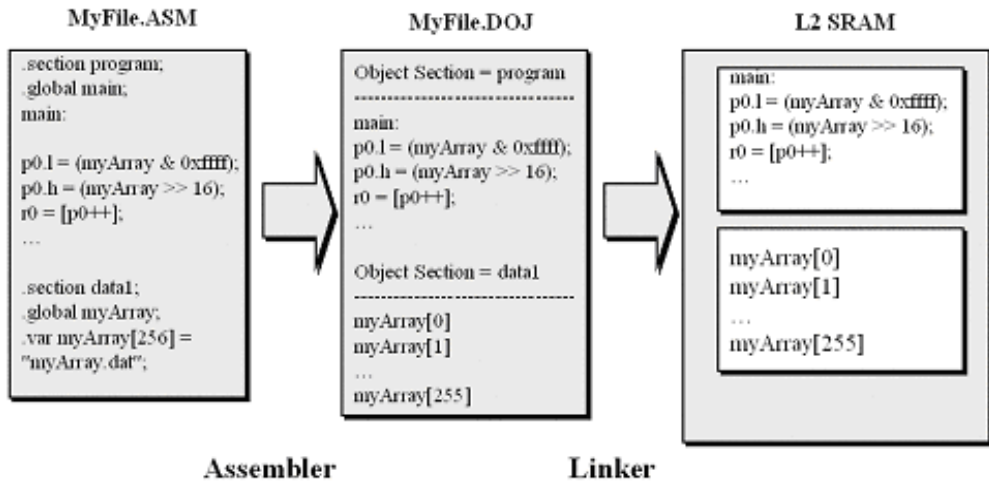



Figure 2-12. Assembly-to-Memory Code Placement

Linking for C Source File – Example 1

[Listing 2-10](#) shows an example LDF that describes the memory placement of a simple C source file ([Listing 2-9](#)) which contains code and data that is to reside in, and execute from, L2 SRAM. This example also assumes that the code and data declared in L2 memory is cacheable within L1 code and data memories. The LDF file includes two commands, `MEMORY` and `SECTIONS`, which are used to describe specific memory and system information. Refer to Notes for [Listing 2-1 on page 2-4](#) for information on items in this basic example.

Listing 2-9. Simple C Source File Example 1

```
int myArray[256];

void main(void){
    int i;

    for(i=0; i<256; i++)
        myArray[i] = i;

} // end main ()
```

Listing 2-10. Example: Simple C-based .LDF File for ADSP-BF535 Processor

```
ARCHITECTURE(ADSP-BF535)
SEARCH_DIR( $ADI_DSP\Blackfin\lib )

#define LIBS libsmall535.dlb libc535.dlb libm3free535.dlb
libevent535.dlb libio535.dlb libcpp535.dlb libcpprt535.dlb
libdsp535.dlb libsftflt535.dlb libetsi535.dlb idle535.doj

$LIBRARIES = LIBS, librt535.dlb;
```

```
$OBJECTS = crts535.doj, $COMMAND_LINE_OBJECTS crtn535.doj;
MEMORY{
    // 248KB of L2 SRAM memory segment for user code and data
    MemL2SRAM {TYPE(RAM) START(0xf0000000) END(0xF003dfff)
               WIDTH(8)}
    // 4KB of L2 SRAM memory for C run-time stack (user mode)
    MemStack {TYPE(RAM) START(0xf003e000) END(0xf003efff) WIDTH(8)}
    // 4KB of L2 SRAM for stack memory segment (supervisor mode)
    MemSysStack {TYPE(RAM) START(0xf003f000) END(0xf003ffff)
                WIDTH(8)}
    // 4KB of Scratch SRAM for Heap memory segment
    MemHeap {TYPE(RAM) START(0xFFB00000) END(0xFFB00FFF) WIDTH(8)}
}

PROCESSOR p0{

    OUTPUT($COMMAND_LINE_OUTPUT_FILE)

    SECTIONS{}
    // Declare L2 Input objects below...
    DXE_L2_SRAM{
        // Align L2 instruction segments on a 2-byte boundaries
        INPUT_SECTION_ALIGN(2)
        INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
        // Align L2 data segments on a 1-byte boundary
        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
        INPUT_SECTIONS( $OBJECTS(cp1b) $LIBRARIES(cp1b))
        INPUT_SECTIONS( $OBJECTS(cp1b_code) $LIBRARIES(cp1b_code))
        INPUT_SECTIONS( $OBJECTS(cp1b_data) $LIBRARIES(cp1b_data))
        // Align L2 constructor data segments on a 1-byte boundary
        // (C++ only)
        INPUT_SECTION_ALIGN(1)
        INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
    }
```

LDF Programming Examples

```
}>MemL2SRAM

// Allocate memory segment for C run-time stack segment stack
// Assign start address of stack to 'ldf_stack_space'
// variable using the LDF's current location counter "."
ldf_stack_space = .;
// Assign end address of stack to 'ldf_stack_end' variable
ldf_stack_end = ldf_stack_space + MEMORY_SIZEOF(MemStack) - 4;
}>MemStack

// Allocate memory segment for system stack sysstack
// Assign start address of sys stack to 'ldf_sysstack_space'
// variable using the LDF's current location counter "."
ldf_sysstack_space = .
// Assign end address of stack to 'ldf_sysstack_end' variable
ldf_sysstack_end = ldf_sysstack_space + MEMORY_SIZEOF
                  (MemSysStack) - 4;
}>MemSysStack

// Allocate a heap segment (for dynamic memory allocation)
// heap
// Assign start address of heap to 'ldf_heap_space' variable
// using the LDF's current location counter "."
ldf_heap_space = .;
// Assign end address of heap to 'ldf_heap_length' variable
ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MemHeap) - 1;
// Assign length of heap to 'ldf_heap_length' variable
ldf_heap_length = ldf_heap_end - ldf_heap_space;
}>MemHeap

// end SECTIONS{}
// end PROCESSOR p0{}
```

Linking for Complex C Source File – Example 2

Listing 1-3 shows an example LDF that describes the memory placement of a C source file, which contains code and data that is to reside in, and execute from, L1, L2, and Scratchpad SRAM, as well as external SDRAM Banks 0 through 3. The LDF file includes two commands, `MEMORY` and `SECTIONS`, which are used to describe specific memory and system information. Refer to Notes for [Listing 2-1 on page 2-4](#) for information on items in this complex example.

Listing 2-11. Complex C Source File Example

```
static section ("Fast_Code") void MEM_DMA_ISR(void){

...
}

static section ("SDRAM_0") int page_buff1[0x08000000];
static section ("SDRAM_1") int page_buff2[0x08000000];
static section ("SDRAM_2") int page_buff3[0x08000000];
static section ("SDRAM_3") int page_buff4[0x08000000];

static section ("Data_BankA") int coeffs1[256];
static section ("Data_BankB") int input_array[0x2000];

int x, y, z;
void main(void){

int i;
x = 0x5555;

...
}
```

LDF Programming Examples

The following is an example of an LDF file (for ADSP-BF535 DSP) which is based on the complex C source from Listing 1-13.

Listing 2-12. C .LDF File Example - SDRAM.LDF

```
ARCHITECTURE(ADSP-BF535)
SEARCH_DIR($ADI_DSP\Blackfin\lib

#define LIBS libsmall535.dlb libc535.dlb libm3free535.dlb
libevent535.dlb libio535.dlb libcpp535.dlb libcppprt535.dlb
libdsp535.dlb libsftflt535.dlb libetsi535.dlb idle535.doj

$LIBRARIES = LIBS, librt535.dlb;

$OBJECTS = crts535.doj, $COMMAND_LINE_OBJECTS crtn535.doj;

// Define physical system memory below...
MEMORY{
    // 16KB of user code in L1 SRAM segment
    Mem_L1_Code_SRAM {TYPE(RAM) START(0xFFA00000) END(0xFFA03FFF)
                      WIDTH(8)}
    // 16KB of user data in L1 Data Bank A SRAM
    Mem_L1_DataA_SRAM {TYPE(RAM) START(0xFF800000) END(0xFF803FFF)
                      WIDTH(8)}
    // 16KB of user data in L1 Data Bank B SRAM
    Mem_L1_DataB_SRAM {TYPE(RAM) START(0xFF900000) END(0xFF903FFF)
                      WIDTH(8)}

    // 4KB of L1 Scratch memory for C run-time stack (user mode)
    Mem_Scratch_Stack {TYPE(RAM) START(0xFFB00000) END(0xFFB007FF)
                      WIDTH(8)}

    // 248KB of user code and data in L2 SRAM segment
    Mem_L2_SRAM {TYPE(RAM) START(0xF0000000) END(0xF003DFFF)
```

```
        WIDTH(8)}
// 4KB for heap in L2 SRAM (for dynamic memory allocation)
Mem_Heap {{TYPE(RAM) START(0xF003E000) END(0xF003EFFF)
        WIDTH(8)}}

// 4KB for system stack in L2 SRAM (supervisor mode stack)
Mem_SysStack {TYPE(RAM) START(0xF003F000) END(0xF003FFFF)
        WIDTH(8)}

// 4 x 128MB External SDRAM memory segments
Mem_SDRAM_Bank0 {TYPE(RAM) START(0x00000000) END(0x07FFFFFF)
        WIDTH(8)}
Mem_SDRAM_Bank1 {TYPE(RAM) START(0x08000000) END(0x0FFFFFFF)
        WIDTH(8)}
Mem_SDRAM_Bank2 {TYPE(RAM) START(0x10000000) END(0x17FFFFFF)
        WIDTH(8)}
Mem_SDRAM_Bank3 {TYPE(RAM) START(0x18000000) END(0x1FFFFFFF)
        WIDTH(8)}
} // end MEMORY{}

PROCESSOR p0{
    OUTPUT($COMMAND_LINE_OUTPUT_FILE)

    SECTIONS{
        // Input section declarations for L1 code memory
        DXE_L1_Code_SRAM{
            // Align L1 code segments on a 2-byte boundary
            INPUT_SECTION_ALIGN(2)
            INPUT_SECTIONS($OBJECTS(Fast_Code)
        }>Mem_L1_Code_SRAM

        // Input section declarations for L1 data bank A memory
        DXE_L1_DataA_SRAM{
            // Align L1 data segments on a 1-byte boundary
```

LDF Programming Examples

```
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(Data_BankA)
}>Mem_L1_BankA_SRAM

// Input section declarations for L1 data bank B memory
DXE_L1_BankB_SRAM{
// Align L1 data segments on a 1-byte boundary
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(Data_BankB)
}>Mem_L1_BankB_SRAM

stack{
    ldf_stack_space = .;
    ldf_stack_end = ldf_stack_space +
                    MEMORY_SIZEOF(Mem_Scratch_Stack) - 4;
}>Mem_Scratch_Stack

sysstack{
    ldf_sysstack_space = .;
    ldf_sysstack_end = ldf_sysstack_space +
                     MEMORY_SIZEOF(Mem_SysStack) - 4;
}>Mem_SysStack

heap{
    ldf_heap_space = .;
    ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(Mem_Heap) - 1;
    ldf_heap_length = ldf_heap_end - ldf_heap_space;
}>Mem_Heap

DXE_L2_SRAM{
// Align L2 code segments on a 2-byte boundary
INPUT_SECTION_ALIGN(2)
INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
// Align L2 data segments on a 1-byte boundary
```



```
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
// Align L2 constructor data segments on a 1-byte boundary
// (C++ only)
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
}>Mem_L2_SRAM

DXE_SDRAM_0{
// Align external SDRAM data segments on a 1-byte boundary
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(SDRAM_0))
}>Mem_SDRAM_Bank0

DXE_SDRAM_1{
// Align external SDRAM data segments on a 1-byte boundary
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(SDRAM_1))
}>Mem_SDRAM_Bank1

DXE_SDRAM_2{
// Align external SDRAM data segments on a 1-byte boundary
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(SDRAM_2))
}>Mem_SDRAM_Bank2

DXE_SDRAM_3{
// Align external SDRAM data segments on a 1-byte boundary
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS($OBJECTS(SDRAM_3))
}>Mem_SDRAM_Bank3

} // End Sections{}
} // End PROCESSOR p0{}
```

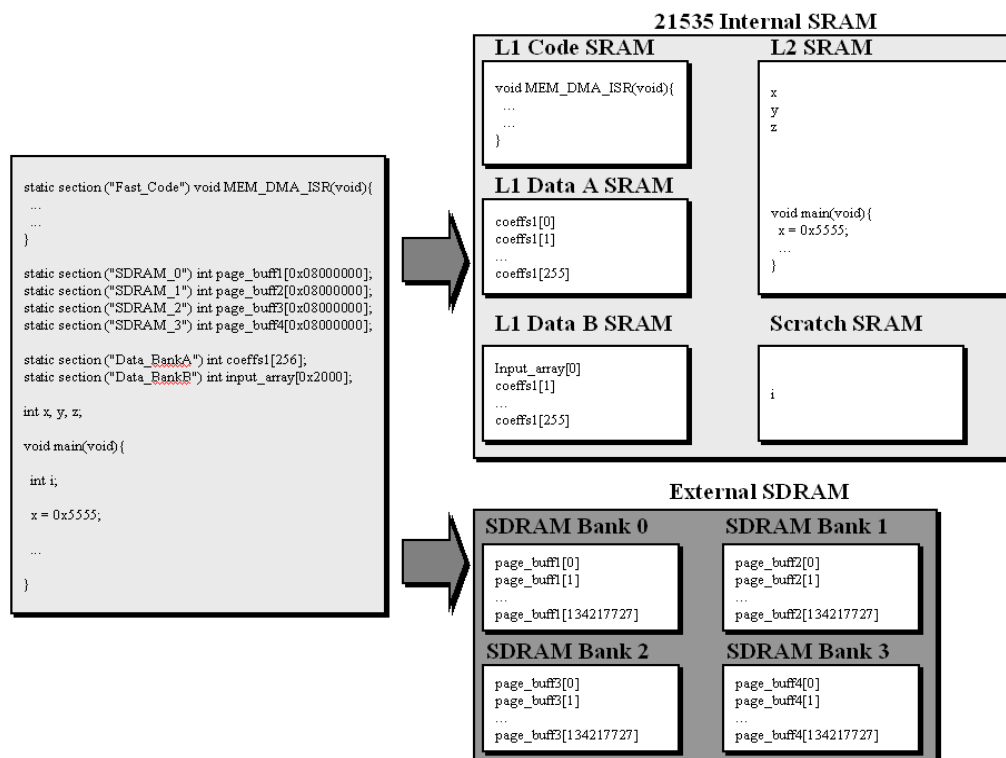


Figure 2-13. C-to-Memory Code Placement

Linking for Overlay Memory Example

When linking executable files for an overlay memory system, the .LDF file describes the overlay memory, the processor(s) that use the overlay memory, and each processor's unique memory. The .LDF file places code for each processor and the special PLIT{} section.

Listing 2-13 shows an example .LDF file for an overlay-memory system. For more information on this .LDF file, see the comments in the listing.

Listing 2-13. Example .LDF File for an Overlay-Memory System

```

ARCHITECTURE(BF535)
SEARCH_DIR( $ADI_DSP\Blackfin\lib )

{
MAP(overlay.map)
// This simple example uses internal memory for overlays
// (Real overlays would never “live” in internal memory)

MEMORY
{
    MEM_PROGRAM { TYPE(RAM) START(0xF0000000) END(0xF002FFFF)
                  WIDTH(8) }
    MEM_HEAP    { TYPE(RAM) START(0xF0030000) END(0xF0037FFF)
                  WIDTH(8) }
    MEM_STACK   { TYPE(RAM) START(0xF0038000) END(0xF003DFFF)
                  WIDTH(8) }
    MEM_SYSSTACK { TYPE(RAM) START(0xF003E000) END(0xF003EFFF)
                  WIDTH(8) }
    MEM_OVLY    { TYPE(RAM) START(0x00000000) END(0x08000000)
                  WIDTH(8) }
}

PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS
    {
        dxreset { INPUT_SECTIONS($OBJECTS(IVreset))
        } >MEM_PROGRAM
        dxetab { INPUT_SECTIONS($OBJECTS(IVpwrdown))
    }
}

```

LDF Programming Examples

```
// Processor and application specific assembly language
// instructions, generated for each symbol that is resolved
// in overlay memory.

PLIT
{
    P0 = PLIT_SYMBOL_OVERLAYID;
    P1.L = PLIT_SYMBOL_ADDRESS;
    P1.H = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}

#define LIBS libsmall535.dlb libc535.dlb libm3free535.dlb
libevent535.dlb libio535.dlb libcpp535.dlb libcpprt535.dlb
libdsp535.dlb libsftflt535.dlb libetsi535.dlb idle535.doj

$LIBRARIES = LIBS, librt535.dlb;

$OBJECTS = crts535.doj, $COMMAND_LINE_OBJECTS crtn535.doj;

PROCESSOR P0 {
    $P0_OBJECTS = main.doj , manager.doj;
    OUTPUT(mgrovly.dxe)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
        program
        {
            // Align all code sections on 2 byte boundary
            INPUT_SECTION_ALIGN(2)
            INPUT_SECTIONS
                ( $OBJECTS(program) $LIBRARIES(program) )
            INPUT_SECTION_ALIGN(1)
        }
    }
}
```

```

INPUT_SECTIONS
    ( $OBJECTS(data1) $LIBRARIES(data1) )
INPUT_SECTIONS( $OBJECTS(cplb) $LIBRARIES(cplb))
INPUT_SECTIONS
    ( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
INPUT_SECTIONS
    ( $OBJECTS(cplb_data) $LIBRARIES(cplb_data))
INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS
    ( $OBJECTS(constdata) $LIBRARIES(constdata))

INPUT_SECTION_ALIGN(1)
INPUT_SECTIONS
    ( $OBJECTS(ctor) $LIBRARIES(ctor) )
} >MEM_PROGRAM

stack
{
    INPUT_SECTIONS $OBJECTS(stack) )
} >MEM_STACK

heap
{
    // Allocate a heap for the application
    ldf_heap_space = .;
    ldf_heap_end =
        ldf_heap_space + MEMORY_SIZEOF(MEM_HEAP) - 1;
    ldf_heap_length = ldf_heap_end - ldf_heap_space;
} >MEM_HEAP

OVERLAY_INPUT {
    // The output archive file "overlay1.ovl" will
    // contain the code and symbol table for this
    // overlay

```

LDF Programming Examples

```
OVERLAY_OUTPUT( overlay1.ovl )

    /* Only take the code from the file overlay1.doj.
       If this code needs data, it must be either the INPUT of a
       data overlay or the INPUT to non-overlay data memory. */
    INPUT_SECTIONS(overlay1.doj( program))

    // Tell the linker that all of the code in the overlay must
    // fit into the “run” memory all at once. ALGORITHM(ALL_FIT)
    // allows the linker to break the code into several
    // overlays as necessary (in the event that not all
    // of the code fits).

    ALGORITHM( ALL_FIT )
    SIZE(0x100)

} > mem_ovly

    // This is the second overlay. Note that these
    // OVERLAY_INPUT commands must be contiguous in the LDF
    // to occupy the same “run-time” memory.
    OVERLAY_INPUT {
    OVERLAY_OUTPUT( overlay2.ovl )
    INPUT_SECTIONS(overlay2.doj( program)))
    ALGORITHM( ALL_FIT )
    SIZE( 0x100)
} > mem_ovly

} > program

/* The instructions generated by the linker in the .plit
   section must be placed in non-overlay memory. Here is
   the sole specification telling the linker where to
```

```
place these instructions */

.plit {      // linker insert instructions here
} > MEM_PROGRAM

DXE_DATA1 {
    INPUT_SECTIONS ( $OBJECTS(data1) $LIBRARIES(data1))
    INPUT_SECTION_ALIGN(1)
    INPUT_SECTIONS( $OBJECTS(constdata) $LIBRARIES(constdata))
    INPUT_SECTION_ALIGN(1)
    INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
} >MEM_PROGRAM

stack
{
    INPUT_SECTIONS( $OBJECTS(stack) )
} >MEM_STACK

heap
{
    // Allocate a heap for the application
    ldf_heap_space = .;
    ldf_heap_end =
        ldf_heap_space + MEMORY_SIZEOF(HEAP) - 1;
    ldf_heap_length = ldf_heap_end - ldf_heap_space;
} >MEM_HEAP
}
```


3 EXPERT LINKER

The linker (`linker.exe`) combines object files into a single executable object module. Using the linker, you can create a new Linker Description File (LDF), modify an existing LDF, and produce an executable file(s). The linker is described in Chapter 1, “[Linker](#)”, of this manual.

The *Expert Linker* is a graphical tool that simplifies complex tasks such as memory map manipulation, code and data placement, overlay and shared memory creation, and C stack/heap usage. This tool complements the existing VisualDSP++ .LDF file format by providing a visualization capability enabling new users to take immediate advantage of the powerful LDF format flexibility.



Graphics in this chapter demonstrate Expert Linker features. Some graphics show features not available to all DSP families. DSP-specific features are noted in neighboring text.

This chapter contains:

- “[Expert Linker Overview](#)” on page 3-2
- “[Launching the Create LDF Wizard](#)” on page 3-4
- “[Expert Linker Window Overview](#)” on page 3-9
- “[Using the Input Sections Pane](#)” on page 3-10
- “[Using the Memory Map Pane](#)” on page 3-16
- “[Managing Object Properties](#)” on page 3-40

Expert Linker Overview

Expert Linker is a graphical tool that allows you to:

- Define a DSP target's memory map
- Place a project's object sections into that memory map
- View how much of the stack or heap has been used after running the DSP program

Expert Linker takes available project information in an `.LDF` file as input (object files, LDF macros, libraries, and target memory description) and graphically displays it. You can then use drag-and-drop action to arrange the object files in a graphical memory mapping representation. When you are satisfied with the memory layout, you can generate the executable file (`.DXE`) via VisualDSP++ project options.



You can use default `.LDF` files that come with VisualDSP++, or you can use the Expert Linker interactive wizard to create new `.LDF` files.

When you open Expert Linker in a project that has an existing `.LDF` file, Expert Linker parses the `.LDF` file and graphically displays the DSP target's memory map and the object mappings. The memory map displays in the **Expert Linker** window.

Use this display to modify the memory map or the object mappings. When the project is about to be built, Expert Linker saves the changes to the `.LDF` file.

Expert Linker is able to show graphically how much space is allocated for program heap and stack. After loading and running the program, Expert Linker can show how much of the heap and stack has been used. You can interactively reduce the amount of space allocated to heap or stack if they are using too much memory. This frees up memory to store other things like DSP code or data.

There are three ways to launch the Expert Linker from VisualDSP++:

- Double-click the .LDF file in the **Project** window.
- Right-click the .LDF file in the **Project** window to display a menu and then choose **Open in Expert Linker**.
- From the VisualDSP++ main menu, choose **Tools -> Expert Linker-> Create LDF**.

The Expert Linker window appears.

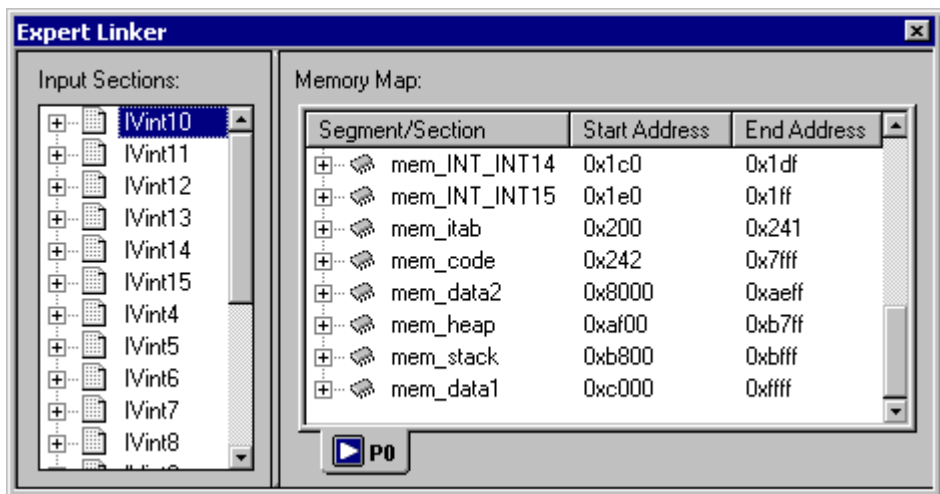


Figure 3-1. Expert Linker Window

Launching the Create LDF Wizard

From the VisualDSP++ main menu, choose **Tools -> Expert Linker-> Create LDF** to invoke a wizard that allows you to create and customize a new .LDF file. The **Create LDF** option is mostly used when creating a new project.

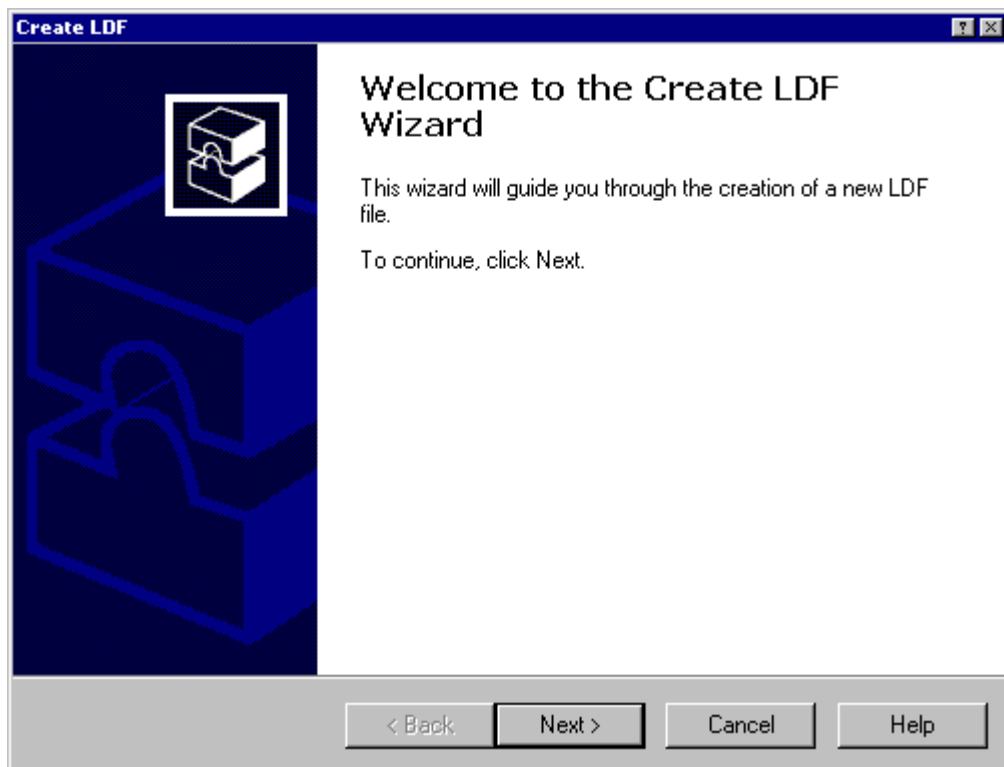


Figure 3-2. Welcome Page of the Create LDF Wizard

If there is already an LDF in the project, you are prompted to confirm whether to create a new .LDF file to replace the existing one. This menu command is disabled when VisualDSP++ does not have a project opened or when the project's processor-build target is not supported by Expert Linker. Press **Next** to run the wizard.

Step 1: Specifying Project Information

The first wizard window is displayed.

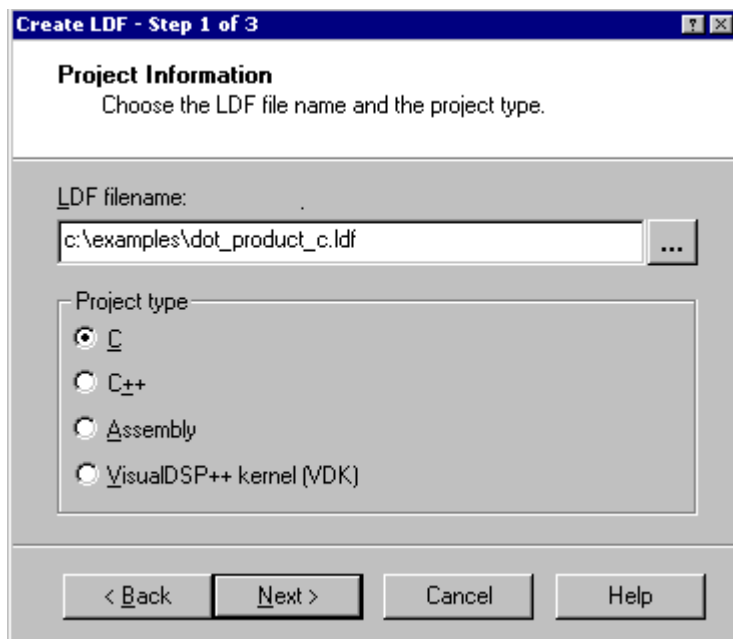


Figure 3-3. Selecting File Name and Project Type

You may use or specify the default file name for the .LDF file. The default file name is `project_name.ldf` where `project_name` is the name of the currently opened project.

Launching the Create LDF Wizard

The **Project type** selection specifies whether the LDF is for a C, C++, assembly, or a VDK project. The default setting depends on the source files in the project. For example, if there are .C files in the project, the default is C; if there is a VDK.H file in the project, the default is VDK, and so on. This setting determines which template is used as a starting point.

Press **Next**.

Step 2: Specifying System Information

You must now choose whether the project is for a single-processor system or a multiprocessor (MP) system.

- For a single-processor system, the **Processors** list shows only one processor and the MP address columns do not display.
- For a multiprocessor system, you must add the list of processors, name each processor, and set the processor order, which will determine each processor's MP memory address range.

Processor type identifies the DSP system's processor architecture. This is derived from the processor target specified via the **Project Options** dialog box in VisualDSP++.

If you select **Set up system from debug session settings**, the processor information (number of processors and the processor names) will be filled automatically from the current settings in the debug session. This field is gray when the current debug session is not supported by the Expert Linker.

You can also specify the **Output file name** and the **Executables to link against** (object libraries, macros, etc.).

When you select a processor in the **Processors** list, the output file name and the list of executables to link against for that processor appear to the right of the **Processors** list. You can change these files by typing a new file name. The file name may include a relative path and/or an LDF macro.

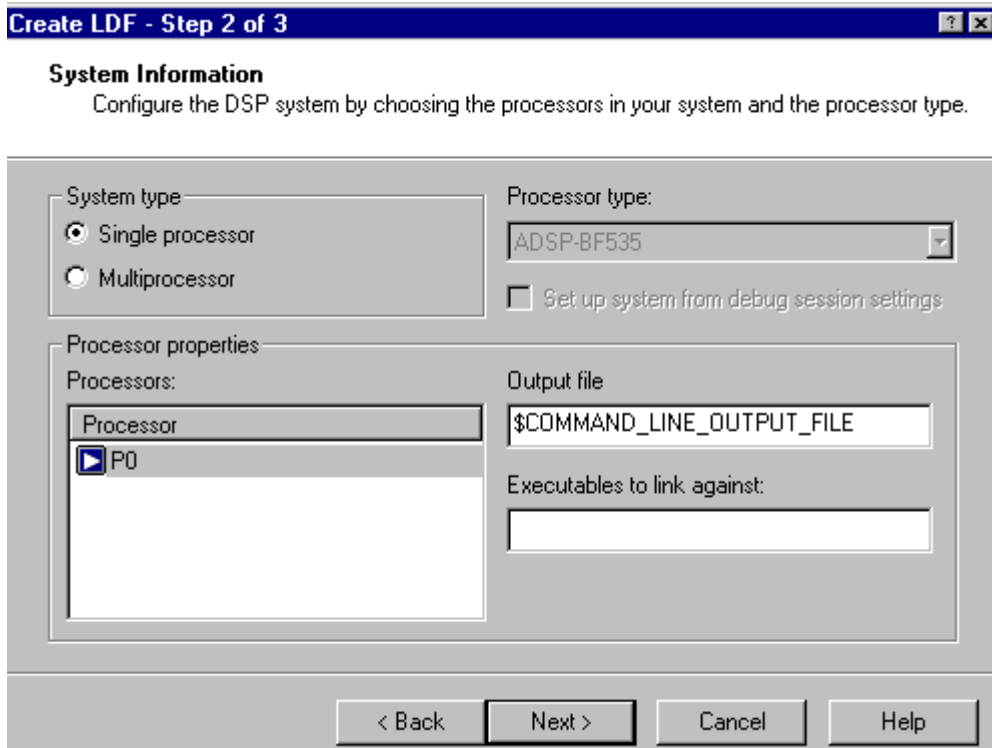



Figure 3-4. Selecting System and Processor Types

For multiprocessor systems, the window shows the list of processors in the project and the MP address range for each processor. In addition, if Expert Linker can detect the ID of the processor, the processor is placed in the correct position in the processor list.

 The MP address range is available only for processors that have MP memory space, such as SHARC family DSPs.

Press **Next** to advance to the **Wizard Completed** page.

Step 3: Completing the LDF Wizard

The system displays the **Wizard Completed** page. From this page you can go back and verify and/or modify selections made up to this point.

When you click the **Finish** button, Expert Linker copies a template .LDF file to the same directory as the project file and adds it to the current project. The Expert Linker window appears, displaying the contents of the new .LDF file.

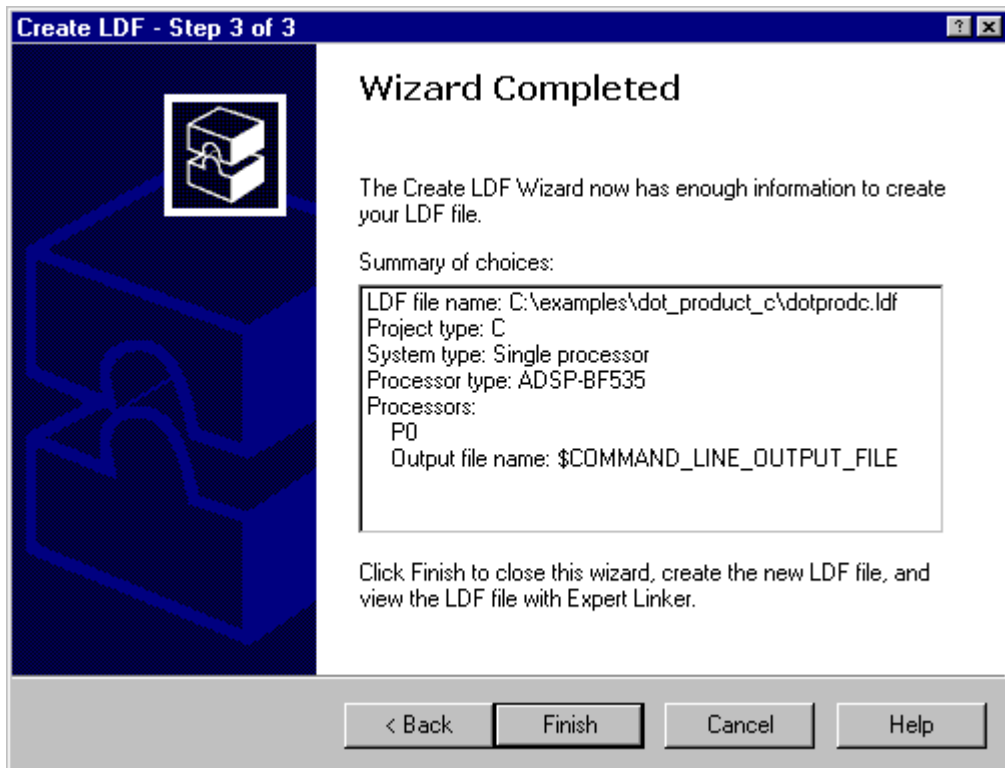


Figure 3-5. Wizard Completed Page of the Create LDF Wizard

Expert Linker Window Overview

The Expert Linker window contains two panes:

- The **Input Sections** pane provides a tree display of the project's input sections (see [“Using the Input Sections Pane” on page 3-10](#)).
- The **Memory Map** pane displays each memory map in a tree or graphical representation (see [“Using the Memory Map Pane” on page 3-16](#)).

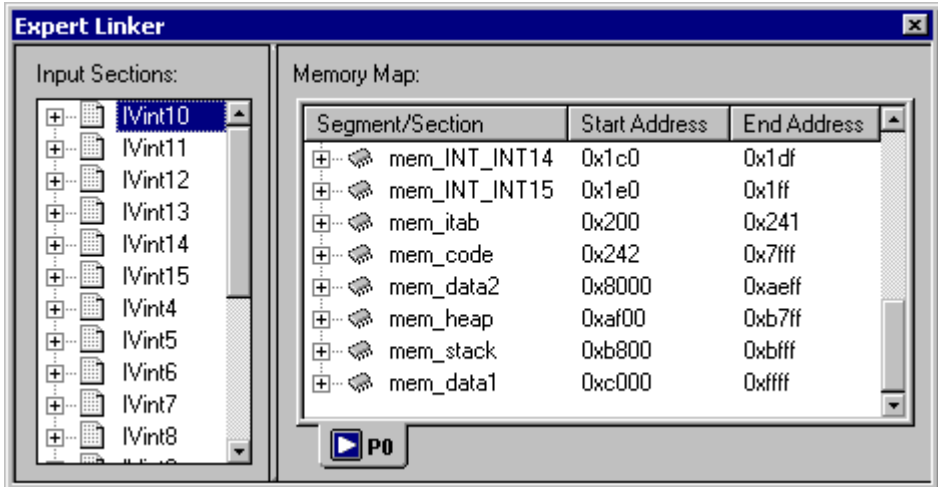



Figure 3-6. Expert Linker Window

Using commands in the LDF, the linker reads the input sections from object files (.OBJ) and places them in output sections in the executable file. The LDF defines the DSP's memory and indicates where within that memory the linker is to place the input sections.

Using drag-and-drop, you can map an input section to an output section in the memory map. Each memory segment may have one or more output sections under it. Input sections that have been mapped to an output sec-

Using the Input Sections Pane

tion display under that output section. For more information, refer to [“Using the Input Sections Pane” on page 3-10](#) and [“Using the Memory Map Pane” on page 3-16](#).

-  Access various Expert Linker functions with your mouse. Right-click to display appropriate menus and make selections.

Using the Input Sections Pane

The **Input Sections** pane ([Figure 3-6](#)) initially displays a list of all the input sections referenced by the .LDF file, and all input sections contained in the object files and libraries. Under each input section, there may be a list of LDF macros, libraries, and object files contained in that input section. You can add or delete input sections, LDF macros, or objects/library files in this pane.

Using the Input Sections Menu

Right-click an object in the **Input Sections** pane, a menu appears. See [Figure 3-7 on page 3-11](#).

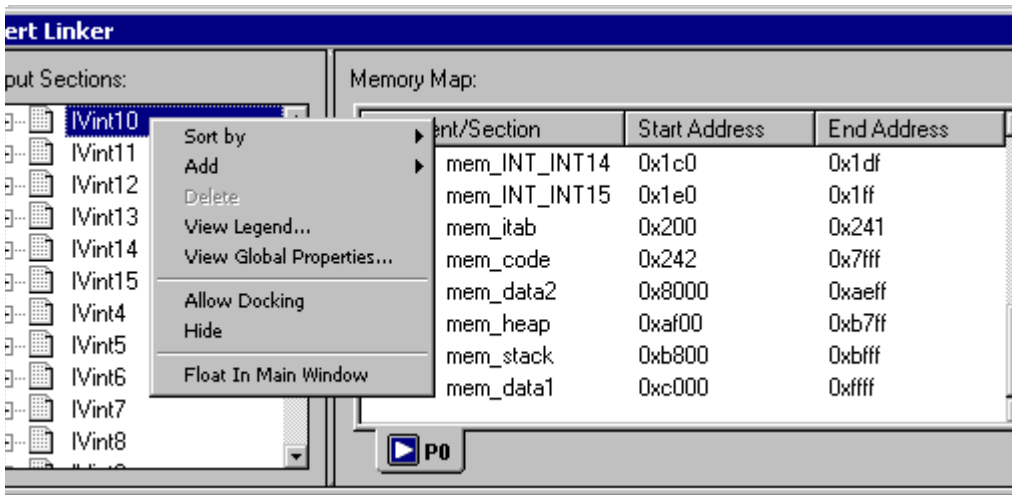


Figure 3-7. Input Sections Right-Click Menu

The main menu functions include:

- **Sort by** – Sorts objects by input sections or LDF macros. These selections are mutually exclusive.
- **Add** – Adds input sections, object/library files, and LDF macros. Appropriate menu selections are grayed out if you right-click on a position (area) in which you cannot create a corresponding object.
You can create an input section as a shell, without object/library files or LDF macros in it. You can even map this section to an output section. However, input sections without data are grayed out.
- **Delete** – Deletes the selected object (input section, object/library file, or LDF macro).
- **View Legend** – Displays the **Legend** dialog box showing icons and colors used by the Expert Linker.

Using the Input Sections Pane


- **View Section Contents** – Opens the **Section Contents** dialog box, which displays the section contents of the object file, library file, or .DXE file. This command is available only after you link or build the project and then right-click on an object or output section.
- **View Global Properties** – Displays the **Global Properties** dialog box that provides the map file name (of the map file generated after linking the project) as well as access to various processor and setup information (see [Figure 3-31 on page 3-41](#)).

Mapping an Input Section to an Output Section

Using the Expert Linker, you can map an input section to an output section. To do that, use Windows drag-and-drop action—click on the input section, drag the mouse pointer to an output section, and then release the mouse button to drop the input section onto the output section.

All objects, such as LDF macros or object files under that input section, map to the output section. Once an input section has been mapped, the icon next to the input section changes to denote that it is mapped.

If an input section is dragged onto a memory segment with no output section in it, an output section with a default name is automatically created and displayed.

A red “x” on an icon (for example, ) indicates the object/file is not mapped. Once an input section has been completely mapped (that is, all object files that contain the section are mapped), the icon next to the input section changes to indicate that it has been mapped; the “x” disappears. See [Figure 3-8](#).

While dragging the input section, the icon changes to a circle with a diagonal slash if it is over an object where you are not allowed to drop the input section.

Viewing Icons and Colors

Use the **Legend** dialog box to displays all possible icons in the tree pane and a short description of each icon. (Figure 3-8)

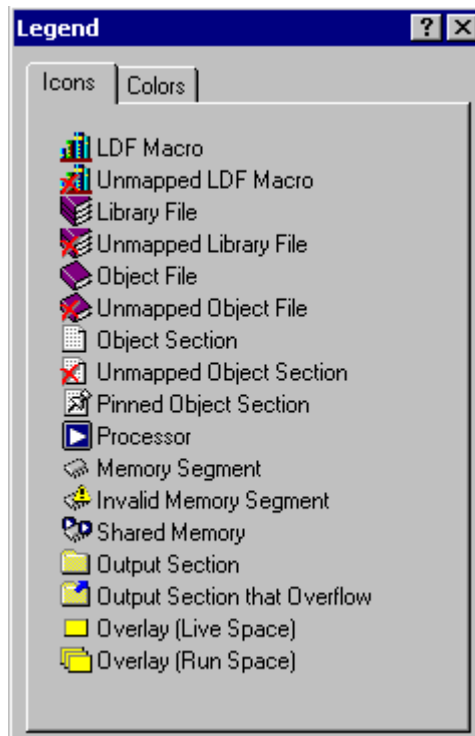



Figure 3-8. Legend Dialog Box – Icons Page

 The red “x” on an icon indicates this object/file is not mapped.

Click the **Colors** tab to view the **Colors** page (Figure 3-9). This page contains a list of colors used in the graphical memory map view; each item’s color can be customized. The list of displayed objects depends on the DSP family.

Using the Input Sections Pane

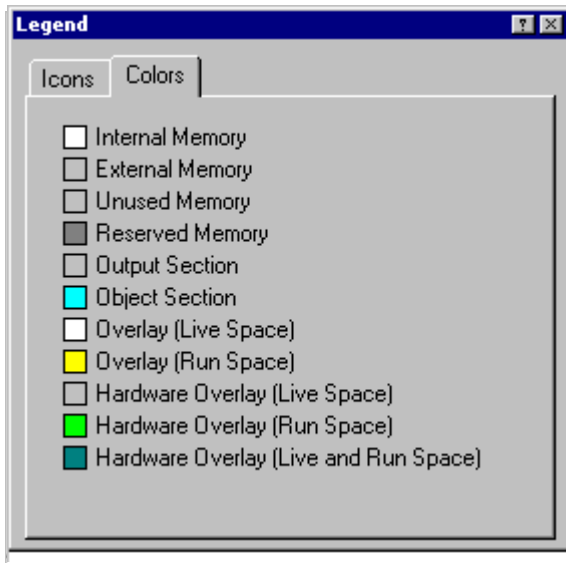


Figure 3-9. Legend Dialog Box – Colors Page

To change a color:

1. Double-click the color. You can also right-click on a color and select **Properties**.

The system displays the **Select a Color** dialog box ([Figure 3-10](#)).

2. Select a color and click **OK**.

Click **Other** to select other colors from the advanced palette.

Click **Reset** to reset all memory map colors to the default colors.



Figure 3-10. Selecting Colors

Sorting Objects

You can sort objects in the **Input Sections** pane by input sections (default) or by LDF macros, like `$OBJECTS` or `$COMMAND_LINE_OBJECTS`. The **Input Sections** and **LDF Macros** menu selections are mutually exclusive—only one can be selected at a time. For example,

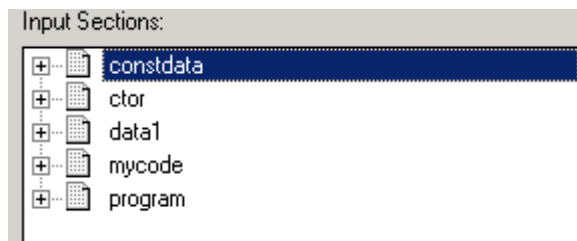


Figure 3-11. Expert Linker Window – Sorted by Input Sections

Other macros, object files, or libraries may appear under each macro. Under each object file are input sections contained in that object file.



When the tree is sorted by LDF macros, only input sections can be dragged onto output sections.

Using the Memory Map Pane

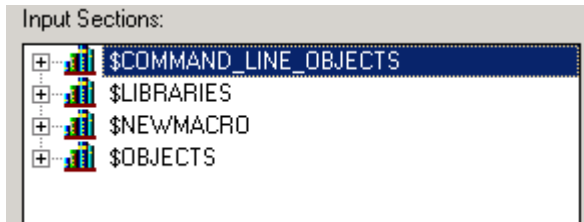


Figure 3-12. Expert Linker Window – Sorted by LDF Macros

Using the Memory Map Pane

In an .LDF file, the linker's `MEMORY()` command defines the target system's physical memory. Its argument list partitions memory into memory segments and assigns labels to each, specifying start and end addresses, memory width, and memory type (such as program, data, stack, and so on). It connects your program to the target system. The `OUTPUT()` command directs the linker to produce an executable (.DXX) file and specifies its file name.

The combo box (located to the right of the **Memory Map** label) applies only to ADSP-218x DSPs.

The **Memory Map** pane has tabbed pages. You can page through the memory maps of the processors and shared memories to view their makeup. There are two viewing modes—a **tree** view and a **graphical** view.

Select these views and other memory map features by means of the right-click (context) menu. All procedures involving memory map handling assume the Expert Linker window is open.

The **Memory Map** pane displays a tooltip when you move the mouse cursor over an object in the display. The tooltip shows the object's name, address, and size. The system also uses representations of overlays, which display in “run” space and “live” space.

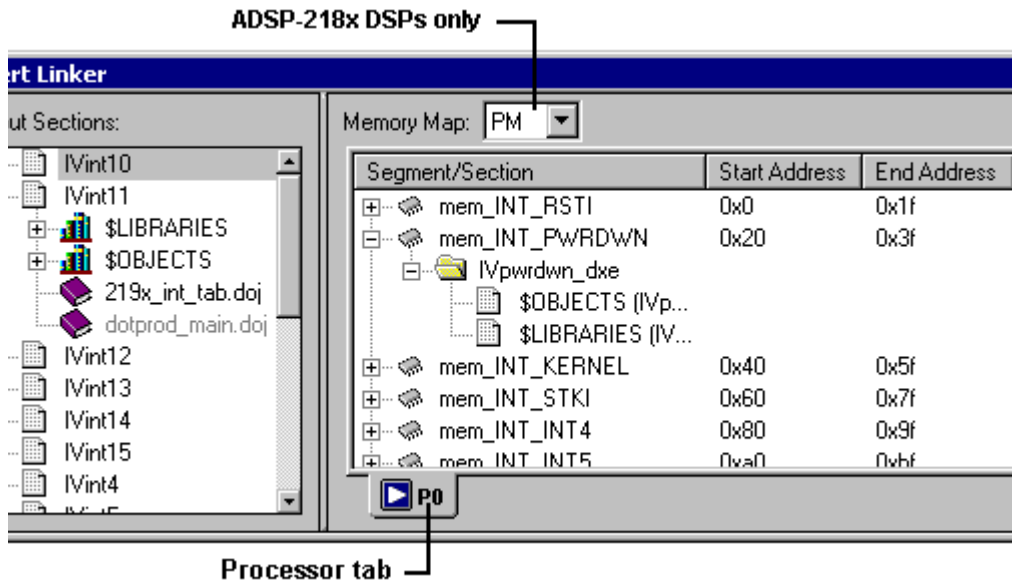


Figure 3-13. Expert Linker Window – Memory Map

Invalid Memory Segment Notification:

When a memory segment is invalid (for example, when a memory range overlaps another memory segment), the memory width is invalid. The tree shows an **Invalid Memory Segment** icon (also see [Figure 3-8 on page 3-13](#)). Move the mouse pointer over the icon and a tooltip displays a message describing why the segment is invalid.

Using the Context Menu

Display the context menu by right-clicking in the **Memory Map** pane. The menu allows you to select and perform major functions. The available right-click menu commands are listed below.

Using the Memory Map Pane

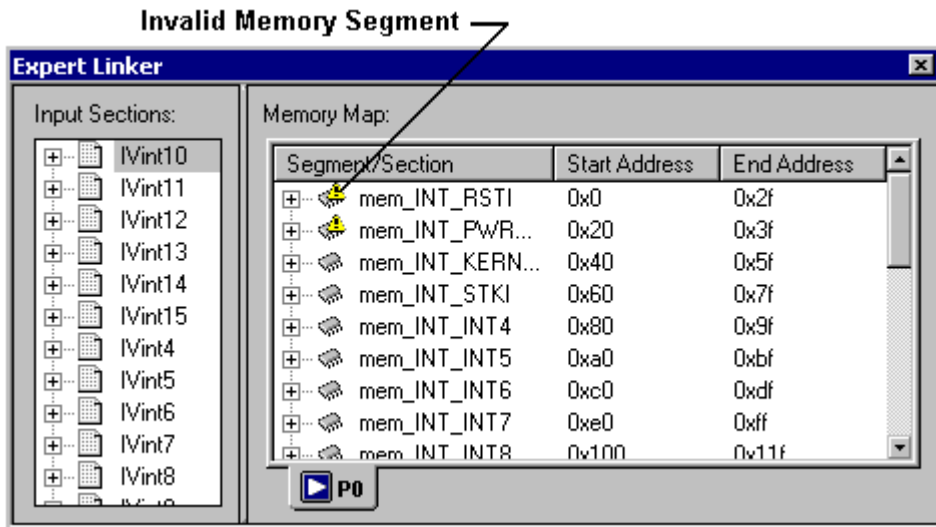


Figure 3-14. Memory Map With Invalid Memory Segments

View Mode

- **Memory Map Tree** – Displays the memory map in a tree representation (see [Figure 3-15 on page 3-21](#)).
- **Graphical Memory Map** – Displays the memory map in graphical blocks (see [Figure 3-16 on page 3-23](#)).

View

- **Mapping Strategy (Pre-Link)** – Displays the memory map that shows where you plan to place your object sections.
- **Link Results (Post-Link)** – Displays the memory map that shows where the object sections were actually placed.

New

- **Memory Segment** – Allows you to specify the name, address range, type, size, and so on for memory segments you want to add.
- **Output Section** – Adds an output section to the selected memory segment. (Right-click on the memory segment to access this command.) If you do not right-click on a memory segment, this option is disabled.

The options are: **Name**, **Overflow** (name of output section to catch overflow), **Packing**, and **Number of bytes** (number of bytes to be reordered at one time).

- **Shared Memory** – Adds a shared memory to the memory map.
- **Overlay** – Invokes a dialog box that allows you to add a new overlay to the selected output section or memory segment. The selected output section is the new overlay's run space (see [Figure 3-42 on page 3-58](#)).

Delete – Deletes the selected object.

Pin to Output Section – Appears only when you right-click on an object section that is part of an output section specified to overflow to another output section. Pinning an object section to an output section prevents it from overflowing to another output section.

View Section Contents – Available only after linking or building the project and then right-clicking on an input or object section. This view invokes a dialog box that displays the contents of the input or output section (see [Figure 3-28 on page 3-37](#)).

View Symbols – Available only after linking the project and then right-clicking on a processor, overlay, or input section. Invokes a dialog box that displays the symbols for the project, overlay, or input section (see [Figure 3-31 on page 3-41](#)).

Using the Memory Map Pane

Properties – Displays a **Properties** dialog box for the selected object. The **Properties** menu is context-sensitive; different properties display for different objects. Right-click a memory segment and choose **Properties** to specify a memory segment's attributes (name, start address, end address, size, width, memory space, PM/DM/(BM), RAM/ROM, and internal/external flag).

View Legend – Displays the **Legend** dialog box showing tree view icons and a short description of each icon. The **Colors** page lists the colors used in the graphical memory map. You can customize each object's color. See [Figure 3-8 on page 3-13](#) and [Figure 3-9 on page 3-14](#).

View Global Properties – Displays a **Global Properties** dialog box that lists the map file generated after linking the project. It also provides access to some processor and setup information (see [Figure 3-32 on page 3-42](#)).

Tree View Memory Map Representation

In the tree view (right-click and choose **View Mode -> Memory Map Tree**), the memory map displays with memory segments at the top level.

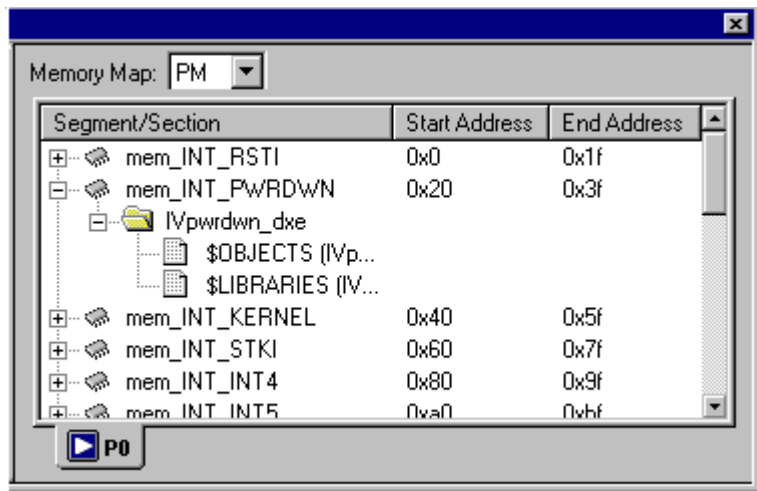


Figure 3-15. Expert Linker Window – Memory Map

Each memory segment may have one or more output sections under it. Input sections mapped to an output section display under that output section.

The start address and size of the memory segments display in separate columns. If available, the start address and the size of each output section are displayed (for example, after linking the project).

Graphical View Memory Map Representation

In the graphical view (selected by right-clicking and choosing **View Mode** -> **Graphical Memory Map**), the graphical memory map displays the processor's hardware memory map (refer to your DSP's *Hardware Reference* manual or data sheet). Each hardware memory segment contains a list of user-defined memory segments.

View the memory map from two perspectives—pre-link view and post-link view (see [“Specifying Pre- and Post-Link Memory Map View” on page 3-28](#)).

[Figure 3-16](#) shows an example of a graphical memory map.

In graphical view, the memory map comprises blocks of different colors, representing memory segments, output sections, objects, and so on. The memory map is drawn with these rules:

- An output section is represented as a vertical header with a group of objects to the right of it.
- A memory segment's border and text change to red (from its normal black color) to indicate that it is invalid. When you move the mouse pointer over the invalid memory segment, a tooltip displays a message, describing why the segment is invalid.
- The height of the memory segments is not scaled as a percentage of the total memory space. However, the width of the memory segments is scaled as a percentage of the widest memory.
- Object sections are drawn as horizontal blocks stacked on top of each other. Before linking, the object section sizes are not known and display in equal sizes within the memory segment. After link-

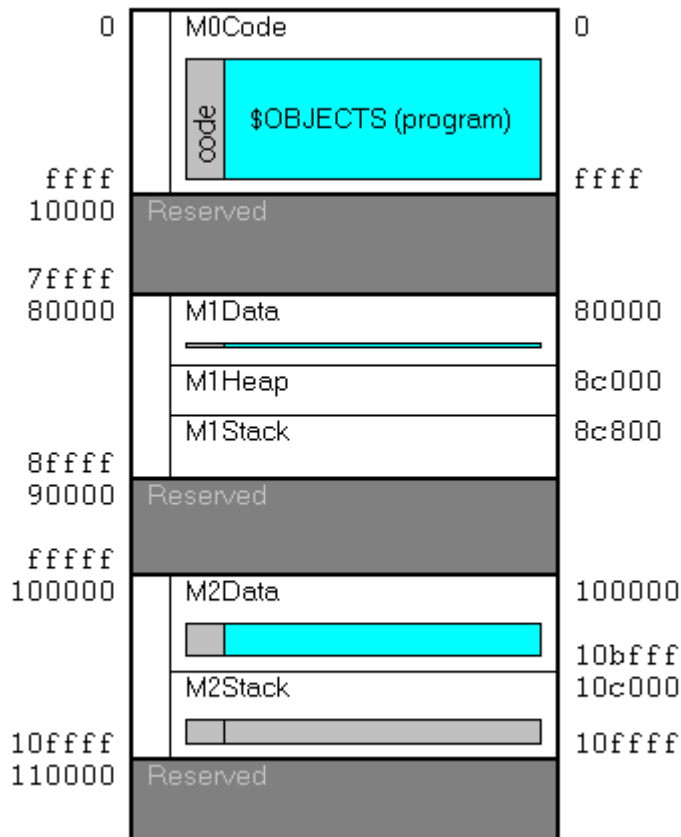


Figure 3-16. Graphical Memory Map Representation

ing, the height of the objects is scaled as a percentage of the total memory segment size. Object section names display only when there is enough room for display.

- Addresses are ordered in ascending order from top to bottom.

Using the Memory Map Pane

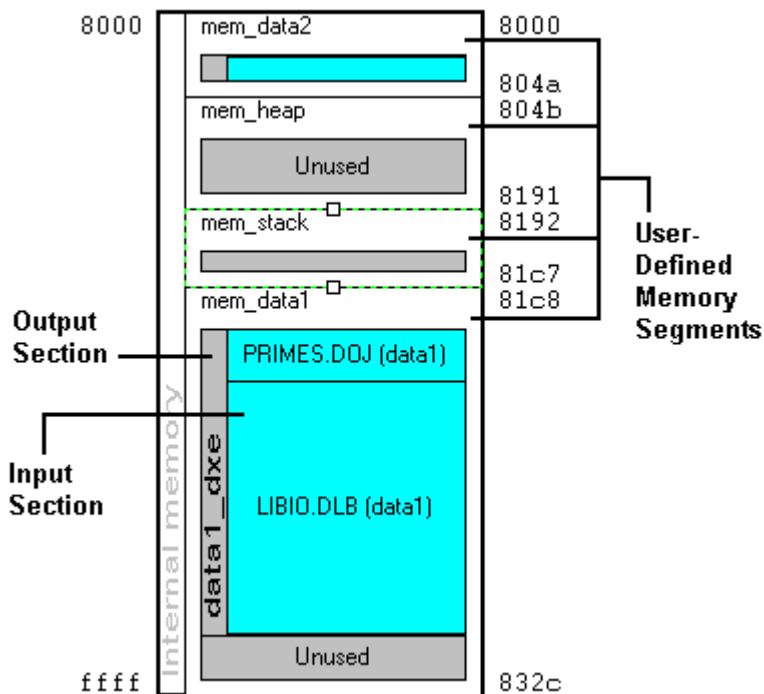


Figure 3-17. Viewing Sections and Segments in Memory Map

Three buttons at the top right of the **Memory Map** pane permit zooming. If there is not enough room to display the memory map when zoomed in, horizontal and/or vertical scroll bars allow you to view the entire memory map (for more information, see [“Zooming In and Out on the Memory Map”](#) on page 3-28).

You can drag and drop any object except memory segments. See [Figure 3-19](#).

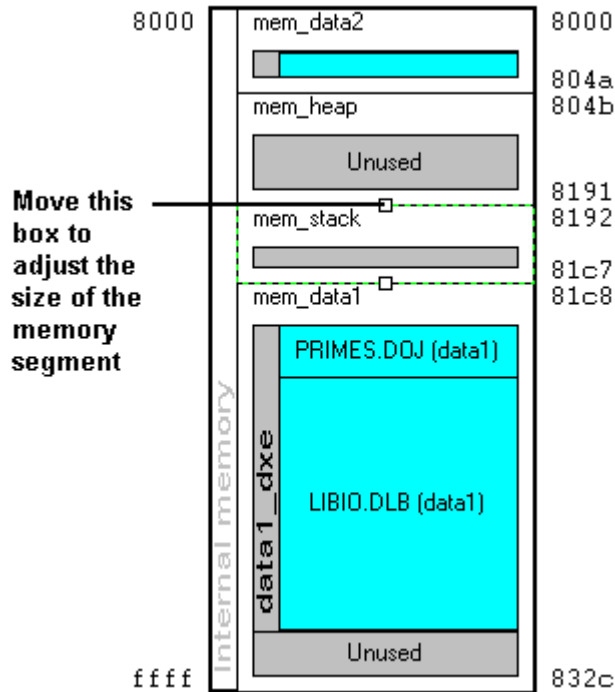


Figure 3-18. Adjusting the Size of a Memory Segment

Select a memory segments to display its border. Drag the border to change the memory segment's size. The size of the selected and adjacent memory segments change.

Using the Memory Map Pane

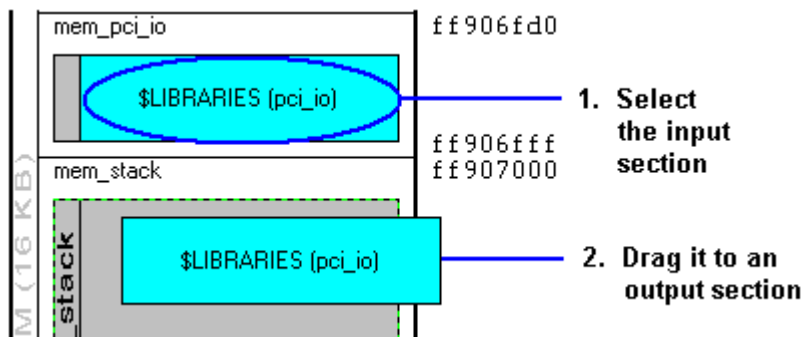


Figure 3-19. Dragging and Dropping an Object

When the mouse pointer is on top of the box, the resize cursor appears as follows.



When an object is selected in the memory map, it highlights as shown in [Figure 3-20 on page 3-27](#). If you move the mouse pointer over an object in the graphical memory map, a yellow tooltip appears displaying the information about the object (such as name, address, and size).

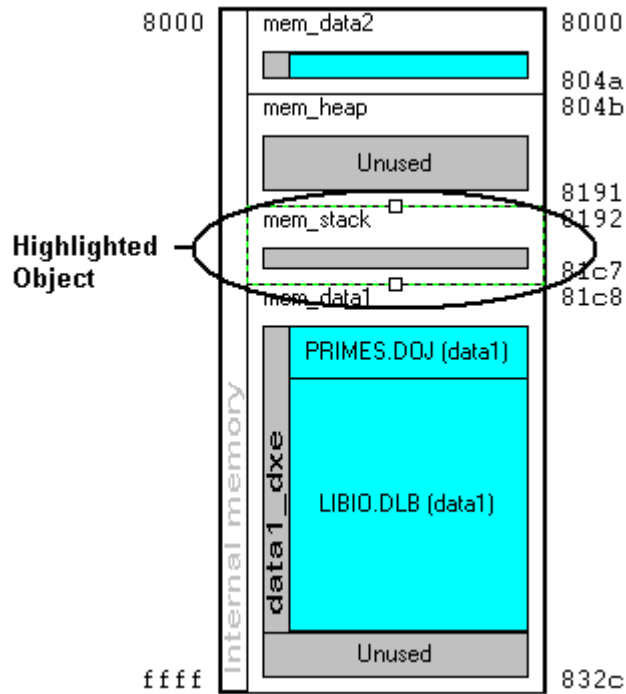


Figure 3-20. A Highlighted Memory Segment in the Memory Map

Specifying Pre- and Post-Link Memory Map View

View the memory map from two perspectives: pre-link view and post-link view. Pre-link view is typically used to place input sections. Post-link view is typically used to view where the input sections were placed after linking the project. Other information (such as the sizes of each section, symbols, and the contents of each section) is available after linking.

- To enable pre-link view from the **Memory Map** pane, right-click and choose **View and Mapping Strategy (Pre-Link)**. [Figure 3-22 on page 3-30](#) illustrates memory map before linking.
- To enable post-link view from the **Memory Map** pane, right-click and choose **View and Link Results (Post-Link)**. [Figure 3-23 on page 3-31](#) illustrates memory map after linking.

Zooming In and Out on the Memory Map

From the **Memory Map** pane, you can zoom in or out incrementally or zoom in or out completely. Three buttons at the top right of the pane perform zooming operations. Horizontal and/or vertical scroll bars appear when there is not enough room to display a zoomed memory map in the **Memory Map** pane.

To:

- Zoom in, click on the magnifying glass icon with the + sign above the upper right corner of the memory map window.
- Zoom out, click on the magnifying glass icon with the - sign above the upper right corner of the memory map window.
- Exit zoom, click on the magnifying glass icon with the “x” above the upper right corner of the memory map window.

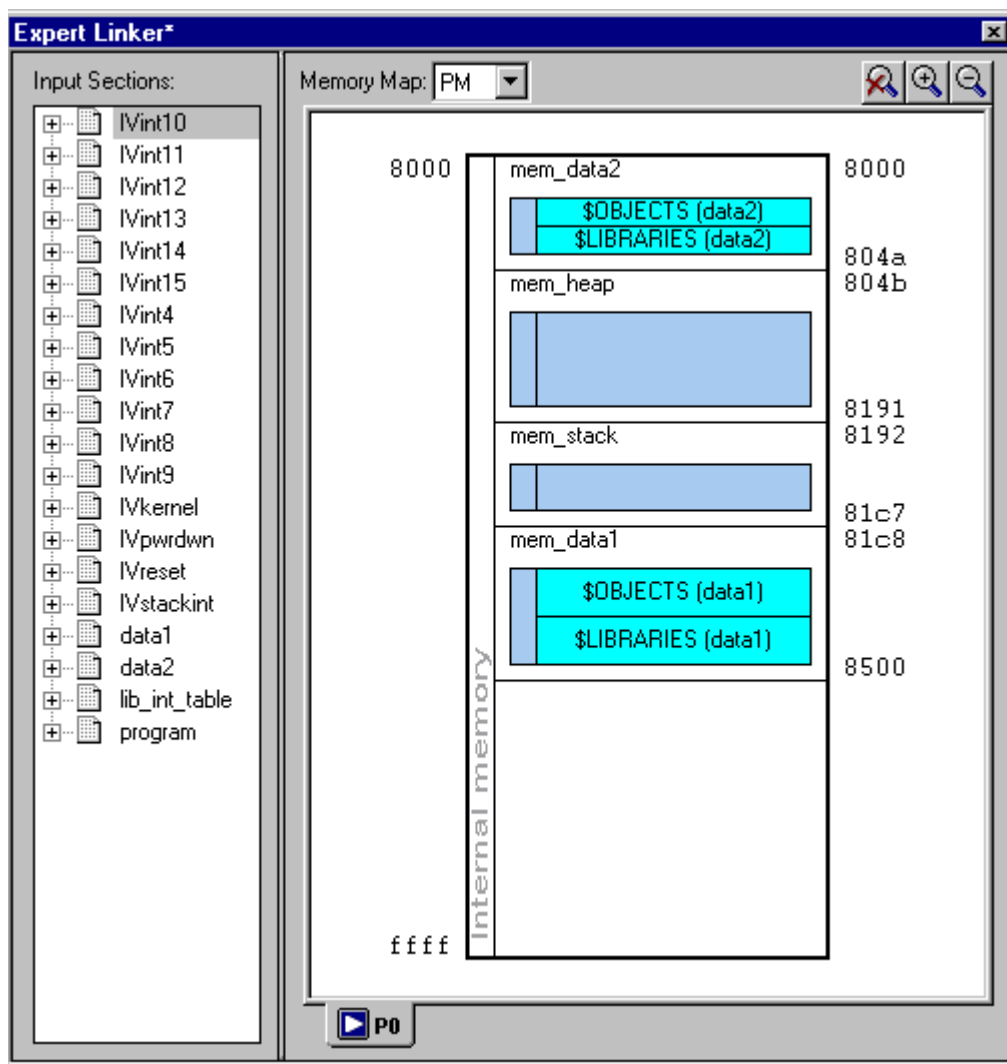


Figure 3-21. Memory Map Pane in Pre-Link View

Using the Memory Map Pane

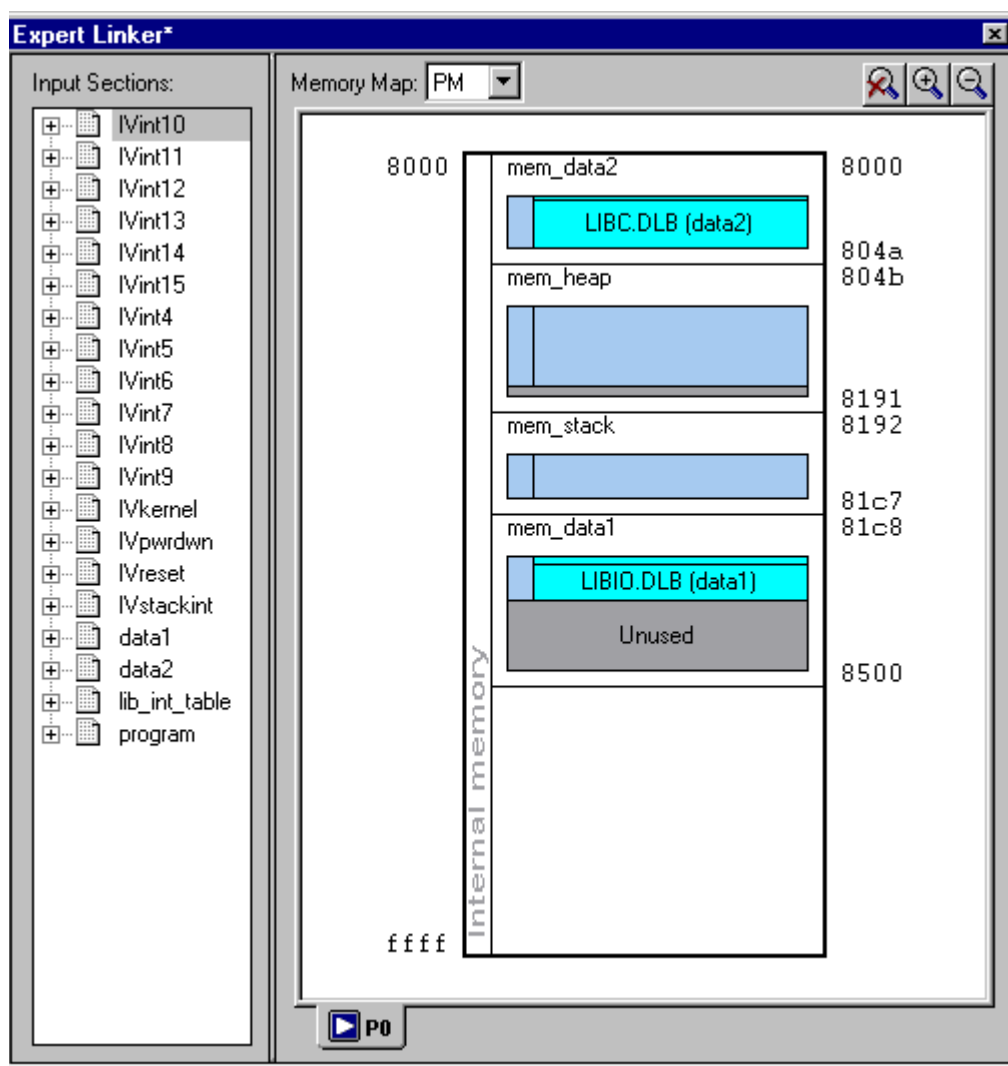


Figure 3-22. Memory Map Pane in Post-Link View

- View a memory object by itself by double-clicking on the memory object.
- View the memory object containing the current memory object by double-clicking on the white space around the memory object

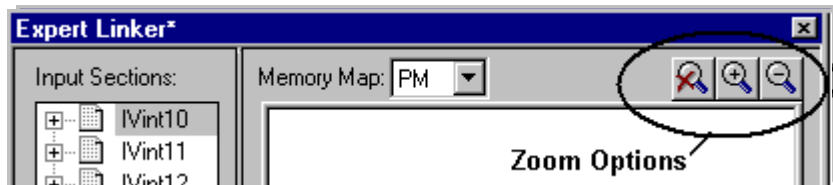


Figure 3-23. Memory Map – Zoom Options

Inserting a Gap into a Memory Segment

A gap may be inserted into a memory segment in the graphical memory map.

To insert a gap:

1. Right-click on a memory segment.
2. Choose **Insert gap**. The **Insert Gap** dialog box appears.

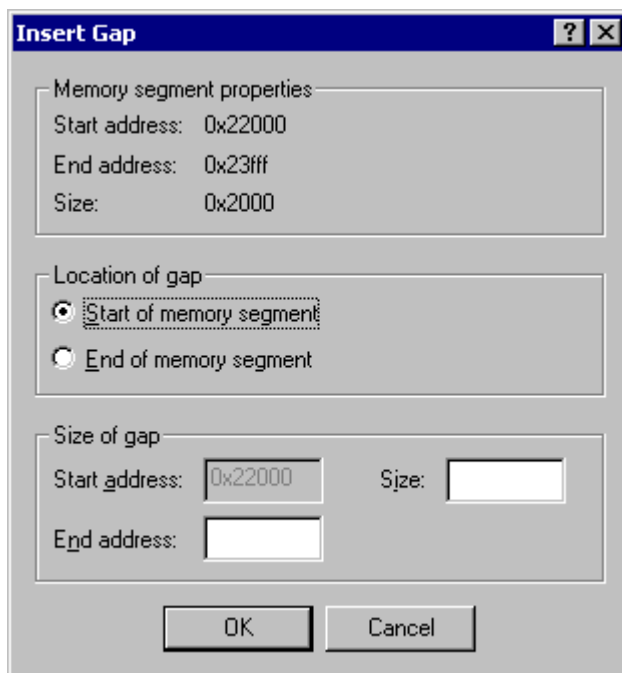


Figure 3-24. Insert Gap Dialog Box

You may insert a gap at the start of the memory segment or the end of it. If the start is chosen, the **Start address** for the gap is grayed out and you must enter an end address or size (of the gap). If the end is chosen, the **End address** of the gap is grayed out and you must enter a start address or size.

Working With Overlays

Overlays appear in the memory map window in two places: “run” space and “live” space. Live space is where the overlay is stored until it is swapped into run space. Because multiple overlays can exist in the same “run” space, the overlays display as multiple blocks on top of each other in cascading fashion.

[Figure 3-26](#) shows an overlay in live space, and [Figure 3-27](#) shows an overlay in run space.

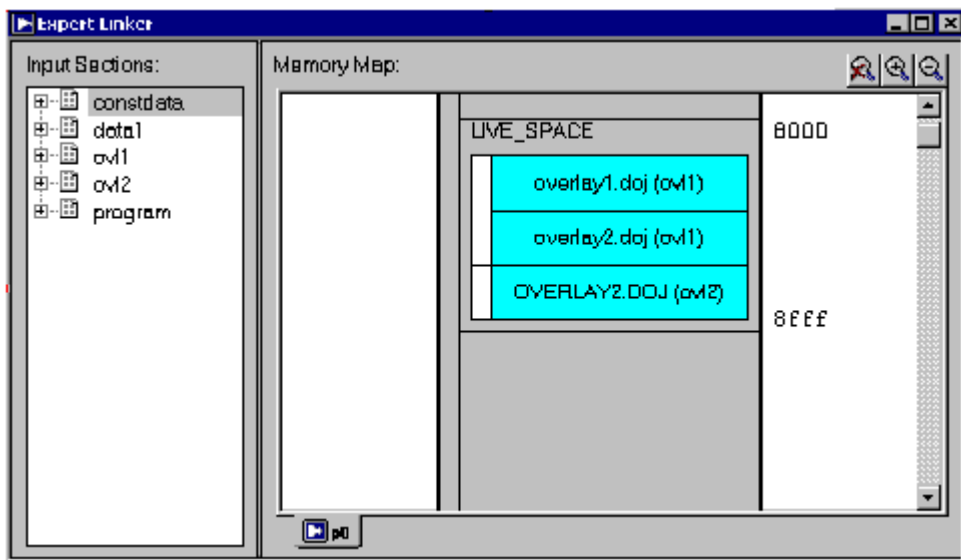


Figure 3-25. Graphical Memory Map Showing Overlay Live Space

Using the Memory Map Pane

Overlays in a “run” space display one at a time in the graphical memory map. The scroll bar next to an overlay in “run” space allows you to specify an overlay to be shown on top. You can drag the overlay on top to another output section to change the “run” space for an overlay.

Click the up arrow or down arrow button in the header to display previous or next overlay in “run” space. Click the browse button to display the list of all available overlays. The header shows the number of overlays in this “run space” and the current overlay number.

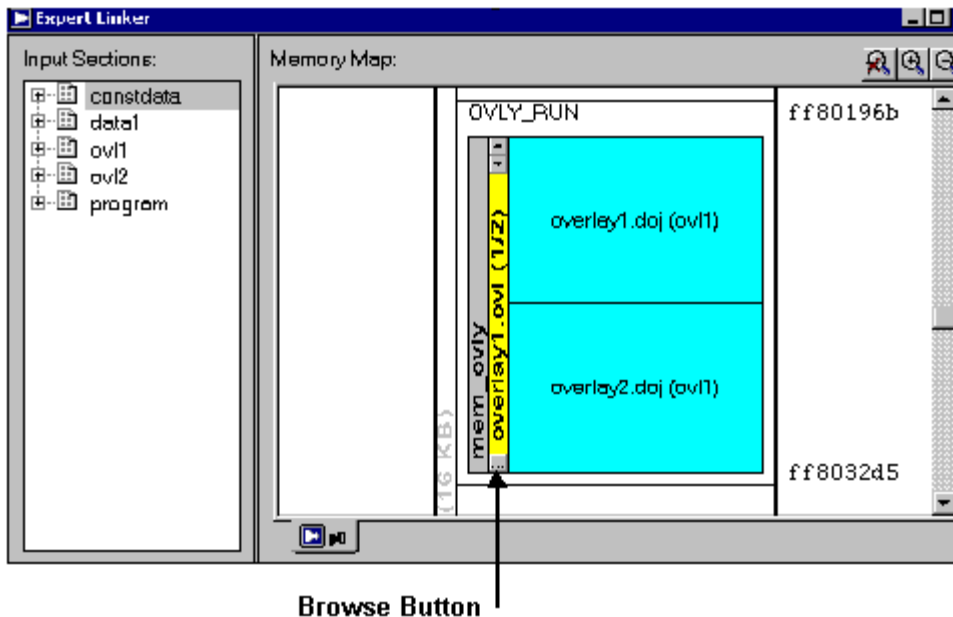


Figure 3-26. Graphical Memory Map Showing Overlay Run Space

To create an overlay in the “run” space:

1. Right-click on an output section.
2. Choose New -> Overlay.

3. Select the “live” space from the **Overlay Properties** dialog box. The new overlay displays in the “run” and “live” spaces in two different colors in the memory map.
4. Drag the “live” space overlay to a different output section. This can change the “live” to “run” space.

Viewing Section Contents

You can view the contents of an input section or an output section. You must specify the particular memory address and the display’s format.

This capability employs the `elfdump.exe` utility to obtain the section contents and display it in a window similar to a memory window in VisualDSP++. Multiple **Section Contents** dialog boxes may be displayed.

To display the contents of an output section:

1. In the **Memory Map** pane, right-click an output section.
2. Choose **View Section Contents** from the menu.
The **Section Contents** dialog box appears.

By default, the memory section content displays in **Hex** format.
3. Right-click anywhere in the section view to display a menu with these selections:
 - **Go To** – Displays an address in the window.
 - **Select Format** — Provides a list of formats: **Hex**, **Hex and ASCII**, and **Hex and Assembly**. Select a format type to specify the memory format.

Using the Memory Map Pane

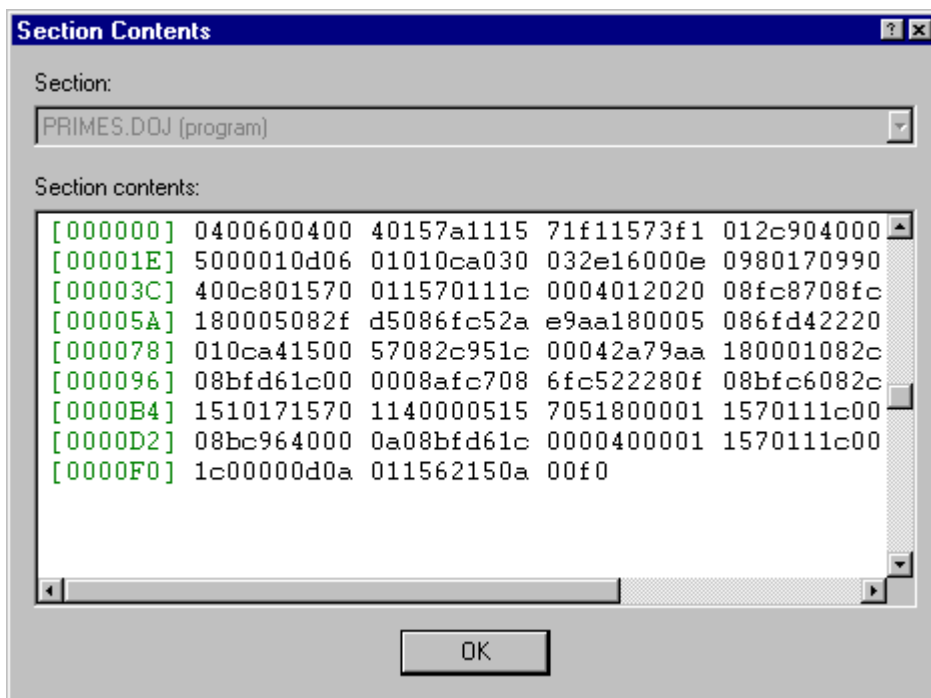


Figure 3-27. Output Section Contents in Hex Format

Figure 3-28, Figure 3-29, and Figure 3-30 illustrate memory data formats available for the selected output section.

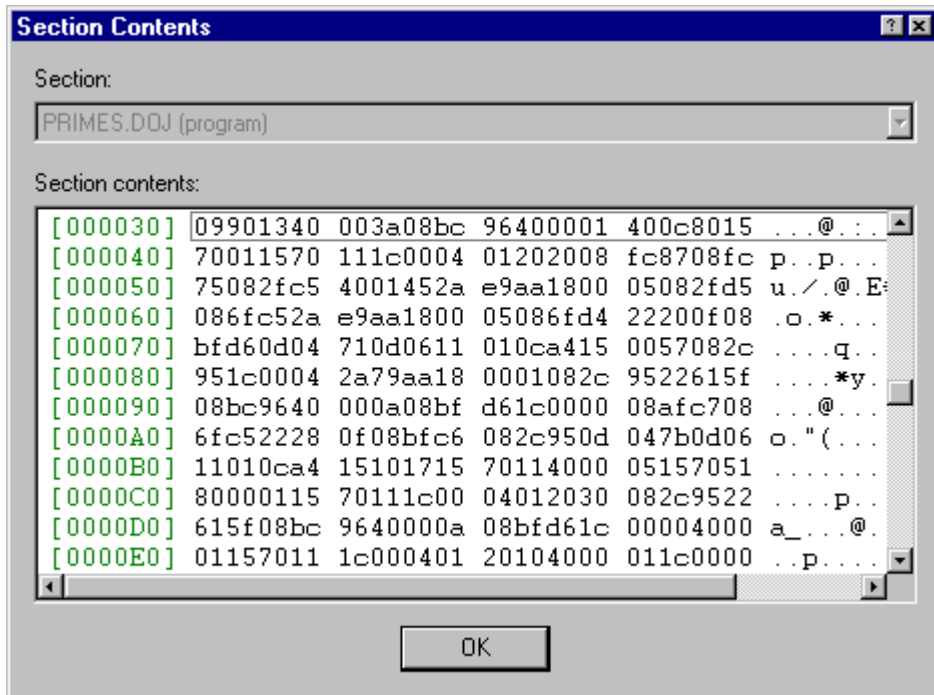


Figure 3-28. Output Section Contents in Hex and ASCII Format

Viewing Symbols

Symbols can be displayed per processor program (.DXE), per overlay (.OVL), or per input section. Initially, symbol data is in the same order that it appears in the linker's map output. Sort symbols by name, address, and so on by clicking the column headings.

Using the Memory Map Pane

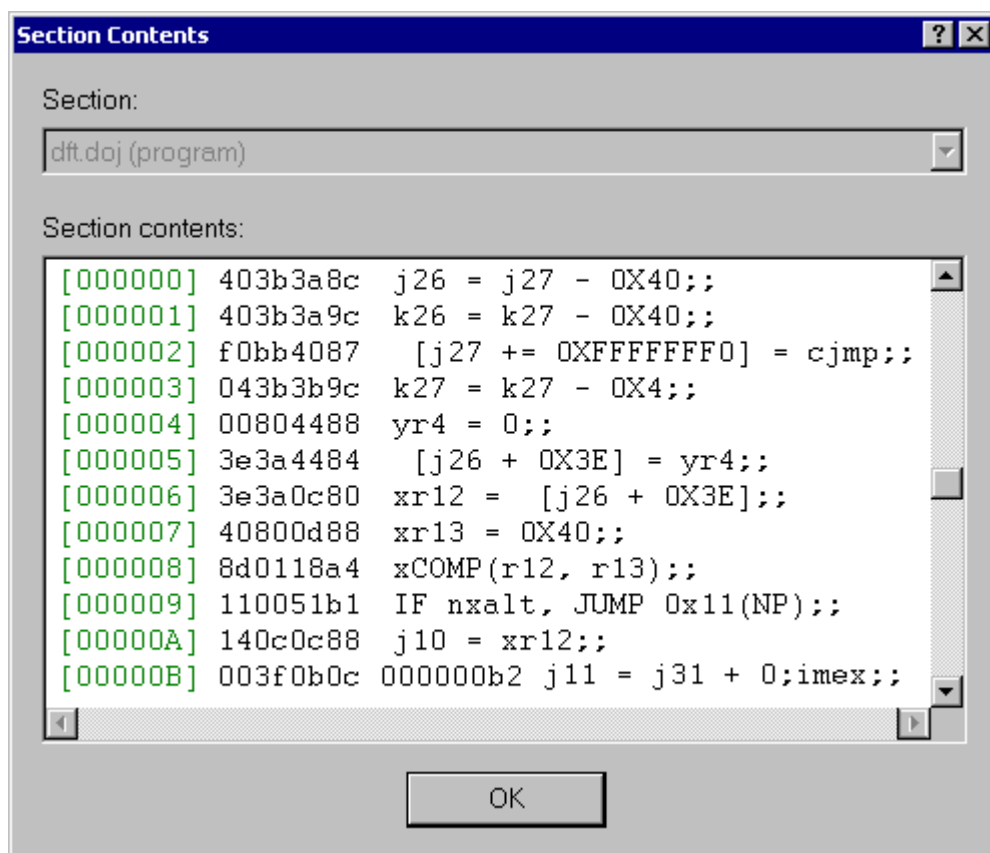


Figure 3-29. Output Section Contents in Hex and Assembly Format

To view symbols:

1. In the post-link view of the **Memory Map** pane, select the item (memory segment, output section, or input section) whose symbols you want to view.
2. Right-click and choose **View Symbols**.

The **View Symbols** dialog box displays the selected item's symbols. The symbol's address, size, binding, file name, and section appear beside the symbol's name.

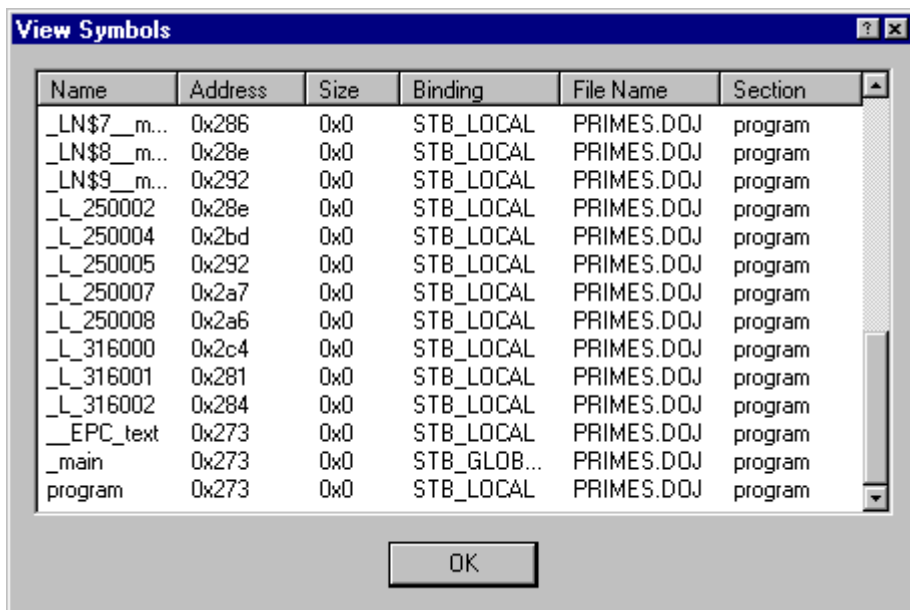



Figure 3-30. View Symbols Dialog Box

Managing Object Properties

You can display different properties for each type of object. Since different objects may share certain properties, their **Properties** dialog boxes share pages.

 The following procedures assume the Expert Linker window is open.

To display a **Properties** dialog box, right-click an object and choose **Properties**. You may choose these functions:

- [“Managing Global Properties” on page 3-41](#)
- [“Managing Processor Properties” on page 3-42](#)
- [“Managing PLIT Properties for Overlays” on page 3-44](#)
- [“Managing Elimination Properties” on page 3-45](#)
- [“Managing Symbols Properties” on page 3-47](#)
- [“Managing Memory Segment Properties” on page 3-51](#)
- [“Managing Output Section Properties” on page 3-53](#)
- [“Managing Packing Properties” on page 3-55](#)
- [“Managing Alignment and Fill Properties” on page 3-56](#)
- [“Managing Overlay Properties” on page 3-58](#)
- [“Managing Stack and Heap in DSP Memory” on page 3-60](#)

Managing Global Properties

Use the context menu to display Global Properties:

1. Right-click in a section pane of the Expert Linker window.
2. From the context menu, choose **Global Properties**.
The **Global Properties** dialog box appears.

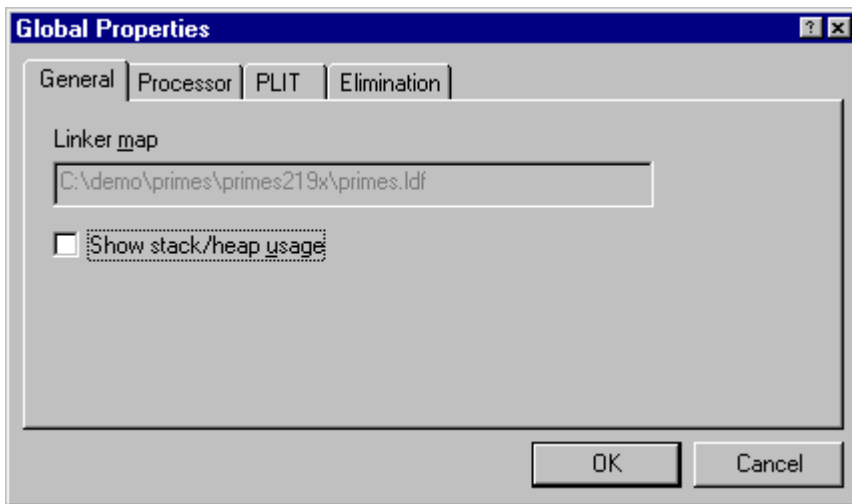


Figure 3-31. General Page of the Global Properties Dialog Box

The **Global Properties** dialog box provides these selections:

- **Linker map file** displays the map file generated after linking the project. This is a read-only field.
- If **Show stack/heap usage** is selected, after you run a project, Expert Linker shows how much of the stack and heap were used.

Managing Processor Properties

To specify processor properties:

1. In the **Memory Map** pane, right-click on a **Processor** tab and choose **Properties**.

The **Processor Properties** dialog box appears.

2. Click the **Properties** tab.

The **Processor** tab allows you to reconfigure the processor setup.

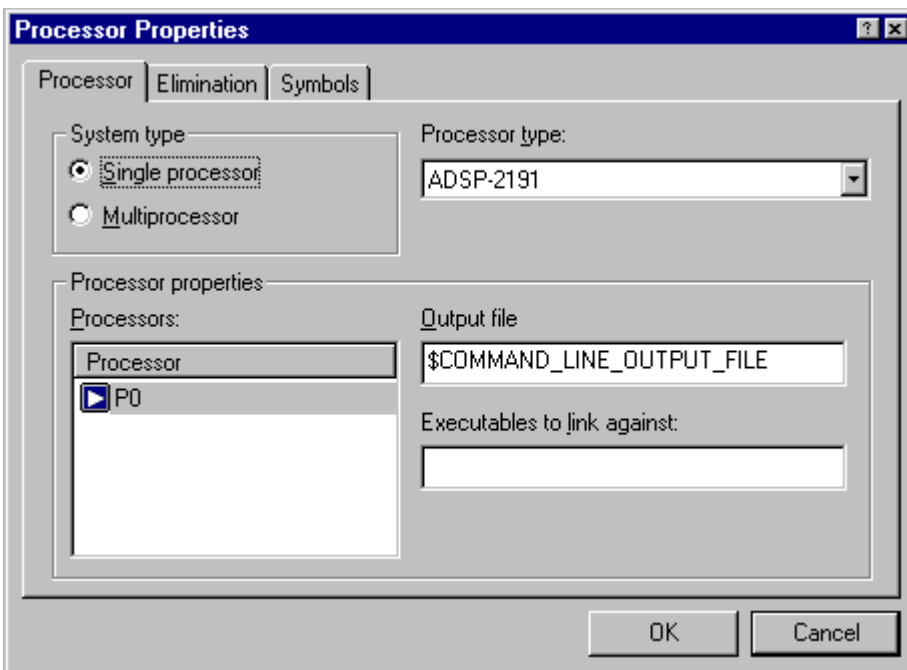


Figure 3-32. Processor Page of the Processor Properties Dialog Box


With a **Processor** tab in focus, you can:

- Specify **System Type** – Use the **Single processor** selection.
- Select a **Processor type** (such as ADSP-BF532).
- Specify an **Output file** name – The file name may include a relative path and/or LDF macro.
- Specify **Executables to link against** – Multiple files names are permitted, but must be separated with space characters. Only .SM, .DLB, and .DXE files are permitted. A file name may include a relative path and/or LDF macro.

Additionally, you can rename a processor by selecting the processor, right-clicking, and choosing **Rename Processor**, and typing a new name.

Managing PLIT Properties for Overlays

The **PLIT** tab allows you to view and edit the function template used in overlays. Assembly instructions observe the same syntax coloring as specified for editor windows.

 You can enter assembly code only. Comments are not allowed.

To view and edit **PLIT** information:

1. Right-click in the **Input Sections** pane.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. Click the **PLIT** tab.

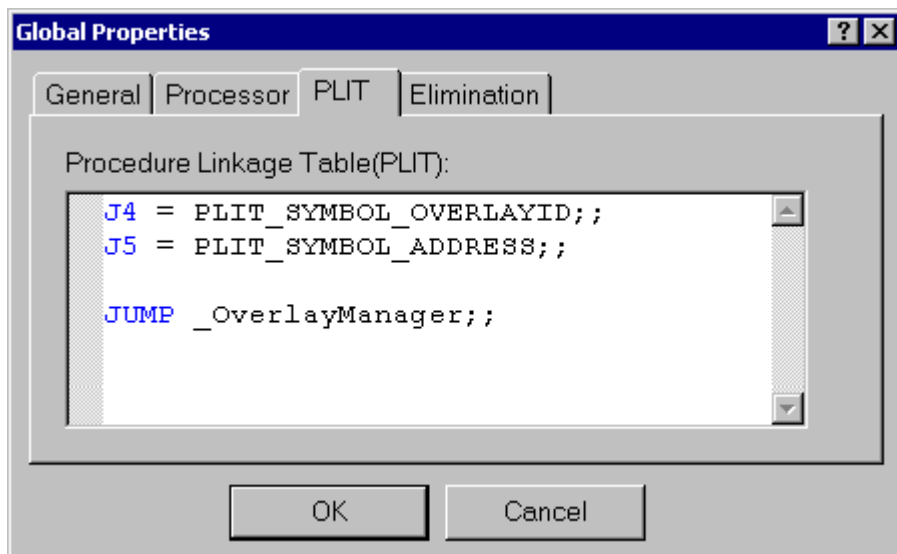


Figure 3-33. PLIT Page of the Global Properties Dialog Box

Managing Elimination Properties

You can eliminate unused code from the target .DxE file. Specify the input sections from which to eliminate code and the symbols you want to keep.

The **Elimination** tab allows you to perform elimination.

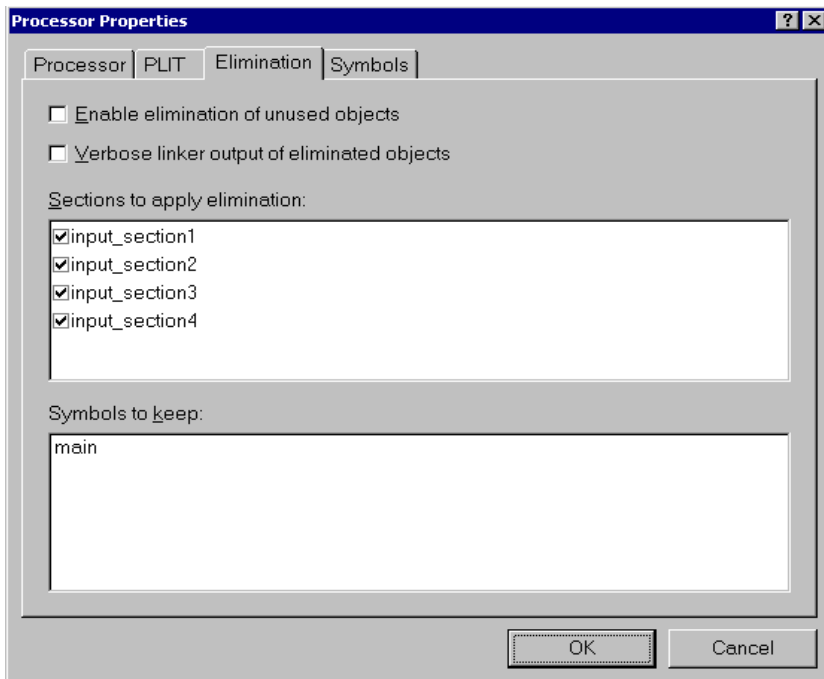


Figure 3-34. Processor Properties Dialog Box – Elimination Tab

Selecting the **Enable elimination of unused objects** option enables elimination. This check box is grayed out when elimination is enabled through the linker command line or when the .LDF file is read-only.

Managing Object Properties

When **Verbose linker output of eliminated objects** is selected, the eliminated objects are shown as linker output in the **Output** window's **Build** tab during linking. This check box is grayed out when the **Enable elimination of unused objects** check box is cleared. It is also grayed out when elimination is enabled through the linker command line or when the `.LDF` file is read-only.

The **Sections to apply elimination** box lists all input sections with a check box next to each section. Elimination applies to the sections that are selected. By default, all input sections are selected.

Symbols to keep is a list of symbols to be retained. The linker will not remove these symbols. If you right-click in this list box, a menu allows you to:

- Add a symbol by typing in the new symbol name in the edit box at the end of the list
- Remove the selected symbol

Managing Symbols Properties

You can view the list of symbols resolved by the linker. You can also add and remove symbols from the list of symbols kept by the linker. The symbols can be resolved to an absolute address or to a program file (.DxE). It is assumed that you have enabled the elimination of unused code.

To add or remove a symbol:

1. Right-click in the **Input Sections** pane of the Expert Linker window.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. Click the **Elimination** tab to add or remove a symbol.
4. Right-click in the **Symbols to keep** window.

Choose **Add Symbol**. In the ensuing dialog box, type new symbol names at the end of the existing list. To delete a symbol, select the symbol, right-click, and choose **Remove Symbol**.

To specify symbol resolution:

1. In the **Memory Map** pane, right-click a processor tab.
2. Choose **Properties**. The **Processor** page of the **Processor Properties** dialog box appears. The **Symbols** tab allows you to specify how symbols are to be resolved by the linker.

The symbols can be resolved to an absolute address or to a program file (.DxE). When you right-click in the **Symbols** field, a menu enables you to add or remove symbols.

Managing Object Properties

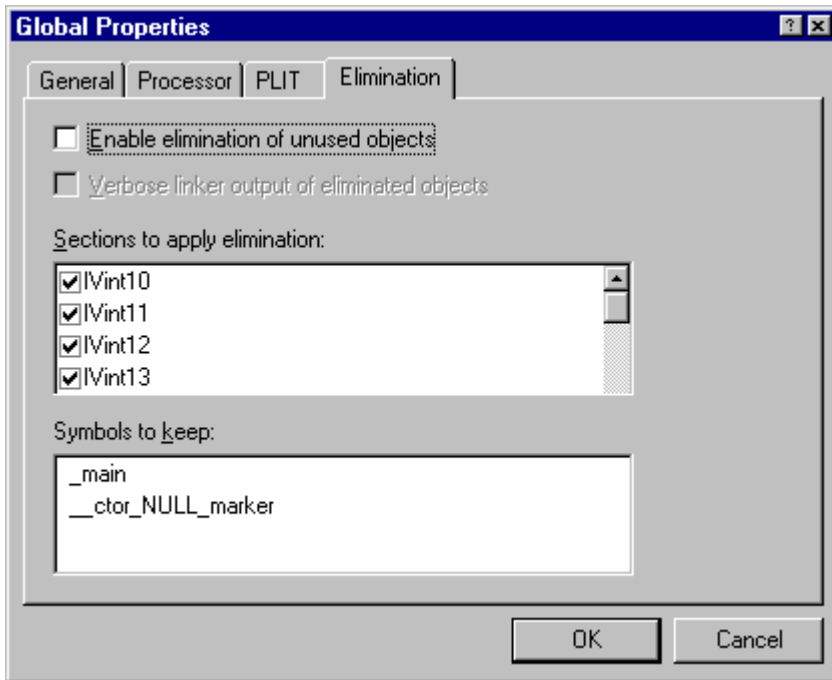


Figure 3-35. Elimination Page of the Global Properties Dialog Box

Choosing **Add Symbol** from the menu invokes the **Add Symbol to Resolve** dialog box allowing you to pick a symbol by either typing the name or browsing for a symbol. Using **Resolve with**, you can also decide whether to resolve the symbol from a known absolute address or file name (.DXE or .SM file).

The **Browse** button is grayed out when no symbol list is available; for example, if the project has not been linked. When this button is active, click it to display the **Browse Symbols** dialog box, which shows a list of all the symbols.

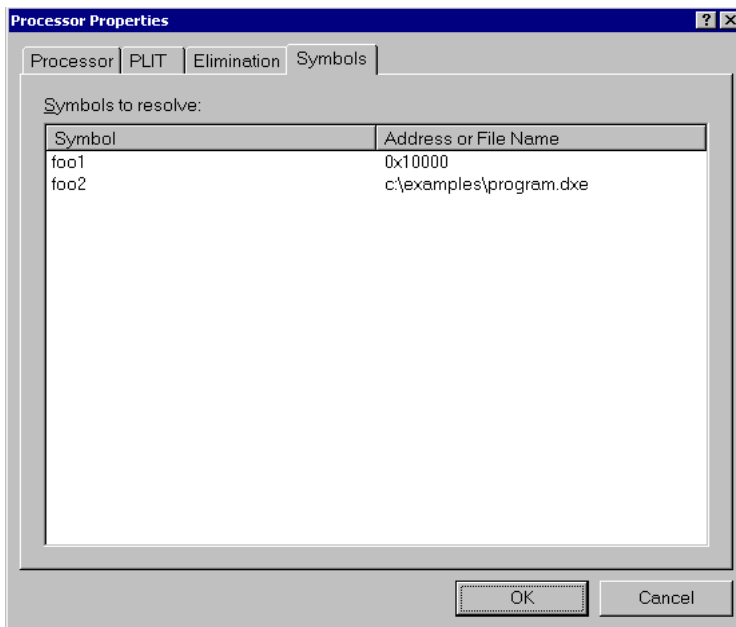


Figure 3-36. Processor Properties Dialog Box – Symbols Tab

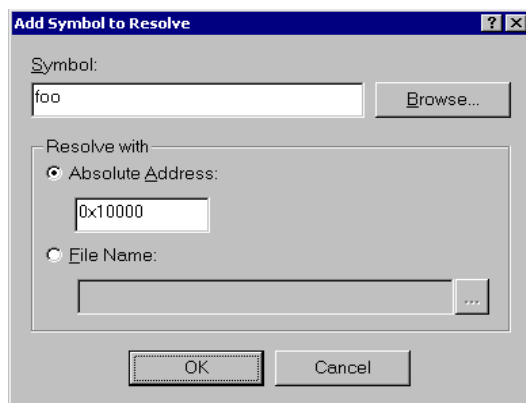


Figure 3-37. Add Symbol to Resolve Dialog Box

Managing Object Properties

Selecting a symbol from that list places it in the **Symbol** box of the **Edit Symbol to Resolve** dialog box.

To delete a symbol from the resolve list:

1. Click **Browse** to display the **Symbols to resolve** list in the **Symbols** pane ([Figure 3-36](#)).
2. Select the symbol you want to delete.
3. Right-click and choose **Remove Symbol**.

Managing Memory Segment Properties

You can specify/change the memory segment's name, start address, end address, size, width, memory space, memory type, and internal/external flag.



The BM memory space option applies only to ADSP-218x DSPs. The DATA64 memory space option applies only to ADSP-21160 and ADSP-21161N DSPs.

To display the **Memory Segment Properties** dialog box:

1. Right-click a memory segment (for example, `PROGRAM`) in the **Memory Map** pane.
2. Choose **Properties**. The selected segment properties are displayed.

Managing Object Properties

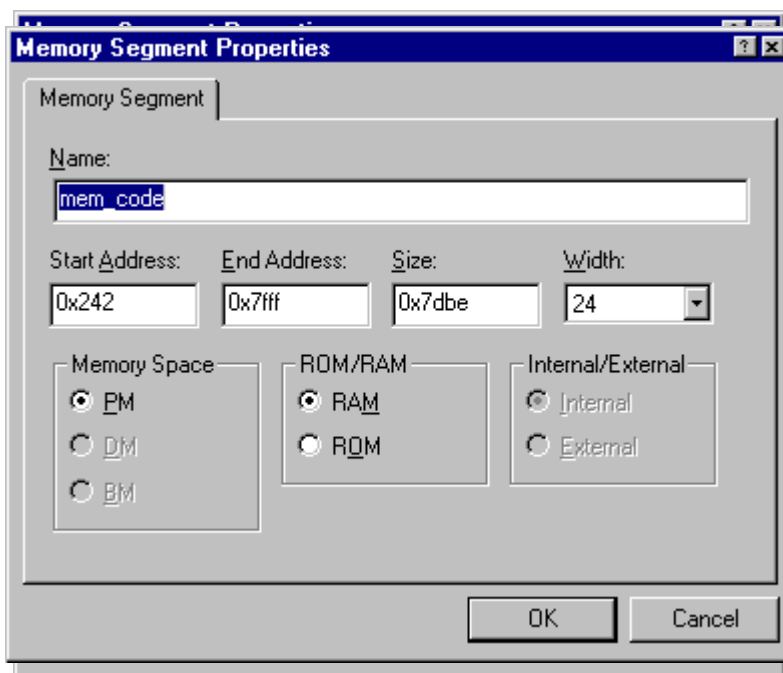


Figure 3-38. Memory Segment Properties Dialog Box

Managing Output Section Properties

The **Output Section** tab allows you to change the output section's name or to set the overflow. Overflow allows objects that do not fit in the current output section to spill over into the specified output section. By default, all objects that do not fit (except objects that are manually pinned to the current output section) overflow to the specified section.

To specify output section properties:

1. Right-click an output section in the **Memory Map** pane.
2. Choose **Properties**.



Figure 3-39. Output Section Properties Dialog Box – Output Section Tab

Managing Object Properties

The selections in the output section/segment list includes “None” (for no overflow) and all output sections. Objects can be pinned to an output section by right-clicking the object and then choosing **Pin to output section**.

You can:

- Type a name for the output section in **Name**.
- Select an output section into which the selected output section will overflow in **Overflow**. Or select **None** for no overflow. This setting appears in the **Placement** box.

Before linking the project, the **Placement** box indicates the output section’s address and size as “Not available”. After linking, the box displays the output section’s actual address and size.

Specify the **Packing** and **Alignment** (with **Fill Value**) properties as needed.

Managing Packing Properties

The **Packing** tab allows you to specify the packing format that the linker uses to place bytes into memory. The choices include **No packing** or **Custom** packing. You can view byte order, which defines the order that bytes will be placed into memory. With Blackfin processors, **No packing** is the only packing method available.

To specify packing properties:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties** and click the **Packing** tab.

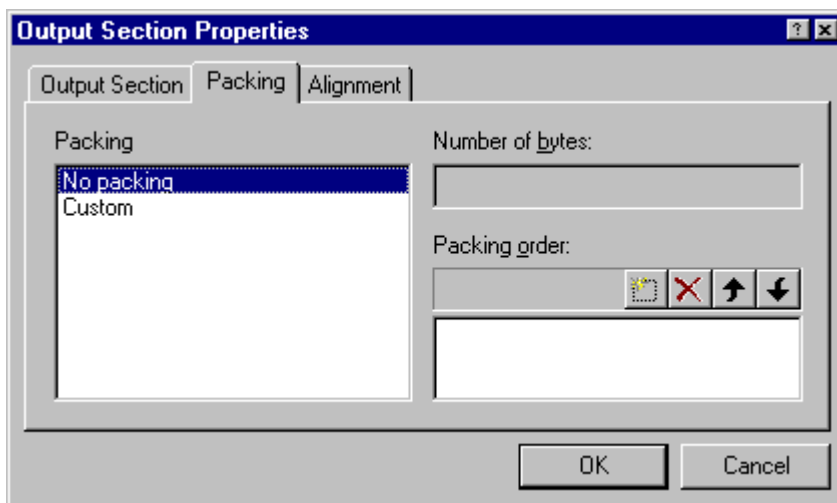


Figure 3-40. Memory Segment Properties Dialog Box – Packing Tab

Managing Alignment and Fill Properties

The **Alignment** tab allows you to set the alignment and fill values for the output section. When the output section is aligned on an address, the linker fills the gap with zeros (0), NOP instructions, or a specified value.

To specify alignment properties:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties**.
3. Click the **Alignment** tab.

No Alignment specifies no alignment.

If you select **Align each input section to the next address that is a multiple of**, select an integer value from the drop-down list to specify the output section alignment.

When the output section is aligned on an address, there is a gap that is filled by the linker. Based on the processor architecture, the Expert Linker determines the opcode for the NOP instruction.

The **Fill value** is either 0, a NOP instruction, or a user-specified value (a hexadecimal value entered in the entry box).

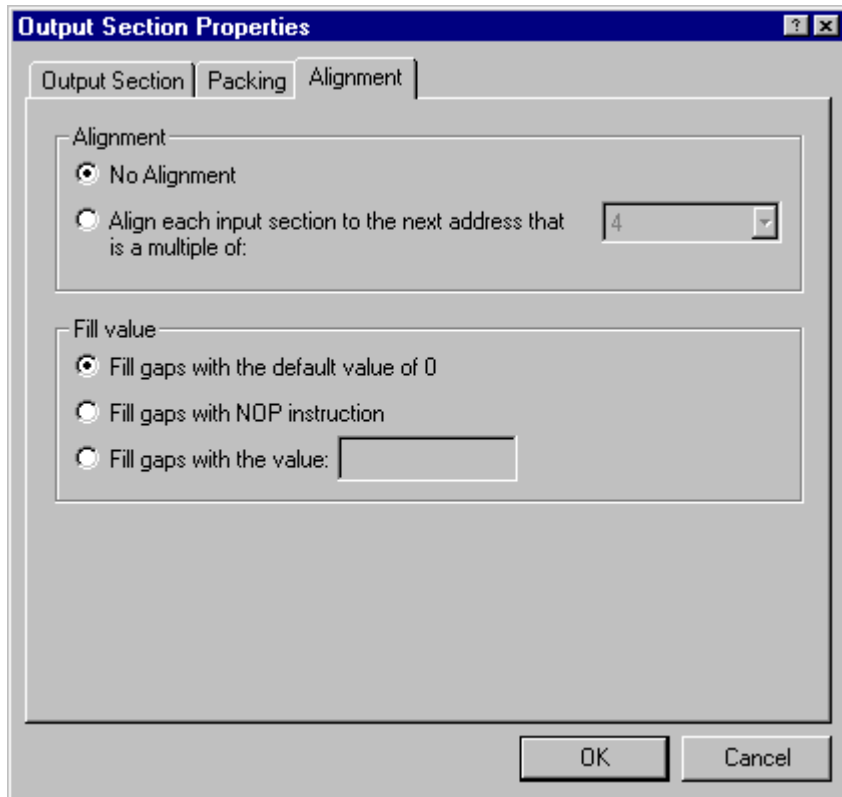


Figure 3-41. Output Section Properties – Alignment Tab

Managing Overlay Properties

The **Overlay** tab allows you to choose the output file for the overlay, its live memory, and its linking algorithm.

To specify overlay properties:

1. Right-click an overlay object in the **Memory Map** pane.
2. Choose **Properties**.

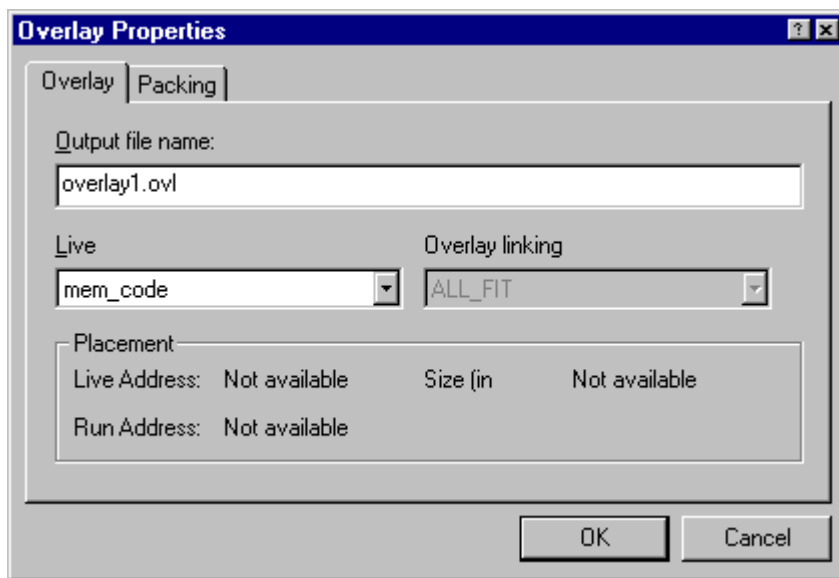


Figure 3-42. Overlay Properties Dialog Box – Overlay Tab

Live Memory contains a list of all output sections or memory segments with one output section. The live memory is where the overlay is stored before it is swapped into memory.

The **Overlay linking algorithm** box permits one overlay algorithm — `ALL_FIT`. Expert Linker does not allow you to change this setting. When using `ALL_FIT`, the linker tries to fit all of the mapped objects into one overlay.

The **Browse** button is available only if the overlay has already been built and the symbols are available. Clicking **Browse** opens the **Browse Symbols** dialog box.

You can choose the address for the symbol group or let the linker choose the address.

Managing Stack and Heap in DSP Memory

The Expert Linker show how much space is allocated for your program's heap and stack.

[Figure 3-43](#) shows stack/heap output sections in the **Memory Map** pane. Right-click on either of them to display its properties.

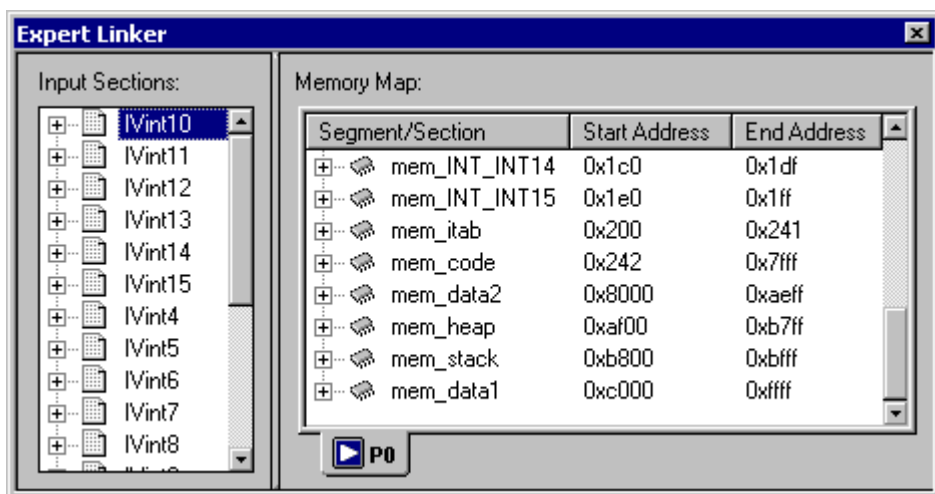


Figure 3-43. Memory Map Window With Stack/Heap Sections

Use the **Global Properties** dialog box to select **Show stack/heap usage**. This option graphically displays the stack/heap usage in memory ([Figure 3-44](#)).

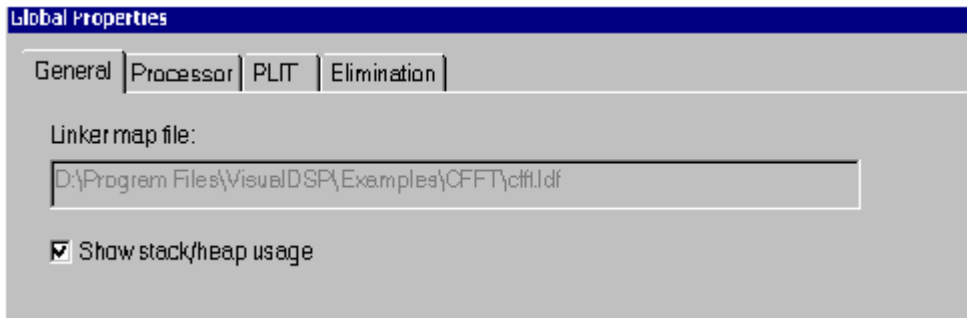


Figure 3-44. Global Properties – Selecting Stack/Heap Usage

The Expert Linker can:

- Locate stacks/heaps and fill them with a marker value.
This occurs after you load the program into a DSP target. The stacks and heaps are located by their output section names, which may vary across processor families.
- Search the heap and stack for the highest memory locations written to by the DSP program.

This occurs when the target halts after running the program. Assume this as the start of the unused portion of the stack or heap. The Expert Linker updates the memory map to show how much of the stack and heap are unused.

Use this information to adjust the size of your stack and heap. This information helps make better use of the DSP memory, so the stack and heap segments do not use too much memory.

Use the graphical view (**View Mode -> Graphical Memory Map**) to display stack/heap memory map blocks. [Figure 3-45](#) shows a possible memory map after running a Blackfin processor project program.

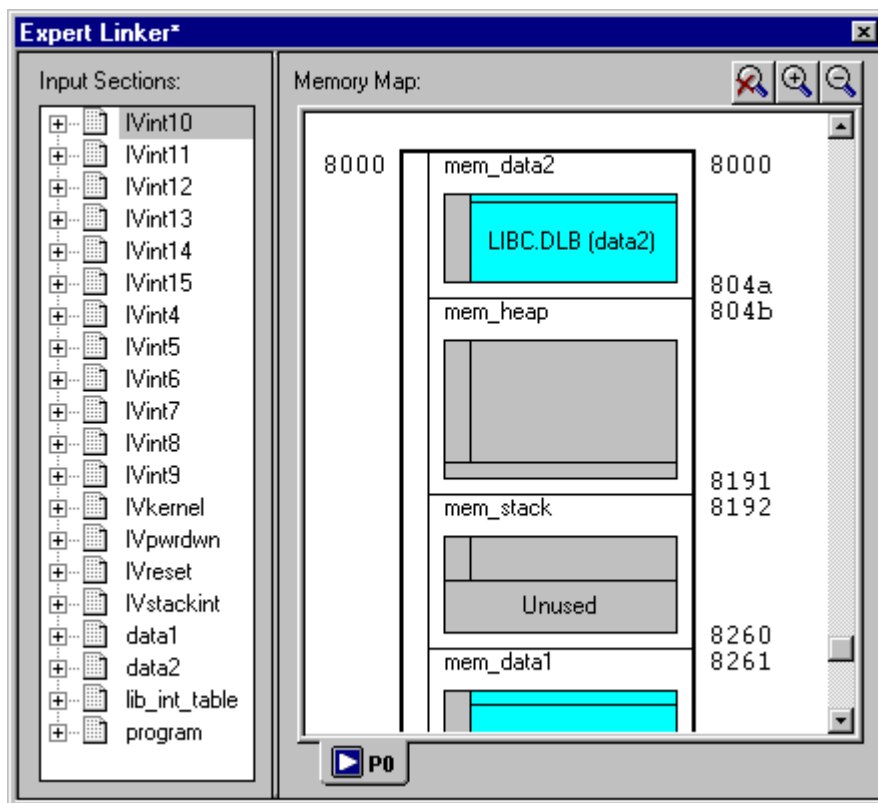


Figure 3-45. Graphical Memory Map Showing Stack/Heap Usage

4 ARCHIVER

The VisualDSP++ archiver, `elfar.exe`, combines object files (`.OBJ`) into library files, which serve as reusable resources for code development. The VisualDSP++ linker rapidly searches library files for routines (library members) referred to by other object files and links these routines into your executable program.

This chapter provides the following archiver information:

- “[Archiver Guide](#)” on page 4-2 introduces the archiver’s functions
- “[Archiver Command-Line Reference](#)” on page 4-7 reference information on archiver operations

The archiver can be run from a command line, or you can produce an archive file as the output of a VisualDSP++ project.

Archiver Guide

The `elfar.exe` utility combines and indexes object files (or any other files), producing a searchable library file. It performs the following operations, as directed by options on the `elfar` command line:

- Creates a library file from a list of object files
- Appends one or more object files to an existing library file
- Deletes file(s) from a library file
- Extracts file(s) from a library file
- Prints the contents of a specified object file of an existing library file to `stdout`
- Replaces file(s) in an existing library file
- Encrypts symbol(s) in an existing library file

The archiver can run only one of these operations at a time. However, for commands that take a list of file names as arguments, the archiver can input a text file that contains the names of object files (separated by white space). The operation makes long lists easily manageable.

The archiver, which is sometimes called a librarian, is general-purpose. It can combine and extract arbitrary files. This manual refers to DSP object files (`.DOJ`) because they are relevant to DSP code development.

Creating a Library From VisualDSP++

Within the VisualDSP++ development environment, you can choose to create a library file as your project's output. To do so, specify **DSP library file** as the target type on the **Project** page of the **Project Options** dialog box.

VisualDSP++ writes its output to `<projectname>.DLB`. To modify or list the contents of a library file, or perform any other operations on it, run the archiver from the `elfar` command line.

File Name Conventions

To maintain code consistency, use the conventions in [Table 4-1](#).



When creating a library, VisualDSP++ writes `<projectname>.DLB`.

Table 4-1. File Name Extensions

Extension	File Description
.DLB	Library file
.DOJ	Object file. Input to archiver.
.TXT	Text file used as input with the <code>-i</code> switch

Making Archived Functions Usable

In order to use the archiver effectively, you must know how to write archive files which make your DSP functions available to your code (via the linker), and how to write code that accesses these archives.

Archive usage consists of two tasks:

- Writing *archive routines*, functions that can be called from other programs
- Accessing archive routines from your code

Writing Archive Routines: Creating Entry Points

An archive routine is a routine (or function) in code that can be accessed by other programs. Each routine must have a globally visible start label (*entry point*). Code that accesses that routine must declare the entry point's name as an external variable in the calling code.

To create entry points

1. Declare the start label of each routine as a global symbol with the assembler's `.GLOBAL` directive. This defines the entry point.

The following code fragment has two entry points, `dIriir` and `FAE`.

```
...  
.global dIriir;  
.section data1;  
.byte2 FAE = 0x1234,0x4321;  
  
.section program;  
.global FAE;  
dIriir: R0=N-2;  
  
P2 = FAE;
```

2. Assemble and archive the code containing the routines. Use either of the following methods.
 - Direct VisualDSP++ to produce a library (see [“Creating a Library From VisualDSP++”](#)). When building the project, the object code containing the entry points is packaged in `<projectname>.DLB`. You can extract an object file (`.DOJ`), for example, to incorporate it in another project.
 - When creating executable or unlinked object files from VisualDSP++, archive them afterwards from the `elfar` command line.

Accessing Archived Functions From Your Code

Programs that call a library routine must use the assembler's `.EXTERN` directive to specify the routine's start label as an external label. When linking the program, specify one or more library files (`.DLB`) to the linker, along with the names of the object files (`.DOJ`) to link. The linker then searches the library files to resolve symbols and links the appropriate routines into the executable file.

Any file containing a label referenced by your program is linked into the executable output file. Linking libraries is faster than using individual object files, and there is no need to enter all the file names, just the library name.

Archiver Guide

In the following example, the archiver creates the `filter.dlb` library, containing the object files: `taps.doj`, `coeffs.doj`, and `go_input.doj`.

```
elfar -c filter.dlb taps.doj coeffs.doj go_input.doj
```

If you then run the linker with the following command line, the linker links the object files `main.doj` and `sum.doj`, uses the default Linker Description File (for example, `ADSP-BF535.ldf`), and creates the executable file (`main.dxe`).

```
linker -DADSP-BF535 main.doj sum.doj filter.dlb -o main.dxe
```

Assuming that one or more library routines from `filter.dlb` are called from one or more of the object files, the linker searches the library, extracts the required routines, and links the routines into the executable.

Archiver File Searches

File searches are important in the archiver's process. The archiver supports relative and absolute directory names, default directories, and user-specified directories for file search paths. File searches include:

- *Specified path* – If you include relative or absolute path information in a file name, the archiver searches only in that location for the file.
- *Default directory* – If you do not include path information in the file name, the archiver searches for the file in the current working directory.

Archiver Command-Line Reference

The archiver processes object files into a library file with a `.DLB` extension, which is the default extension for library files. The archiver can also append, delete, extract, or replace member files in a library, as well as list them to `stdout`. This section provides reference information on the archiver command line and linking.

elfar Command Syntax

Use the following syntax to run `elfar` from the command line.

```
elfar [-a|c|d|e|p|r] [-v] [-M] [-MM] [-i filename]
      lib_file obj_file [obj_file ...]
```

[Table 4-2](#) describes each switch.

Symbol Encryption

When using symbol encryption, use the following syntax.

```
elfar -s [-v] out-archive in-archive exclude-file type-letter
```

Refer to [“Archiver Symbol Name Encryption Reference” on page 4-11](#) for details.

Archiver Command-Line Reference

Example elfar Command

```
elfar -v -c my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

This command line runs the archiver as follows:

-v – Outputs status information

-c my_lib.dlb – Creates a library file named my_lib.dlb

fft.doj sin.doj cos.doj tan.doj – Places these object files in the library file

[Table 4-1 on page 4-3](#) lists typical file types, file names, and extensions.

Parameters and Switches

[Table 4-2](#) describes each archiver part of the command. Switches must appear before the name of the archive file.

Table 4-2. elfar.exe Command-Line Items

Item	Description
<i>lib_file</i>	Specifies the library that the archiver modifies. This parameter appears after the switch.
<i>obj_file</i>	Identifies one or more object files that the archiver uses when modifying the library. This parameter must appear after <i>lib_file</i> . Use the -i switch to input a list of object files.
-a	Appends one or more object files to the end of the specified library file.
-c	Creates a new <i>lib_file</i> containing the listed <i>obj_file(s)</i> .
-d	Removes the listed <i>obj_file(s)</i> from the specified <i>lib_file</i> .
-e	Extracts the specified file(s) from the library.
-i <i>filename</i>	Uses <i>filename</i> , a list of object files, as input. This file lists <i>obj_file(s)</i> to add or modify in the specified <i>lib_file</i> (.DLB).
-M	(available only with the -c switch) Prints dependencies.

Table 4-2. elfar.exe Command-Line Items (Cont'd)

Item	Description
-MM	(available only with the -c switch) Prints dependencies and creates the library.
-p	Prints a list of the <i>obj_file(s)</i> (.DOJ) in the selected <i>lib_file</i> (.DLB) to standard output.
-r	Replaces the specified object file in the specified library file. The object file in the library and the replacement object file must have identical names.
-s	Specifies symbol name encryption. Refer to “Archiver Symbol Name Encryption Reference” on page 4-11 .
-v	Verbose. Outputs status information as the archiver processes files.

Constraints

The `elfar` command is subject to the following constraints:

- Select one action switch (a, c, d, e, p, r, s, M, or MM) only in a single command.
- Do not place the verbose operation switch, -v, in a position where it can be mistaken for an object file. It may not follow the *lib_file* on an append or create operation.
- The file include switch, -i, must immediately precede the name of the file to be included.

The archiver's -i switch lets you input a list of members from a text file, instead of listing each member on the command line.


- Use the library file name first, following the switches. -i and -v are not operational switches, and can appear later.
- When using the archiver's -p switch, there is no need to identify members on the command line.

Archiver Command-Line Reference

- Enclose file names containing white space or colons within straight quotes.
- Append the appropriate file extension to each file. The archiver assumes nothing, and will not do it for you.
- Wildcards are not permitted. To perform an archive operation on a list of member files, write the list in a text file and use the text file as input to the command line with the `-i` switch.
- *obj_file* — The name of an object file (`.OBJ`) to be added, removed, or replaced in the *lib_file*.
- The archiver's command line is *not* case sensitive.

Archiver Symbol Name Encryption Reference

Symbol name encryption protects intellectual property contained in an archive library (.DLB) that might be revealed by the use of meaningful symbol names. Code and test a library with meaningful symbol names, and then use archive library encryption on the fully tested library to disguise the names.

 Source file names in the symbol tables of object files in the archive are not encrypted. The encryption algorithm is not reversible. Also, encryption does not guarantee a given symbol will be encrypted the same way when different libraries, or different builds of the same library, are encrypted.

Command Syntax

The following command line encrypts symbols in an existing archive file.

```
elfar -s [-v] out-archive in-archive exclude-file type-letter
```

where:

-s – Selects the encryption operation.

-v – Selects verbose mode, which provides statistics on the symbols that were encrypted.

out-archive – Specifies the name of the library file (.DLB) to be produced by the encryption process

in-archive – Specifies the name of the archive file (.DLB) to be encrypted. This file is not altered by the encryption process, unless in-archive is the same as out-archive.

Archiver Command-Line Reference

`exclude-file` – Specifies the name of a text file containing a list of symbols not to be encrypted. The symbols are listed one or more to a line, separated by white space.

`type-letter` – The initial letter of `type-letter` provides the initial letter of all encrypted symbols.

Constraints

All local symbols can be encrypted, unless they are correlated (see below) with a symbol having external binding that should not be encrypted. Symbols with external binding can be encrypted when they are used only within the library in which they are defined. Symbols with external binding that are not defined in the library (or are defined in the library and referred to outside of the library) should not be encrypted. Symbols that should not be encrypted must be placed in a text file, and the name of that file given as the `exclude-file` command line argument.

Some symbol names have a prefix or suffix that has special meaning. The debugger does not show a symbol starting with “.” (period), and a symbol starting with “.” and ending with “.end” is correlated with another symbol. For example, “.bar” will not be shown by the debugger, and “._foo.end” is correlated with the symbol “_foo” appearing in the same object file. The encryption process encrypts only the part of the symbol after any initial “.”, and before any final “.end”. This part is called the root of the symbol name. Since only the root is encrypted, a name with a prefix or suffix having special meaning retains that special meaning after encryption.

The encryption process ensures that a symbol with external binding will be encrypted the same way in all object files contained in the library. This process also ensures that correlated symbols (see explanation above) within an object file will be encrypted the same way, so they remain correlated.

The names listed in the `exclude-file` are interpreted as root names. Thus, “_foo” in the `exclude-file` prevents the encryption of the symbol names “_foo”, “._foo”, “_foo.end”, and “._foo.end”.

The `type-letter` argument, which provides the first letter of the encrypted part of a symbol name, ensures that the encrypted names in different archive libraries can be made distinct. If different libraries are encrypted with the same `type-letter` argument, there is a possibility that unrelated external symbols of the same length will be encrypted identically.

5 LOADER

This chapter explains how to convert an executable file (.DXX) into a boot-loadable file (.LDR) for the Blackfin processors using the loader utility (`elfloader.exe`). Once an application program is fully debugged, it is ready to be converted into a boot-loadable file. This file is then programmed/burned into an external memory device within your target system. Upon RESET of the system, the Blackfin processor boots the loadable file from the external memory and begins execution of the application program.

[Figure 5-1](#) shows the block diagram of the Blackfin processor booting process.

This chapter contains:

- [“Boot Sequence” on page 5-3](#)
Provides information on booting and kernel use
- [“Creating an .LDR File Using the Elfloader” on page 5-15](#)
Provides reference information on loader command-line switches and conventions
- [“File Formats” on page 5-30](#)
Provides reference information on loader configurations, loader bootstreams and output file formats

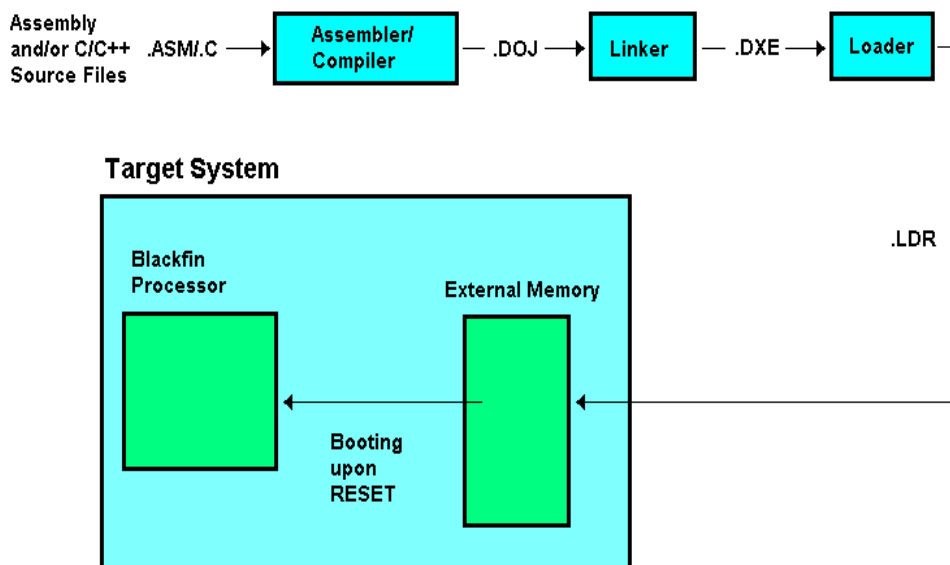


Figure 5-1. Booting Process

Boot Sequence

At powerup, after the reset, the processor transitions into the boot mode sequence configured by the `BMODE` pins. Blackfin processors can be booted from an 8-bit/16-bit FLASH/PROM memory or a serial ROM (8- or 16-bit addressable). There is also a No-Boot (Bypass mode) option, in which case, execution occurs from 16-bit external memory.

This section provides:

- [“ADSP-BF535 Processor Booting”](#)
- [“ADSP-BF531, ADSP-BF532, and ADSP-BF533 Processor Booting” on page 5-12](#)



See appropriate data sheets for more information. Also refer to the *Hardware Reference Manual* of the appropriate processor for more information on system configuration, registers, peripherals, etc.

ADSP-BF535 Processor Booting

[Table 5-1](#) shows booting modes for the ADSP-BF535 processor and their execution start address.

Table 5-1. ADSP-BF535 Processor Reset Vector Addresses and Boot Mode Selections

Boot Source	BMODE[2:0]	Execution Start Address
Execute from 16-bit external memory (Async Bank 0); No-Boot Mode (Bypass On-Chip Boot ROM)	000	0x2000 0000
Boot from 8-bit/16-bit FLASH memory	001	0xF000 0000
Boot from 8-bit address SPI0 serial EEROM	010	0xF000 0000
Boot from 16-bit address SPI0 serial EEROM	011	0xF000 0000
Reserved	100 – 111	N/A

Boot Sequence

Upon reset, the ADSP-BF535 processor will either jump to the On-Chip Boot ROM (if `BMODE = 001, 010, 011`) or jump to external 16-bit memory for execution (if `BMODE = 000`). The On-Chip Boot ROM for the ADSP-BF535 processor does the following (see [Figure 5-2](#)).

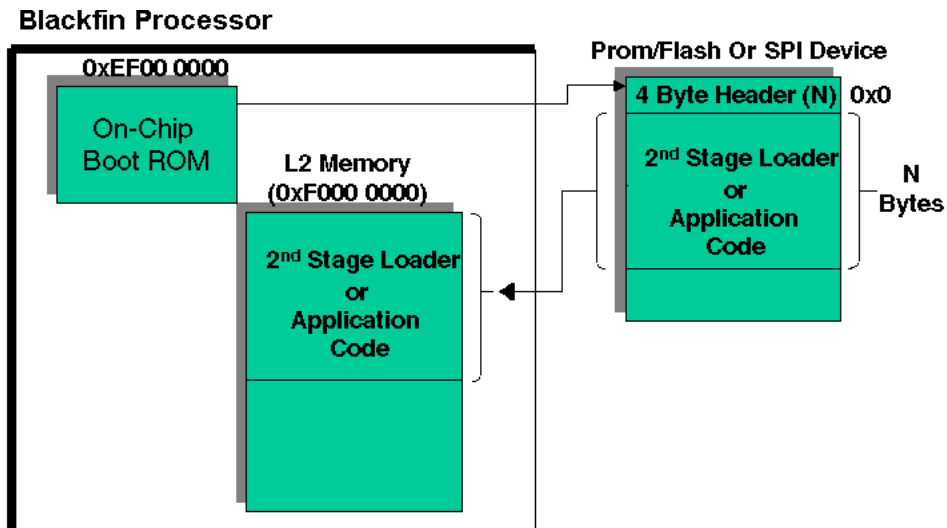


Figure 5-2. Booting Process for ADSP-BF535 Processors

1. Sets up Supervisor Mode by exiting the `RESET` interrupt service routine and jumping into the lowest priority interrupt (`IVG15`).
2. Checks whether the `RESET` was a software reset and if so, whether the user wants to skip the whole boot sequence and jump to the start of L2 memory (`0xF000 0000`) for execution. The On-Chip Boot ROM does this by checking bit 4 of the Reset Configuration Register (`SYSCR`). If bit 4 is not set, the On-Chip Boot ROM performs the full boot sequence. If bit 4 is set, the On-Chip Boot ROM bypasses the full boot sequence and jumps to `0xF000 0000`. See [Figure 5-3](#) as an example of `SYSCR` register bit descriptions.

System Reset Configuration Register (SYSCR)

X - state is initialized from mode pins during hardware reset

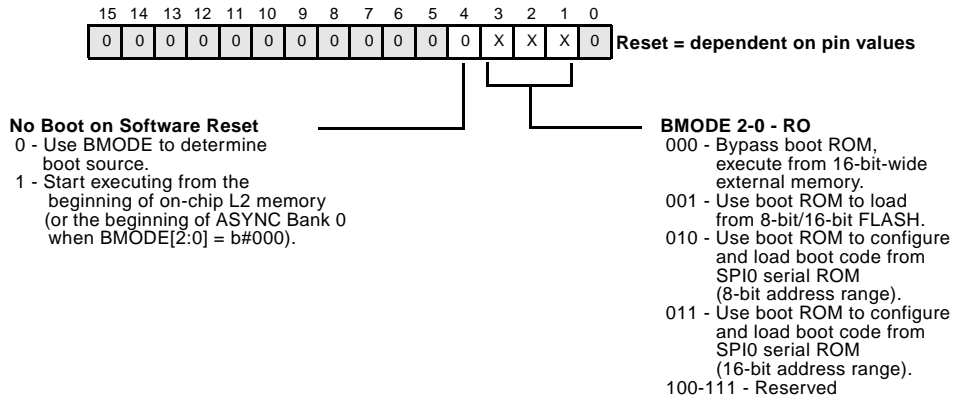


Figure 5-3. ADSP-BF535 System Reset Configuration Register (SYSCR)

3. Finally if bit 4 of the SYSCR register is not set, the On-Chip Boot ROM performs the full boot sequence. The full boot sequence includes:

- Checking the boot source (either FLASH/PROM or SPI memory) by reading BMODE[2:0] from the SYSCR register.
- Reading first four bytes from location 0x0 of the external memory device. These four bytes contain the byte count (N) of how many bytes to boot in.
- Booting in N bytes into internal L2 memory starting at location 0xF000 0000.
- Jumping to the start of L2 memory for execution.

The On-Chip Boot ROM boots in N bytes from external memory. These N bytes can define the size of the actual application code or a second-stage loader (boot kernel) that will boot in the application code. A second-stage loader has to be used in applications where there are multiple segments

Boot Sequence

residing in L2 memory, or when there are sections residing in L1 memory and/or SDRAM. In addition, a second-stage loader must be used to change the *waitstates* or *hold time cycles* for a FLASH/PROM booting or to change the *baudrate* for a SPI boot (see [“Loader Command-Line Switches” on page 5-18](#) for more information on these features).

The only situation where a second-stage loader is not necessary is if the application code contains only one section starting at the beginning of L2 memory (0xF000 0000).

When a boot kernel (second-stage) loader is used for booting, the following sequence takes place:

1. Upon RESET, the On-Chip Boot ROM downloads N bytes (the second-stage loader) from external memory to address 0xF000 0000 in L2 memory ([Figure 5-4](#)).

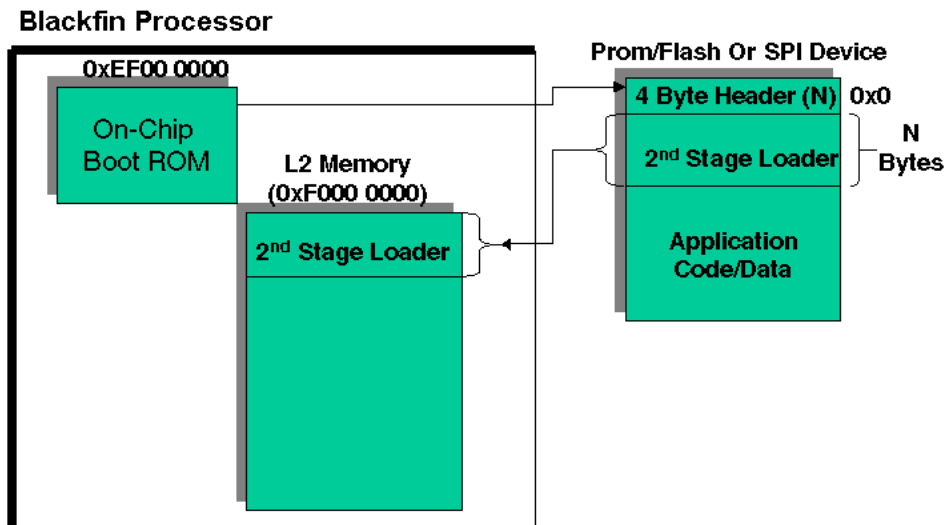


Figure 5-4. Loading the Second-Stage Loader from Boot ROM

2. The second-stage loader copies itself to the bottom of L2 memory (Figure 5-5).

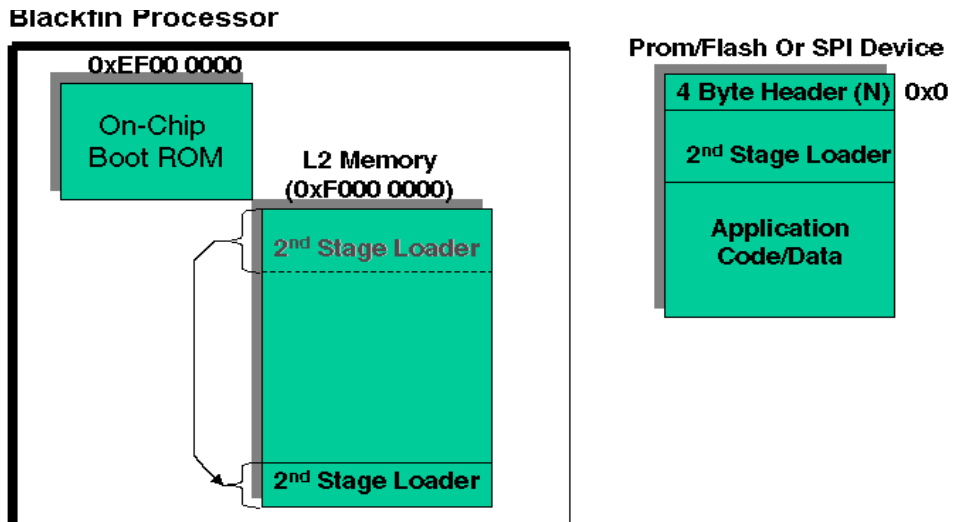


Figure 5-5. Copying Boot Kernel Loader to the Bottom of L2 Memory

3. The second-stage loader boots in the application code/data into the various memories of the Blackfin processor (Figure 5-6).
4. Finally, after booting, the second-stage loader jumps to the start of L2 memory (0xF000 0000) for application code execution (Figure 5-7).

Boot Sequence

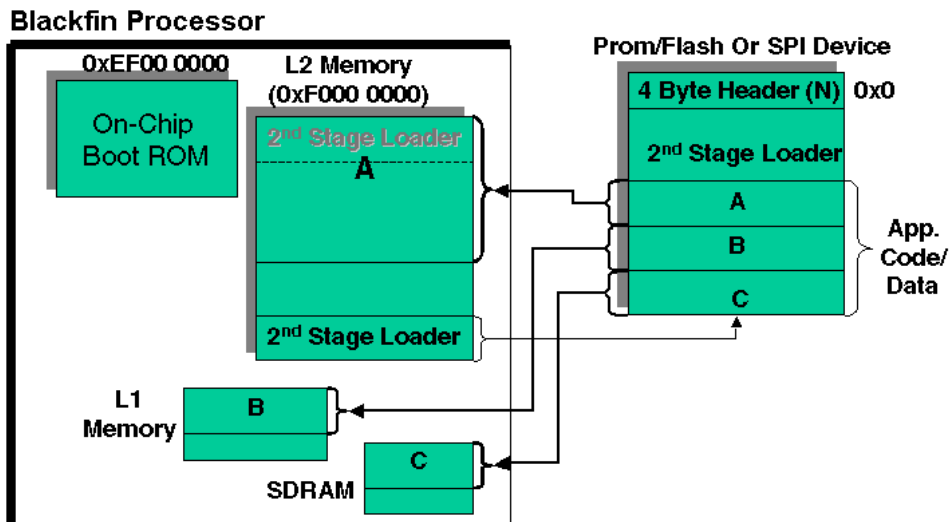


Figure 5-6. Booting Application Code into Blackfin Processor Memories

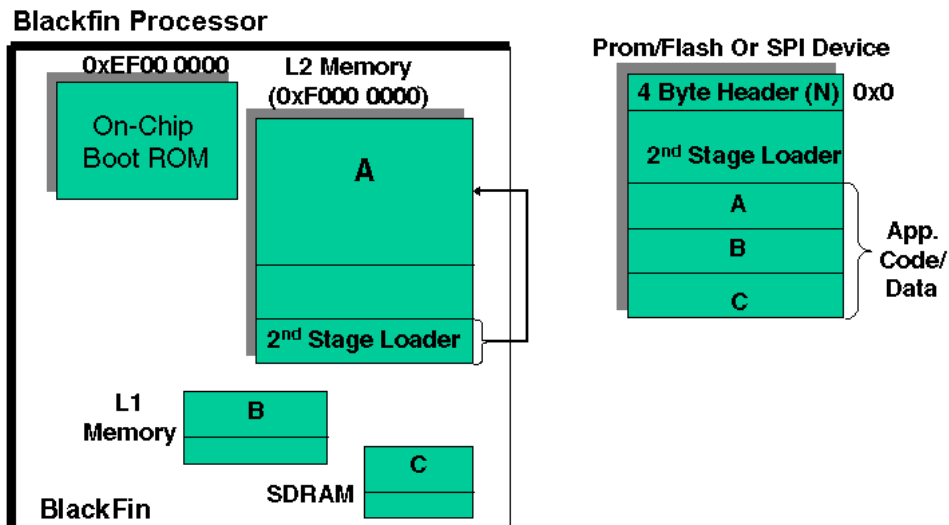


Figure 5-7. Boot Kernel Loader Jumps to the Start of L2 Memory

Second-stage loaders are available in VisualDSP++ 3.0 and higher. They allow booting to:

- L2 Memory (0xF000 0000)
- L1 Memory
 - ☐ Data Bank A SRAM (0xFF80 0000)
 - ☐ Data Bank B SRAM (0xFF90 0000)
 - ☐ Instruction SRAM (0xFFA0 0000)
 - ☐ Scratchpad SRAM (0xFFB0 0000)
- SDRAM
 - ☐ Bank 0 (0x0000 0000)
 - ☐ Bank 1 (0x0800 0000)
 - ☐ Bank 2 (0x1000 0000)
 - ☐ Bank 3 (0x1800 0000)



SDRAM must be initialized by user code before instructions/data are loaded into it.

Restrictions Using the Second-Stage Loader

When using the second-stage loader:

- The bottom of L2 memory has to be reserved during booting. These locations can be reallocated during runtime. The following locations pertain to the current second-stage loaders:
 - ☐ For 8- and 16-bit PROM/FLASH booting, reserve 0xF003 FE00 - 0xF003 FFFF (last 512 bytes)
 - ☐ For 8- and 16-bit addressable SPI booting, reserve 0xF003 FD00 - 0xF003 FFFF (last 768 bytes)
- If segments reside in SDRAM memory, you need to configure the SDRAM registers accordingly in the second-stage loader kernels before booting.
 - ☐ Modify section of code called “SDRAM setup” in the second-stage loader and rebuild the second-stage loader
- Any segments residing in L1 instruction memory (0xFFA00000 - 0xFFA03FFF) have to be 8-byte aligned.
 - ☐ Declare segments, within the .LDF file, that reside in L1 instruction memory at starting locations that are 8-byte aligned (i.e., 0xFFA00000, 0xFFA00008, 0xFFA00010,)
 - ☐ Or use the `.align 8;` qualifiers in the application code



The reason for this restriction is:

- Core Writes into L1 instruction memory are not allowed, and
- DMAing from an 8-bit external memory is not possible since the minimum width of the EBIU is 16 bits

Loading bytes into L1 instruction memory is done by using the instruction test command and data registers (described in the *Memory* chapter of the appropriate *Hardware Reference* manual). These registers transfer 8-byte sections of data from external memory to internal L1 instruction memory.

ADSP-BF531, ADSP-BF532, and ADSP-BF533 Processor Booting

Table 5-2 shows booting modes for the ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors and their execution start address.

Table 5-2. ADSP-BF531, ADSP-BF532, and ADSP-BF532 Processor Reset Vector Addresses and Boot Mode Selections

Boot Source	BMODE[1:0]	Execution Start Address for ADSP-BF531 / ADSP-BF532 Processors	Execution Start Address ADSP-BF533 Processor
Execute from 16-bit external memory (Async Bank 0); No-Boot Mode (Bypass On-Chip Boot ROM)	00	0x2000 0000	0x2000 0000
Boot from 8-bit/16-bit FLASH memory	01	0xFFA0 8000	0xFFA0 0000
Boot from 8-bit address SPI0 serial EEROM	10	0xFFA0 8000	0xFFA0 0000
Boot from 16-bit address SPI0 serial EEROM	11	0xFFA0 8000	0xFFA0 0000

Upon reset, a ADSP-BF531/32/33 processor will either jump to the On-Chip Boot ROM (if BMODE = 01, 10, 11) or jump to external 16-bit memory for execution (if BMODE = 00). The On-Chip Boot ROM for ADSP-BF531/32/33 processors does the following:

1. Sets up Supervisor Mode by exiting the RESET interrupt service routine and jumping into the lowest priority interrupt (IVG15).
2. Checks whether the RESET was a software reset and if so, whether the user wants to skip the whole boot sequence and jump to the start of L1 memory (0xFFA0 0000 for ADSP-BF533 processor or 0xFFA0 8000 for ADSP-BF531 and ADSP-BF532 processors) for execution. The On-Chip Boot ROM does this by checking bit 4 of

the Reset Configuration Register (SYSCR). If bit 4 is not set, the On-Chip Boot ROM performs the full boot sequence. If bit 4 is set, the On-Chip Boot ROM bypasses the full boot sequence and jumps to the start of L1 memory.

3. Finally, if bit 4 of the SYSCR register is not set, the On-Chip Boot ROM performs the full boot sequence.

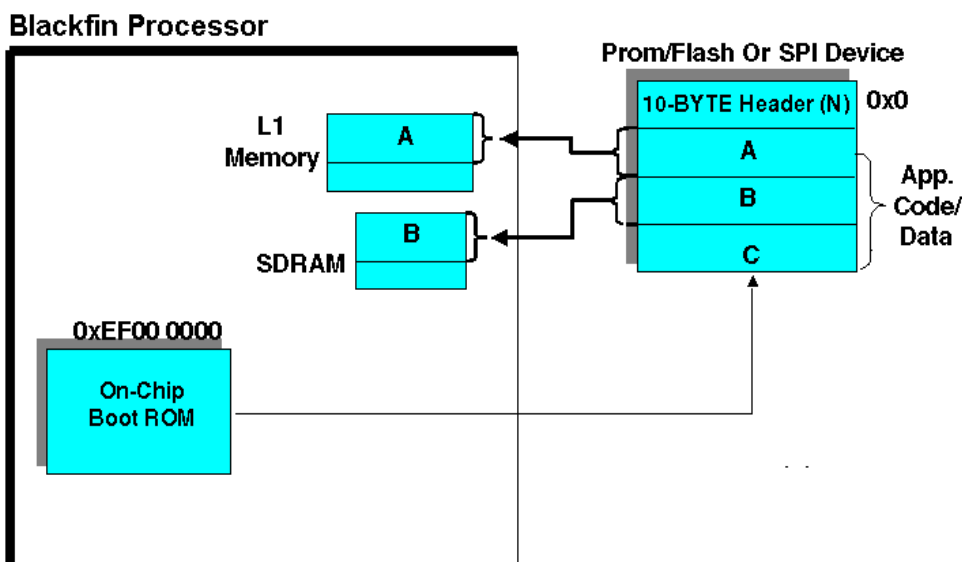


Figure 5-8. Booting in ADSP-BF531/BF532/BF533 Processors

The booting sequence for the ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors is quite different than for the ADSP-BF535 processor. The On-Chip Boot ROM for these processors behaves similar to the second-stage loader of the ADSP-BF535 processor. It has the capability to parse address and count information for each bootable block. This alleviates the need for a second-stage loader for ADSP-BF531/32/33 processors because a full application can be booted to the various Blackfin processor memories with just the On-Chip Boot ROM.

Boot Sequence

The application code .DxE file is converted into blocks by the loader and placed into the .LDR file. Each block is encapsulated within a 10-byte header which is explained in detail in [“Boot Streams for ADSP-BF531, ADSP-BF532, and ADSP-BF533 Processors” on page 5-37.](#)

With this 10-byte header, the On-Chip Boot ROM has all the information on where to boot the block to (Address), how many bytes to boot in (Count), and what to do with the block (Flag).

The On-Chip Boot ROMs on ADSP-BF531 and ADSP-BF532 processors allows booting to:

- L1 Memory
 - ☐ Data Bank A SRAM (0xFF80 4000)
 - ☐ Data Bank B SRAM (0xFF90 4000) (ADSP-BF532 processor only)
 - ☐ Instruction SRAM (0xFFA0 8000)
- SDRAM (0x0000 0000)

The On-Chip Boot ROM on the ADSP-BF533 processor allows booting to:

- L1 Memory
 - ☐ Data Bank A SRAM (0xFF80 0000)
 - ☐ Data Bank B SRAM (0xFF90 0000)
 - ☐ Instruction SRAM (0xFFA0 0000)
- SDRAM (0x0000 0000)



SDRAM must be initialized by user code before instructions/data are loaded into it.

Creating an .LDR File Using the Elfloader

The `elfloader` (or loader) converts the an executable file(s) (.DxE) produced by the linker into a loader file (.LDR). This file is burned/programmed into an external memory device. The Blackfin processor can boot from or execute instructions from this external memory device upon RESET.

You can produce a loader file (.LDR) using the command-line switches described in [“Loader Command-Line Switches” on page 5-18](#) or via the **Load** page of the VisualDSP++ **Project Options** dialog box. The **Load** page may consist of one or several panes opened consequently.

The following sections explains how to create bootable and non-bootable loader files for the Blackfin processors.

This section contains:

- [“Specifying Basic Loader Settings” on page 5-16](#)
- [“Boot Kernel Options for Specifying a Second-Stage Loader” on page 5-24](#)
- [“Creating an .LDR using the ROM Splitter” on page 5-28](#)

Specifying Basic Loader Settings

For all Blackfin processors, the basic **Load** page is the same. For example, [Figure 5-9](#) shows the default **Load** page for booting from PROM for an ADSP-BF532 processor.

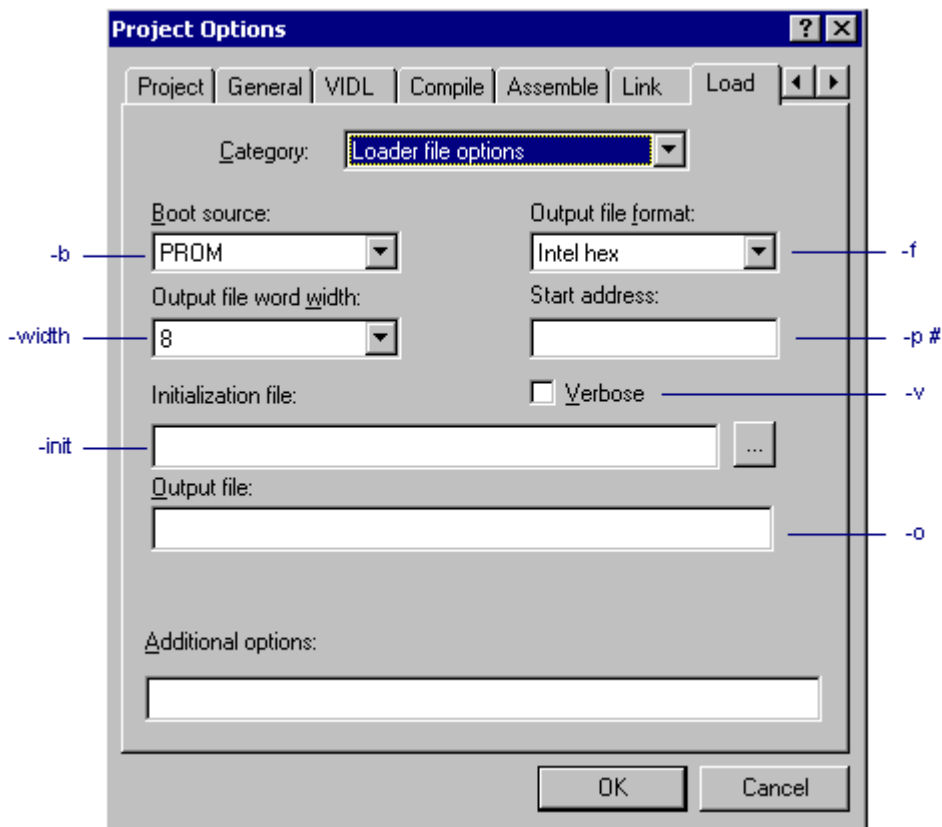


Figure 5-9. Basic Load Page

Refer to [“Load Page Description” on page 5-22](#) for more information on the **Load** page features. Refer to [“Loader Command-Line Switches” on page 5-18](#) for more information on loader switches.

The **Load** page buttons and fields correspond to loader command-line switches and parameters. The loader can also be invoked via the `elfloader.exe` utility. The loader uses the following command-line syntax.

```
elfloader sourcefile -proc processor -switch [-switch ...]
```

where:

- *sourcefile*—Identifies the executable file (.DXE) to be processed for a single-processor boot-loadable file. You may include the drive, directory, file name and file extension. Enclose long file names within straight-quotes, “long file name”.
- *processor*—Specifies the processor (for example, ADSP-BF531) for which the bootloader file is to be built. If the processor is not specified, the default is ADSP-BF535.
- -switch—Processes the optional switch. The loader has many switches to select operations and modes.



Command-line switches may be placed in the command line in any order.

Some loader switches take a file name as an optional parameter. [Table 5-3 on page 5-18](#) lists the expected file types, names, and extensions.

File searches are important in the loader’s process. The loader supports relative and absolute directory names, and default directories. File searches occur as follows.

- *Specified path*—If you include relative or absolute path information in a file name, the loader searches only in that location for the file.
- *Default directory* —If you do not include path information in the file name, the loader searches for the file in the current working directory.

Creating an .LDR File Using the Elfloader

When providing an input or output file as a command-line parameter, use the following guidelines:

- Enclose long file names within straight quotes, “long file name”.
- Append the appropriate file extension to each file.

The loader follows the conventions shown in [Table 5-3](#) for file extensions.

Table 5-3. File Extensions

Extension	File Description
.DXE	Executable files and boot-kernel files. The loader recognizes overlay memory (.OVL) files, but does not expect these files on the command line. Place .OVL files in the same directory as the .DXE file that refers to them so the loader can find them when processing the .LDR file.
.LDR	Loader output file for EPROMs
.KNL	Loader output files containing kernel code only.

Loader Command-Line Switches

Descriptions for each loader switch and file name appear in [Table 5-4](#).

Table 5-4. Blackfin Loader Command-Line Switches¹

Switch	Description
-b prom -b flash -b spi	Specifies the boot mode. The -b switch directs the loader to prepare a bootloadable file for the specified boot mode. Valid boot modes (boot types) include PROM, FLASH, and SPI. If -b does not appear on the command line, the default is -b prom.
-baudrate #	Accepts a baud rate for SPI booting only. Note: Currently supported only for ADSP-BF535 processors. Valid baud rates and corresponding values are: 500K (500 kHz, the default) 1M (1 MHz) 2M (2 MHz) Both 8-bit addressable SPI serial PROMs are supported. Boot kernel loading supports an SPI baud rate up to 2 MHz.

Table 5-4. Blackfin Loader Command-Line Switches¹ (Cont'd)

Switch	Description
-f hex -f ASCII -f binary	Specifies the boot file's format. The -f switch prepares a bootloadable file in the specified format. Valid selections (Intel HEX 32, ASCII, binary) depend on the target processor and boot-type selection (-b switch). For a FLASH or PROM boot-type, select HEX. For SPI booting, select ASCII or binary. If the -f switch does not appear on the command line, the default boot-type format is HEX for FLASH/PROM, and ASCII for SPI.
-h or -help	Command-line help. This switch outputs the list of command-line switches to standard output and exits. For example, type -proc ADSP-BF535 -h to obtain a help file for the ADSP-BF535 processor. By default, the -h switch alone provides help for the loader driver.
-HoldTime #	The -HoldTime # switch allows the loader to specify a number of hold-time cycles for FLASH boot on ADSP-BF535 processors only. The valid values are from 0 through 3. Default value is 3. Note: Currently supported only for ADSP-BF535 processors.
-init filename	Directs the loader to include the initialization block from a <i>filename</i> . When this switch is used, the loader will place the specified .DxE file in the boot stream. The kernel will load this block and then "call" it. It is the responsibility of the code within the block to save/restore state/registers and then perform a RTS back to the kernel. Note: This switch is used with all Blackfin processors except ADSP-BF535 and AD6532 processors.
-kb <i>KernelBootMode</i>	Specifies the boot mode (PROM, FLASH or SPI) for the boot kernel output file if you select to generate two output files from the loader: one for the boot kernel and another for the user application code. This switch must be used in conjunction with the -o2 switch. If the -kb <i>KernelBootMode</i> switch is absent on a command line, the loader generates the file for the boot kernel in the same boot mode as used to output the user application code file.
-kf <i>KernelFormat</i>	Specifies the output file format (HEX, ASCII or binary) for the boot kernel if you select to output two files from the loader: one for the boot kernel and another for the user application code. This switch must be used in conjunction with the -o2 switch. If the -kf <i>KernelFormat</i> switch is absent on the command line, the loader generates the file for the boot kernel in the same format as for the user application code file.

Creating an .LDR File Using the Elfloader

Table 5-4. Blackfin Loader Command-Line Switches¹ (Cont'd)

Switch	Description
-kp #	Specifies a HEX PROM/FLASH output address for kernel code
-kWidth #	<p>Specifies the width of the boot kernel output file when there are two output files: one for boot kernel and one for user application code. Valid values are 8 or 16 for PROM or FLASH, and 8 for SPI.</p> <p>If this switch is absent from a command line, the default file width is:</p> <ul style="list-style-type: none">the -width parameter -- when booting from PROM/FLASH8 -- when booting from SPI. <p>This switch should be used in conjunction with the -o2 switch.</p>
-l <i>userkernel</i>	Specifies the user boot kernel. The loader uses this user-specified kernel and ignores the default boot kernel.
-maskaddr #	<p>Masks all EPROM address bits above or equal to #.</p> <p>For example, -maskaddr 29 masks all the bits above and including A29 (ANDed by 0x1FFF FFFF). For example, 0x2000 0000 becomes 0x0000 0000. The valid #s are integers 0 through 32, but based on your specific input file, the value can be within a subset of [0,32].</p> <p>This switch requires -romsplitter and affects the ROM section address only.</p>
-M	Generates make dependencies only.
-MM	Generates make dependencies while producing the output files
-Mo <i>filename</i>	<p>Writes make dependencies to the <i>filename</i> specified.</p> <p>The -Mo option is for use with either the -M or -MM option. If -Mo is not present, the default is <stdout> display.</p>
-Mt <i>filename</i>	<p>Specifies the make dependencies target name.</p> <p>The -Mt option is for use with either the -M or -MM option. If -Mt is not present, the default is base name plus 'D0J'.</p>
-no2kernel	<p>Produces the output file without the boot kernel, but uses the bootstrap code (from the internal boot ROM). The boot stream generated by the loader is different from the one with the boot kernel loader.</p> <p>Note: Currently supported only for ADSP-BF535 processors.</p>
-o <i>filename</i>	Directs the loader to use the specified the <i>filename</i> as the name for the loader's output file. If not specified, the default name is <i>sourcefile.LDR</i> .

Table 5-4. Blackfin Loader Command-Line Switches¹ (Cont'd)

Switch	Description
-o2	Produces two output files: one for the Init block (if present) and boot kernel and another for the user application code. To have a different format from the application code output file, use the -kf switch to specify the format for the output kernel file.
-p #	Specifies a HEX PROM/FLASH output address for application code
-proc <i>processor</i>	Specifies the processor. The <i>processor</i> can be set as ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF535, ADSP-DM102, or AD6532.
-romsplitter	Creates a non-bootable image only. This switch overwrites -b boot_type and any other switch bounded by boot types. The -romsplitter switch supports HEX and ASCII formats.
-v	Verbose loader messages. This switch outputs status information as the loader processes files.
-waits #	Specifies the number of the wait states for external access. Valid inputs are 0 through 15. Default is 15. Wait states apply to the FLASH/PROM mode only. Note: Currently supported only for ADSP-BF535 processors.
-width #	Specifies the loader output file's word width (in bits). Valid values are 8 and 16, depending on the boot mode. The default is 8. The switch has no effect on boot kernel code processing. The loader processes the kernel in 8-bit widths regardless of selection of the output data width. <ul style="list-style-type: none"> For FLASH/PROM booting, the size of the output file depends on the -width # switch. For SPI booting, the size of the output.LDR file is the same for both -width 8 and -width 16. The only difference is in the header information.

¹ Switches are optional. Items in *italics* are user-definable.

Creating an .LDR File Using the Elfloader

Load Page Description

For all Blackfin processors, the basic **Load** page is the same.

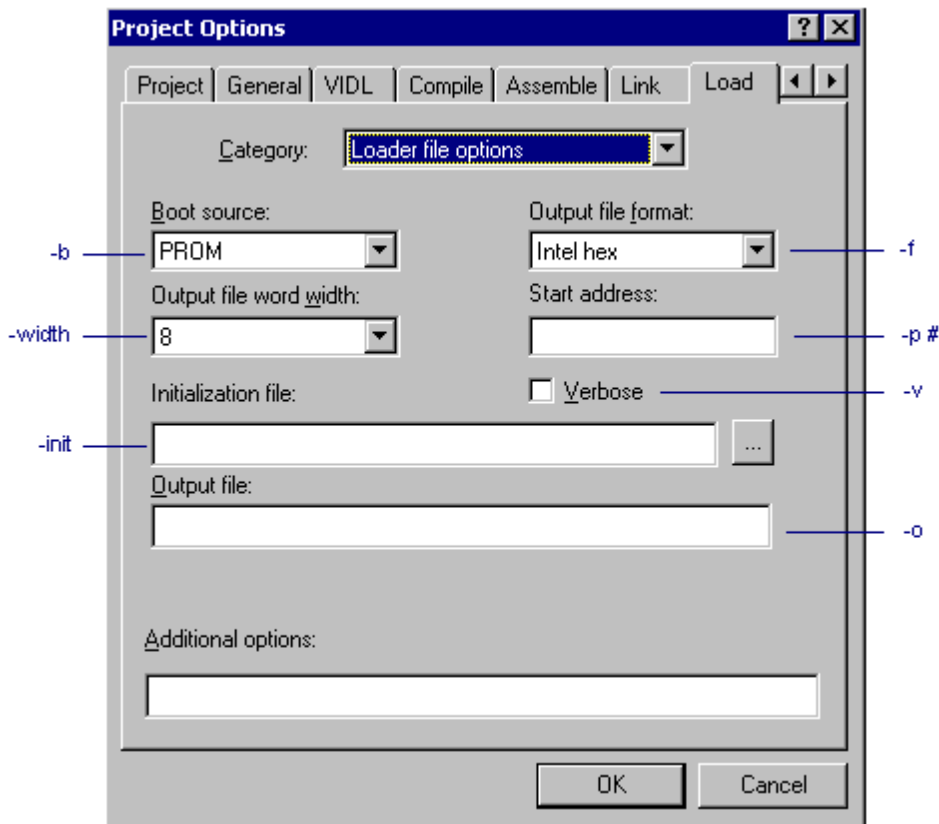


Figure 5-10. Loader Setting Options in Default Load Pane

When you open the **Load** page, the default loader settings for the selected processor are already set. Using the **Load** page, you can select or modify the loader settings.

These settings are:

- Use the **Category** drop-down box to select booting modes. The selections are:

Loader file options — default booting options

Boot kernel options — selecting the second-stage loader

ROM splitter options — selecting the no-boot mode

Note: If you do not use the boot kernel, the second **Load** pane appears with all kernel option fields grayed out. The loader does not search for the boot kernel if you decide to boot from the on-chip ROM only by setting the “-no2kernel” command-line switch [on page 5-20](#).

- **Boot source**—PROM, FLASH, or SPI
- **Output file format**—HEX, ASCII, or Binary
- **Start address**—Specifies a PROM/FLASH output address (in hexadecimal format) for the application code
- **Output file word width**—8 or 16 bits
If `BMODE = 01` or `001`, and FLASH/PROM is 16-bit wide, the “16” bit option must be selected.
- **Verbose**—Generates status information as the loader processes the files
- **Initialization file**—Directs the loader to include the initialization file (INIT code)
Note: The **Initialization file** selection is active for ADSP-BF531, ADSP-BF532 and ADSP-BF533 processors only. For ADSP-BF535 processors, this field is grayed out.
- **Output file**—Enter the name of the loader’s output file (.LDR)
- **Additional options**—Enter additional loader switches.

If you are satisfied with default settings, do not change any settings and click **OK** to complete the loader setup.

Boot Kernel Options for Specifying a Second-Stage Loader

If you decide to use a second-stage loader, select **Boot kernel options** in the **Category** drop-down menu.

Figure 5-11 shows an example **Load** pane for a ADSP-BF535 processor.

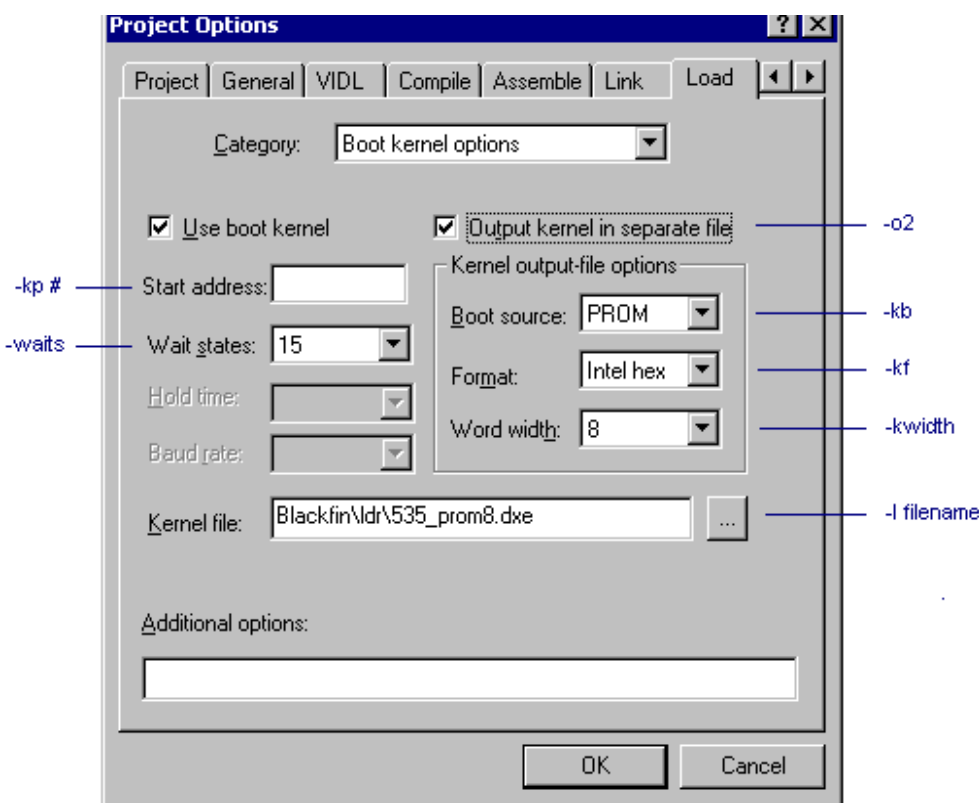


Figure 5-11. Selecting Boot Kernel Settings for ADSP-BF535 Processor

Figure 5-12 shows an example second **Load** pane for a ADSP-BF531, ADSP-BF532, or ADSP-BF533 processor. This page shows how to configure the loader for boot loading and output file generation using the boot kernel.

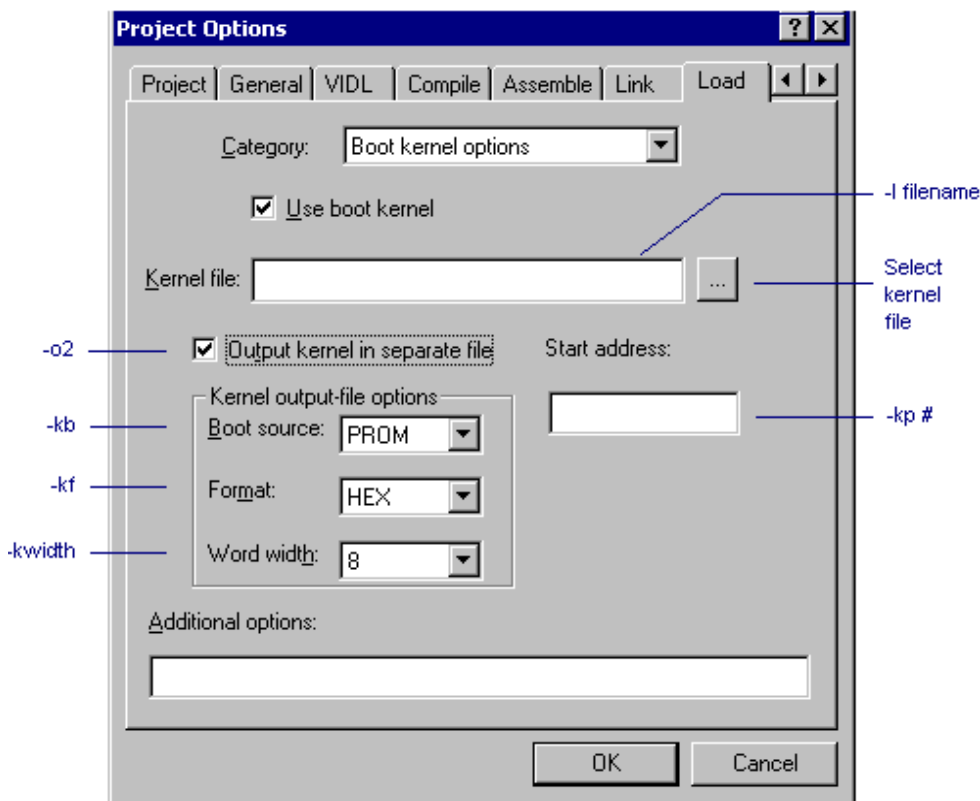


Figure 5-12. Selecting Boot Kernel Settings for ADSP-BF532 Processor

Creating an .LDR File Using the Elfloader

To create a loader file (.LDR) which includes a second-stage loader:

1. Use the **Load file options** pane to set up basic booting options.
2. Select **Boot kernel options** from the **Category** drop-down box to open the second **Load** pane with second-stage loader (boot kernel) settings.
3. Select **Use boot kernel**.
4. If you want to produce two output files (boot kernel file and application code file), select the **Output kernel in separate file** check box. This option allows a user to boot the second-stage loader from one source and the application code from another source. If the **Output kernel in separate file** box is checked, you can specify the **Kernel output file options** such as the **Boot source**, **Format**, and **Word width**.
5. Enter the **Kernel file** (.DXE).

The following second-stage loaders (boot kernels) are currently available for the ADSP-BF535 processor:

Boot Source	Second Stage Loader File (or Boot Kernel File)
8-bit FLASH/PROM	535_prom8.dxe,
16-bit FLASH/PROM	535_prom16.dxe
SPI	535_spi.dxe

Note: For the ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors, no second stage loaders are required and hence no default kernel files are provided. The users can supply their own second-stage loader file, if so desired.

6. Specify the **Start address** (PROM/FLASH output address in hexadecimal format) for the kernel code. This option allows you to place the loader file (.LDR) at a specific location within the FLASH/PROM.
7. For the ADSP-BF535 processors only, you can modify the **Wait states** and **Hold time** cycles for a FLASH/PROM booting or the **Baud rate** for a SPI booting.
8. Click **OK** to complete the loader setup.

Creating an .LDR using the ROM Splitter

The No-Boot Mode option allows users to create an .LDR file for external 16-bit execution. [Figure 5-13](#) shows example **ROM Splitter** settings via the Load page.

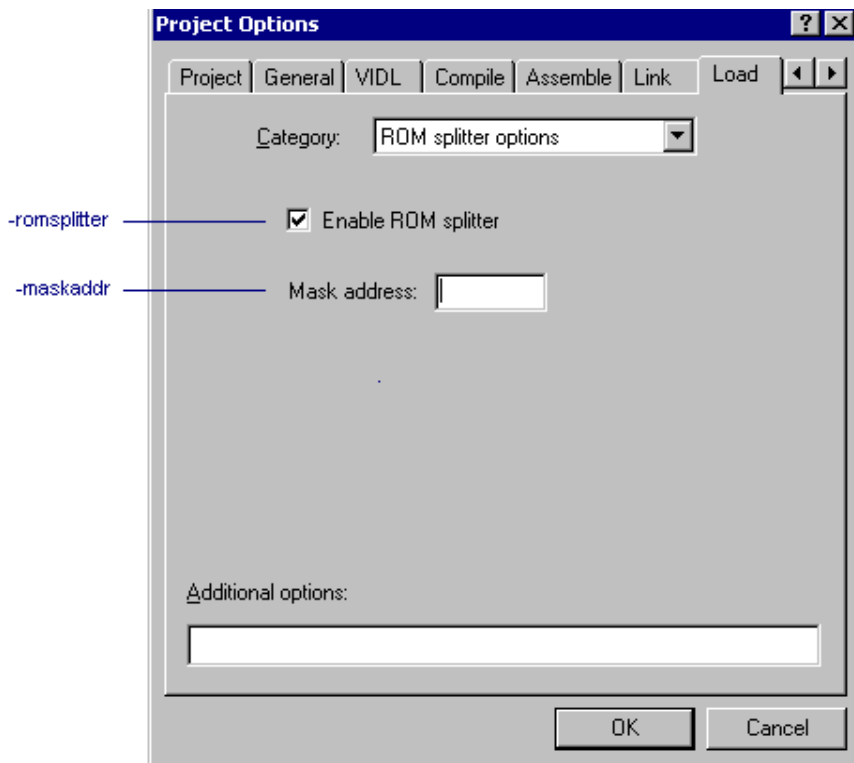


Figure 5-13. ROM Splitter Settings for No Boot Mode

To create an .LDR file for No-Boot mode, select the **ROM splitter options** in the **Category** drop-down menu and enable the **Enable ROM splitter** check box. This mode is used when $BMODE = 000$ for ADSP-BF535 processors and $BMODE = 00$ for ADSP-BF521, ADSP-BF532, and ADSP-BF533 processors.

The **Mask Address** field masks all EPROM address bits above or equal to the number specified. For example, `Mask Address = 29` masks all the bits above and including `A29` (ANDed by `0x1FFF FFFF`).

For example, `0x2000 0000` becomes `0x0000 0000`. The valid #s are integers 0 through 32, but based on your specific input file, the value can be within a subset of `[0, 32]`.

File Formats

The loader generates the boot stream and places the boot stream in the output loader file (.LDR). The loader prepares the boot stream in such a way that the On-Chip Boot ROM (and the second-stage loader for ADSP-BF535 processors) can correctly load the application code and data to the processor memory; therefore, the boot stream contains not only the user application code but also header information that is used by the On-Chip Boot ROM (and the second-stage loader for ADSP-BF535 processors).

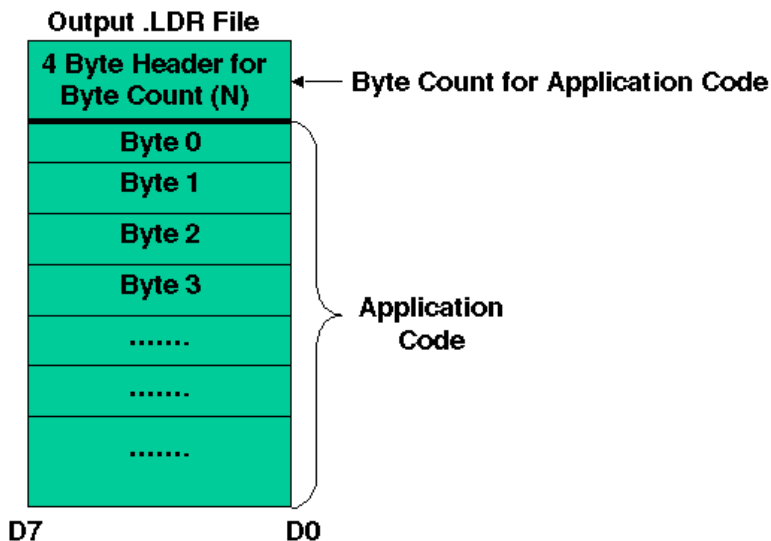
This section provides:

- [“Boot Streams for the ADSP-BF535 Processor”](#)
- [“Boot Streams for ADSP-BF531, ADSP-BF532, and ADSP-BF533 Processors” on page 5-37](#)

Boot Streams for the ADSP-BF535 Processor

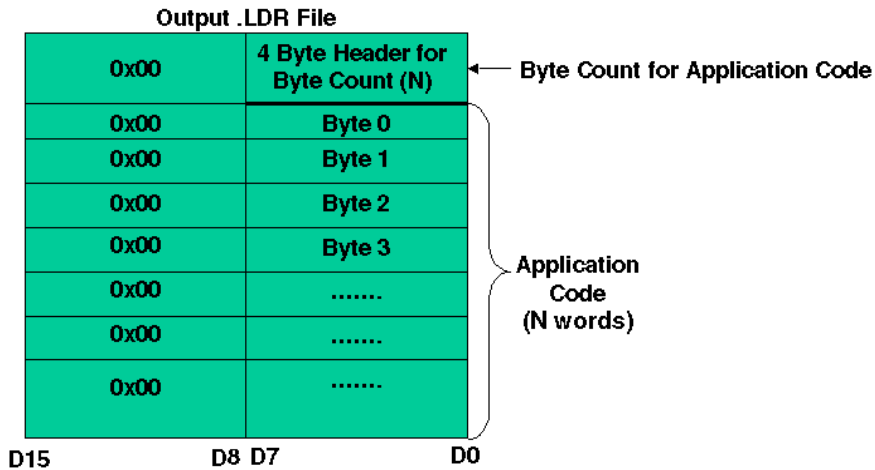
Diagrams in this section illustrate the information included in the boot stream that uses the boot kernel. The boot stream elements described are:

- **Output Loader File**
 - ☐ For 8-bit PROM/FLASH booting and 8-/16-bit addressable SPI booting without the second-stage loader

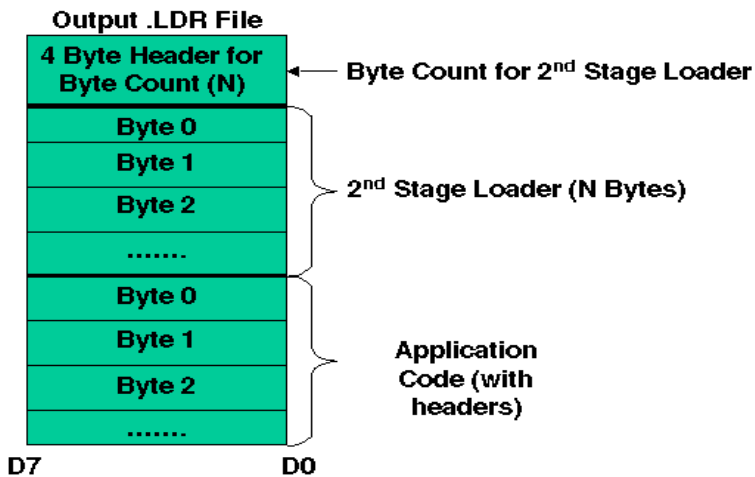


File Formats

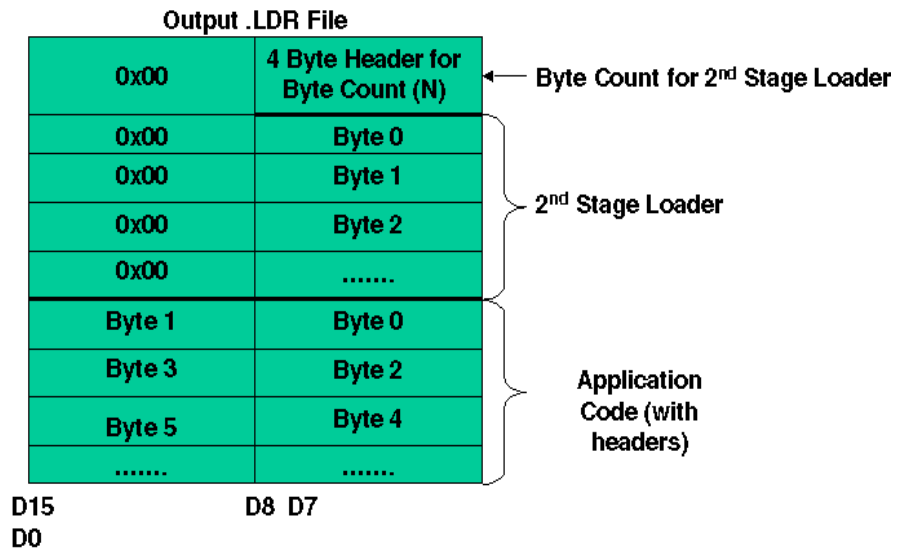
- ☐ For 16-bit PROM/FLASH booting without the second-stage loader



- ☐ For 8-bit PROM/FLASH booting and 8-/16-bit addressable SPI booting with the second-stage loader



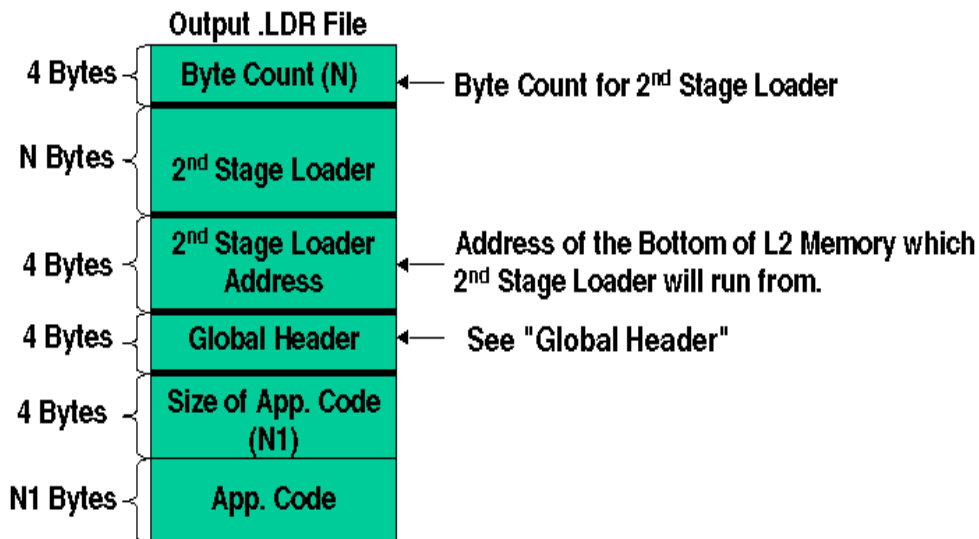
- For 16-bit PROM/FLASH booting with the second-stage loader



File Formats

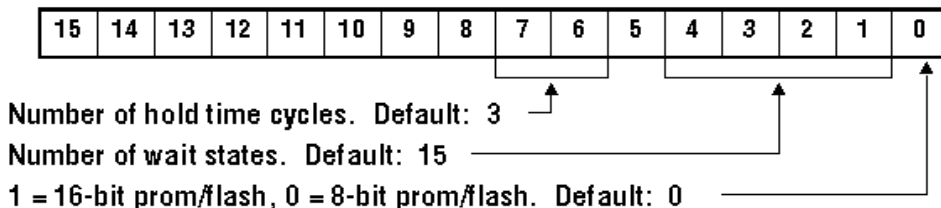
- Header Information

Provides the structure of the output .LDR file with the second-stage loader

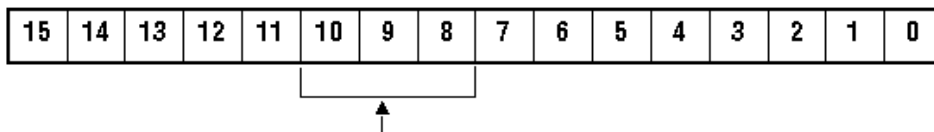


- Global Header

For 8- and 16-bit PROM/FLASH booting



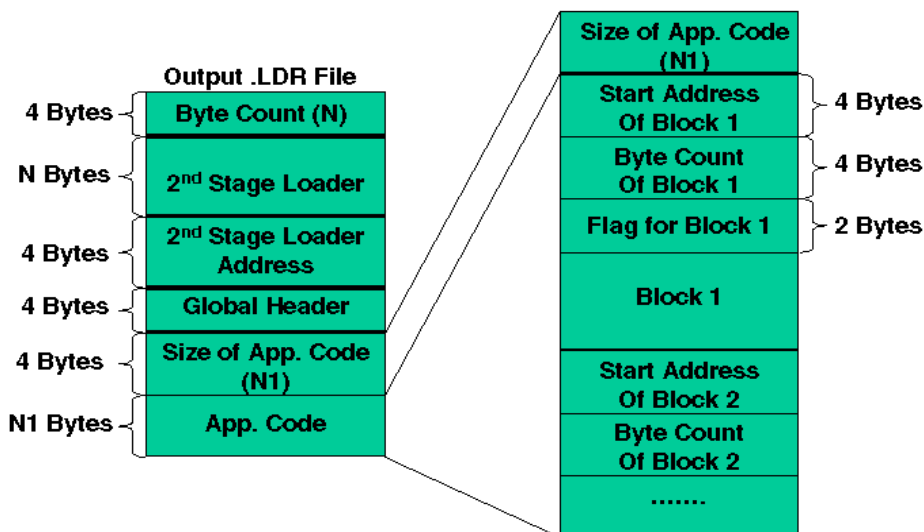
For 8- and 16-bit addressable SPI booting



Baud rate. 0 = 500 kHz, 1 = 1 MHz, 2 = 2 MHz. Default: 500 kHz.

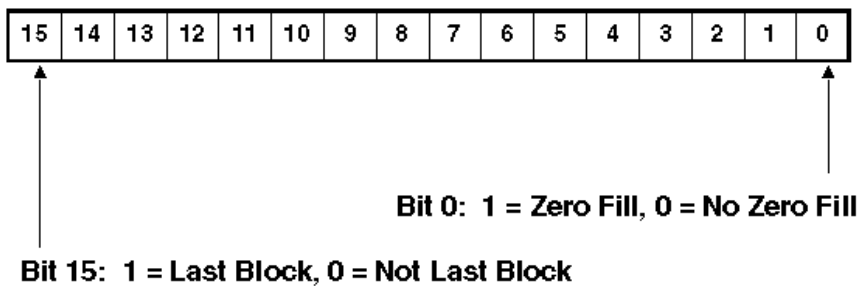
- Header Information (cont.)

Provides the structure of the output .LDR file with the second-stage loader with the application code details



File Formats

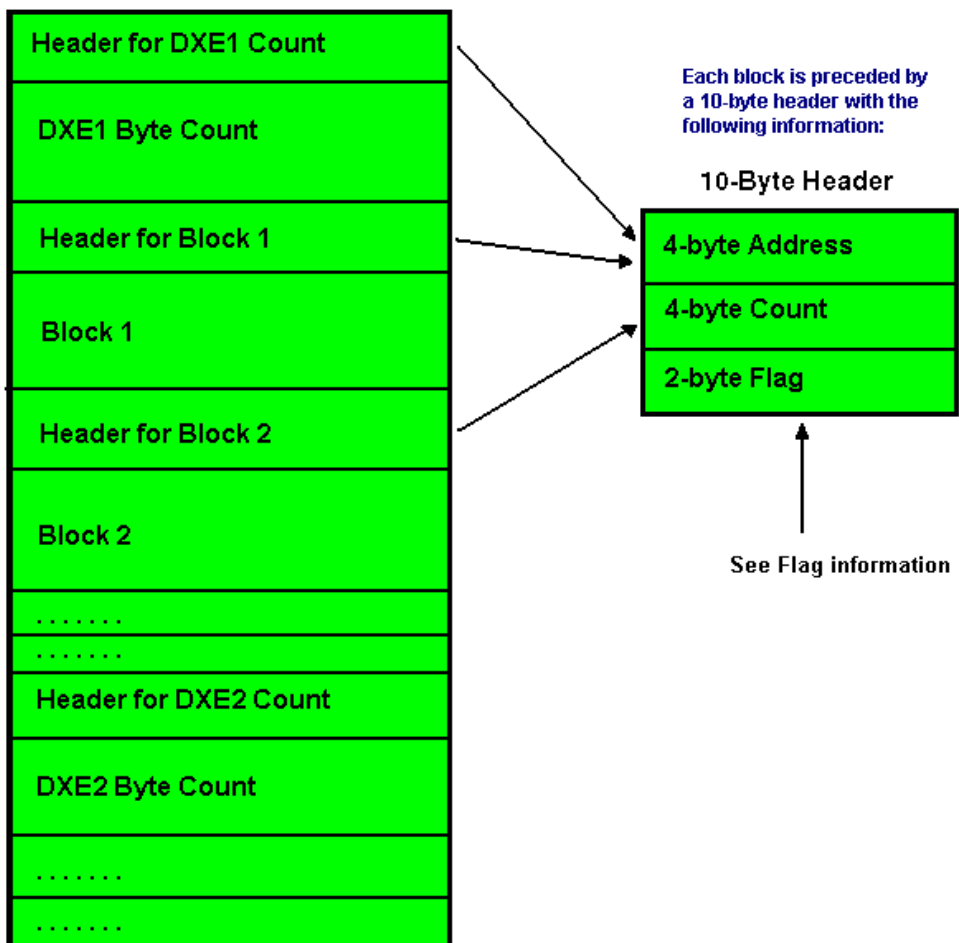
- **Flag Information** (see [on page 5-39](#) for bit descriptions)



Boot Streams for ADSP-BF531, ADSP-BF532, and ADSP-BF533 Processors

The following diagrams show boot stream and flag information for the ADSP-BF531, ADSP-BF532, and ADSP-BF533 processors.

Boot Stream Structure



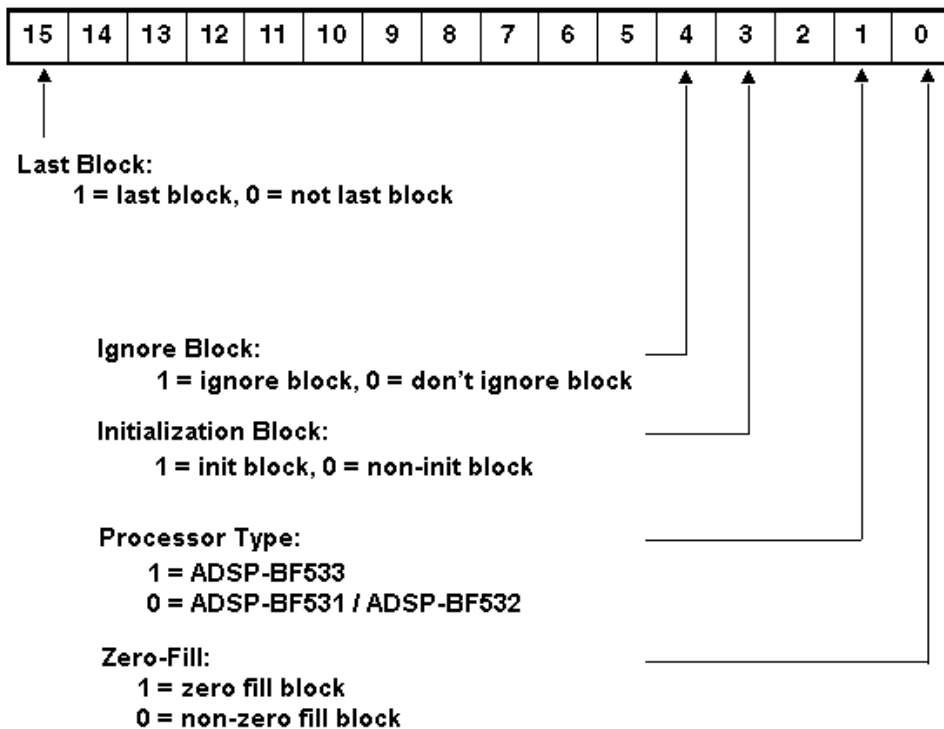
The 10-byte header contains:

- **Address** — where the block resides within memory
- **Count** — how many bytes to boot in
- **Flag** — information about the block (see the register diagram [on page 5-39](#)).
 - **Zero-Fill Block** — indicates that the block is a buffer with zeros. When the loader parses through the .DxE file and encounters a large buffer with zeros, it creates a zero-fill block to reduce .LDR file size and boot time. If this bit is set, then there is no “data” in the block.
 - **Initialization Block** — indicates to execute this block before booting. The Initialization Block indicator allows the On-Chip Boot ROM to execute a number of instructions before booting the actual application code. This option allows the user to run initialization code (such as SDRAM initialization) before the full boot sequence proceeds. Initialization code can be included within the .LDR file by using the `-init` switch (see “[-init filename](#)” [on page 5-19](#)).
 - **Ignore Block** — indicates that block is not to be booted into memory. Skip this block and move on to next one.
 - **Last Block** — indicates whether this block is the last block to be booted into memory. After the last block, processor jumps to the start of L1 memory for execution.

Flag Information



Refer to “-init filename” on page 5-19 for more information on the initialization code section.



A FILE FORMATS

The VisualDSP++ development tools support many file formats, in some cases several for each development tool. This appendix describes file formats that are prepared as input for the tools and points out the features of files produced by the tools.

This appendix describes three types of file formats:

- [“Source Files” on page A-2](#)
- [“Build Files” on page A-5](#)
- [“Debugger Files” on page A-9](#)

Most of the development tools use industry-standard file formats. Sources that describe these formats appear in [“Format References” on page A-10](#).

Source Files

This section describes these input file formats:

- “C/C++ Source Files” on page A-2
- “Assembly Source Files (.ASM)” on page A-3
- “Assembly Initialization Data Files (.DAT)” on page A-3
- “Header Files (.H)” on page A-4
- “Linker Description Files (.LDF)” on page A-4
- “Linker Command-Line Files (.TXT)” on page A-5

C/C++ Source Files

These are text files (with extensions such as .C, .CPP, .CXX, and so on) containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and (typically) preprocessor commands.

Several “dialects” of C code are supported: pure (portable) ANSI C, and at least two subtypes¹ of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives used by the linker to structure and place executable files.

For information on using the C/C++ compiler and associated tools, as well as a definition of ADI extensions to ANSI C, see the *VisualDSP++ 3.1 C/C++ Compiler and Library Manual for Blackfin Processors*.

¹ With and without built-in function support; a minimal differentiator. There are others.

Assembly Source Files (.ASM)

Assembly source files are text files containing assembly instructions, assembler directives, and (optionally) preprocessor commands. For information on assembly instructions, see your DSP's *Programming Reference*.

The DSP's instruction set is supplemented with assembler directives. Preprocessor commands control macro processing and conditional assembly or compilation.

For information on the assembler and preprocessor, see the *VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors*.

Assembly Initialization Data Files (.DAT)

Assembly initialization data (.DAT) files are text files that contain fixed- or floating-point data. These files provide the initialization data for an assembler `.VAR` directive or serve in other tool operations.

When a `.VAR` directive uses a `.DAT` file for data initialization, the assembler reads the data file and initializes the buffer in the output object file (`.DOJ`). Data files have one data value per line and may have any number of lines.

The `.DAT` extension is explanatory or mnemonic. A directive to `#include <file>` can take any file name (or extension) as an argument.

Fixed-point values (integers) in data files may be signed, and they may be decimal-, hexadecimal-, octal-, or binary-base values. The assembler uses the prefix conventions in [Table A-1](#) to distinguish between numeric formats.

For all numeric bases, the assembler uses 16-bit words for data storage; 24-bit data is for the program code only. The largest word in the buffer determines the size for all words in the buffer. If you have some 8-bit data

Source Files

in a 16-bit wide buffer, the assembler loads the equivalent 8-bit value into the most significant 8 bits into the 8-bit memory location and zero-fills the lower eight bits.

Table A-1. Numeric Formats

Convention	Description
<code>0xnumber</code> <code>H#number</code> <code>h#number</code>	Hexadecimal number.
<code>number</code> <code>D#number</code> <code>d#number</code>	Decimal number.
<code>B#number</code> <code>b#number</code>	Binary number.
<code>O#number</code> <code>o#number</code>	Octal number.

Header Files (.H)

Header files are ASCII text files that contain macros or other preprocessor commands that the preprocessor substitutes into source files. For information on macros or other preprocessor commands, see the *VisualDSP++ 3.1 Assembler and Preprocessor Manual for Blackfin Processors*.

Linker Description Files (.LDF)

.LDF files are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language, see [“Linker Command-Line Switches” on page 1-38](#).

Linker Command-Line Files (.TXT)

Linker command-line files (.TXT) are ASCII text files that contain command-line input for the linker. For more information on the linker command line, see [“Linker Command-Line Reference” on page 1-33](#).

Build Files

Build files are produced by the VisualDSP++ development tools when building a project. This section describes these build file formats:

- [“Assembler Object Files \(.DOJ\)” on page A-5](#)
- [“Library Files \(.DLB\)” on page A-6](#)
- [“Linker Output Files \(.DXE, .SM, and .OVL\)” on page A-6](#)
- [“Memory Map Files \(.MAP\)” on page A-6](#)
- [“Loader Output Files in Intel Hex-32 Format \(.LDR\)” on page A-6](#)
- [“Splitter Output Files in ASCII Format \(.LDR\)” on page A-8](#)

Assembler Object Files (.DOJ)

Assembler output object files (.DOJ) are binary, executable and linkable file (ELF) format. Object files contain relocatable code and debugging information for a DSP program’s memory segments. The linker processes object files into an executable file (.DXE). For information on the object file’s ELF format, see the [“Format References” on page A-10](#).

Library Files (.DLB)

Library files, the archiver's output, are in binary, executable and linkable file (ELF) format. Library files (called archive files in previous software releases) contain one or more object files (archive elements).

The linker searches through library files for library members used by the code. For information on the ELF format used for executable files, refer to [“Format References” on page A-10](#).

Linker Output Files (.DXE, .SM, and .OVL)

The linker's output files are binary, executable and linkable file (ELF) format. These executable files contain program code and debugging information. The linker fully resolves addresses in executable files. For information on the ELF format used for executable files, see the TIS Committee texts cited in [“Format References” on page A-10](#).



The archiver automatically converts legacy input objects from COFF to ELF format.

Memory Map Files (.MAP)

The linker can output memory map files (.MAP), which are ASCII text files that contain memory and symbol information for your executable file(s). The map contains a summary of memory defined with `MEMORY{ }` commands in the .LDF file, and provides a list of the absolute addresses of all symbols.

Loader Output Files in Intel Hex-32 Format (.LDR)

The loader can output Intel hex-32 format (.LDR) files. These files support 8-bit-wide PROMs. The files are used with an industry-standard PROM programmer to program memory devices for a hardware system. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel hex-32 format appears in the loader's output file. Each line in the Intel hex-32 file contains an extended linear address record, a data record, or the end-of-file record.

```
:020000040000FA      Extended linear address record
:0402100000FE03F0F9  Data record
:00000001FF          End-of-file record
```

Extended linear address records are used because data records have a 4-character (16-bit) address field, but in many cases, the required PROM size is greater than or equal to 0xFFFF bytes. Extended linear address records specify bits 16-31 for the data records that follow.

[Table A-2](#) shows an example of an extended linear address record.

Table A-2. Example – Extended Linear Address Record

Field	Purpose
:020000040000FA	Example record
:	Start character
02	Byte count (always 02)
0000	Address (always 0000)
04	Record type
0000	Offset address
FA	Checksum

[Table A-3](#) shows the organization of an example data record.

[Table A-4](#) shows an end-of-file record.

Table A-3. Example – Data Record

Field	Purpose
:0402100000FE03F0F9	Example record
:	Start character
04	Byte count of this record
0210	Address
00	Record type
00	First data byte
F0	Last data byte
F9	Checksum

Table A-4. Example – End-of-File Record

Field	Purpose
:00000001FF	End-of-file record
:	Start character
00	Byte count (zero for this record)
0000	Address of first byte
01	Record type
FF	Checksum

Splitter Output Files in ASCII Format (.LDR)

When the loader is invoked as a splitter, it's output can be an ASCII format file. ASCII-format files are text representations of ROM memory images that can be post-processed by users. For more information, refer to no-boot mode .

Debugger Files

Debugger files provide input to the debugger to define support for simulation or emulation of your program. The debugger supports all the executable file types produced by the linker (.DXX, .SM, .OVL). To simulate I/O, the debugger also supports the assembler's data file format (.DAT) and the loader's loadable file formats (.LDR).

The standard hexadecimal format for a SPORT data file is one integer value per line. Hexadecimal numbers do not require a 0x prefix. A value can have any number of digits, but is read into the SPORT register as follows:

- The hexadecimal number is converted to binary.
- The number of binary bits read in matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zero values shorter than the word size or conversely truncates bits beyond the word size on the MSB end.

Example

In this example, a SPORT register is set for 20-bit words and the data file contains hexadecimal numbers. The simulator converts the hex numbers to binary and then fills/truncates to match the SPORT word size. In [Table A-5](#), the A5A5 is filled and 123456 is truncated.

Table A-5. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
A5A5A	1010 0101 1010 0101 1010	1010 0101 1010 0101 1010
FFFF1	1111 1111 1111 1111 0001	1111 1111 1111 1111 0001
A5A5	1010 0101 1010 0101	0000 1010 0101 1010 0101
5A5A5	0101 1010 0101 1010 0101	0101 1010 0101 1010 0101

Format References

Table A-5. SPORT Data File Example (Cont'd)

Hex Number	Binary Number	Truncated/Filled
11111	0001 0001 0001 0001 0001	0001 0001 0001 0001 0001
123456	0001 0010 0011 0100 0101 0110	0010 0011 0100 0101 0110

Format References

The following texts define industry-standard file formats supported by VisualDSP++.

- Gircys, G.R. (1988) *Understanding and Using COFF* by O'Reilly & Associates, Newton, MA
- (1993) *Executable and Linkable Format (ELF) V1.1* from the Portable Formats Specification V1.1, Tools Interface Standards (TIS) Committee.

Go to: <http://developer.intel.com/vtune/tis.htm>.

- (1993) *Debugging Information Format (DWARF) V1.1* from the Portable Formats Specification V1.1, UNIX International, Inc.

Go to: <http://developer.intel.com/vtune/tis.htm>.

B UTILITIES

The VisualDSP++ development software includes several utilities, some of which run from a command line only. Some utilities support legacy code, and others are intended for users who prefer to use the command-line version of the tools instead of accessing them from within the VisualDSP++ environment.

This appendix describes the ELF file dumper utility and mem21k memory initializer utility.

elfdump – ELF File Dumper

The ELF file dumper (`elfdump.exe`) utility extracts data from ELF-format executable files (`.DXE`) yielding a text showing the ELF file's contents.

`elfdump` is often used with the archiver (`elfar.exe`). Refer to [“Disassembling a Library Member” on page B-3](#) for details.

Syntax: `elfdump [switches] [objectfile]`

[Table B-1](#) shows switches used with the `elfdump` command.

Table B-1. ELF File Dumper Command-Line Switches

Switch	Description
-c	Stabs to mdebug conversion
-fh	Prints the file header
-arsym	Prints the library symbol table

Table B-1. ELF File Dumper Command-Line Switches (Cont'd)

Switch	Description
-arall	Prints every library member
-help	Prints the list of elfdump switches to stdout
-ph	Prints the program header table
-sh	Prints the section header table. This is the default when no options are specified
-notes	Prints note segment(s)
-n <i>name</i>	Prints contents of the named section(s). The name may be a simple 'glob'-style pattern, using "?" and "*" as wildcard characters. Each section's name and type determines its output format, unless overridden by a modifier (see below).
-i x0[-x1]	Prints contents of sections numbered x0 through x1, where x0 and x1 are decimal integers, and x1 defaults to x0 if omitted. Formatting rules as are for the -n switch.
-all	Prints everything. Same as -fh -ph -sh -notes -n '*'.
-ost	Omits string table sections
-v	Prints version information
<i>objectfile</i>	<p>The file whose contents are to be printed. It can be a core file, executable, shared library, or relocatable object file. If the name is in the form A(B), A is assumed to be a library and B is an ELF member of the library. B can be a pattern like the one accepted by -n.</p> <p>The -n and -i options can have a modifier letter after the main option character, forcing section contents to be formatted as follows:</p> <ul style="list-style-type: none"> a Dumps contents in hex and ASCII, 16 bytes per line. x Dumps contents in hex, 32 bytes per line. xN Dumps contents in hex, N bytes per group (default is N = 4). t Dumps contents in hex, N bytes per line, where N is the section's table entry size. If N is not in the range 1-32, 32 is used. hN Dumps contents in hexlet, N bytes per group. HN Dumps contents in hexlet (MSB first order), N bytes per group. i Prints contents as list of disassembled machine instructions.

Disassembling a Library Member

The `elfar` and `elfdump` utilities are more effective when their capabilities are combined. One application of these utilities is for disassembling a library member and converting it to source code. Use this technique when the source of a particularly useful routine is missing and is available only as a library routine.

For information about `elfar`, refer to [“Archiver” on page 4-1](#).

The following procedure lists the objects in a library, extracts an object, and converts the object to a listing file. The first archiver command line lists the objects in the library and writes the output to a text file.

```
elfar -p libc.dlb > libc.txt
```



This command assumes the current directory is:

```
C:\Program Files\Analog Devices\VisualDSP\21xxx\lib
```

Open the text file, scroll through it, and locate the object file you need. Then, use the following archiver command to extract the object from the library.


```
elfar -e libc.dlb fir.doj
```

To convert the object file to an assembly listing file with labels, use the following `elfdump` command line.

```
elfdump -ns * fir.doj > fir.asm
```

The output file is practically source code. Just remove the line numbers and opcodes.

Disassembly yields a listing file with symbols. Assembly source with symbols can be useful if you are familiar with the code and hopefully have some documentation on what the code does. If the symbols were stripped during linking, the dumped file contains no symbols.

 Disassembling a third-party's library may violate the license for the third-party software. Ensure there are no copyright or license issues with the code's owner before using this disassembly technique.

Dumping Overlay Library Files

Use the `elfar` and `elfdump` utilities to extract and view the contents of overlay library files (`.OVL`).

For example, the following command lists (prints) the contents (library members) of the `CLONE2.OVL` library file.

```
elfar -p CLONE2.OVL
```


The following command allows you to view one of the library members (`CLONE2.ELF`).

```
elfdump -all CLONE2.OVL(CLONE2.elf)
```

The following commands extract `CLONE2.ELF` and print its contents.

```
elfar -e CLONE2.ovl
```

```
elfdump -all CLONE2.elf
```

 Switches for these commands are case sensitive.

I INDEX

Symbols

`$COMMAND_LINE_LINK_AGAIN`
 ST LDF macro [2-20](#)

`$COMMAND_LINE_OBJECTS`
 LDF macro [2-20](#)

`$COMMAND_LINE_OUTPUT_DIRECTORY`
 LDF macro [2-21](#)

`$COMMAND_LINE_OUTPUT_FILE`
 E LDF macro [2-8](#), [2-21](#)

`$macro` LDF macro [2-21](#)

`$macroname` LDF macro [2-21](#)

`$OBJECTS` LDF macro [2-6](#)

`.align` [8](#) [5-10](#)

`.ASM` files
 assembler [A-3](#)

`.DAT` files
 initialization data [A-3](#)

`.DLB` files [1-35](#), [A-6](#)
 description of [A-6](#)
 symbol name encryption [4-11](#)

`.DOJ` files [1-35](#)
 about [A-5](#)

`.DXE` files [1-6](#), [1-35](#), [A-6](#)
 data extraction [B-1](#)
 linker [A-6](#)

`.H` files [A-4](#)

`.LDF` files [1-35](#), [A-4](#)

 commands in [1-12](#), [2-22](#)

 comments in [2-10](#)

 creating in Expert Linker [3-4](#)

 memory segments [1-13](#)

 operators [2-15](#)

 output sections [1-13](#)

 purpose [1-14](#)

`.LDR` files
 ASCII-format [A-8](#)
 hex-format [A-6](#)
 splitter output [A-8](#)

`.MAP` files [A-6](#)
 linker [A-6](#)

`.OVL` files [1-6](#), [1-35](#), [2-49](#), [A-6](#)
 dumping [B-4](#)
 extracting content from [B-4](#)
 linker [A-6](#)
 viewing contents [B-4](#)

`.SECTION` directive [1-4](#)

`.SM` files [1-6](#), [1-35](#), [A-6](#)
 linker [A-6](#)

`.TXT` files [A-5](#)
 linker [A-5](#)

`@filename` linker command-line switch
 [1-41](#)

`_ov_endaddress_#` [1-51](#), [1-64](#)

`_ov_runtimestartaddress_#` [1-52](#), [1-65](#)

INDEX

`_ov_size_#` [1-51](#), [1-65](#)
`_ov_startaddress_` [1-64](#)
`_ov_startaddress_#` [1-51](#)
`_ov_word_size_live_#` [1-52](#), [1-65](#)
`_ov_word_size_run_#` [1-51](#), [1-65](#)

Numerics

BMODE [5-12](#)
10-byte header [5-14](#), [5-38](#)
BMODE [5-3](#)

A

ABSOLUTE() LDF operator [2-15](#)

adding

input sections [3-10](#)
LDF macros [3-10](#)
library files [3-10](#)
object files [3-10](#)

ADDR() LDF operator [2-16](#)

address

setting for command-line
arguments [1-32](#)

ADSP-BF531 processor

booting [5-12](#)

ADSP-BF532 processor

booting [5-12](#)

ADSP-BF533 processor

booting [5-12](#)

ADSP-BF535 processor

booting [5-3](#)

ALGORITHM() LDF command

[2-49](#)

ALIGN() LDF command [2-23](#)

alignment

specifying properties [3-56](#)
ALL_FIT LDF identifier [3-59](#)

application code

output address [5-21](#)

ARCHITECTURE() LDF

command [2-24](#)

archive files

see library files [A-6](#)

archive members [A-6](#)

archive routines

creating entry points [4-4](#)

archiver

about [4-1](#)

command-line reference [4-7](#)

constraints [4-9](#)

file searches [4-6](#)

running [4-7](#)

symbol name encryption [4-11](#)

use in disassembly [B-3](#)

argv sections [1-28](#)

assembler

initialization data files (.DAT)

[A-3](#)

object files (.DOJ) [A-5](#)

source files (.ASM) [A-3](#)

B

-b flash loader switch [5-18](#)

-b prom loader switch [5-18](#)

-b spi loader switch [5-18](#)

base address

setting for command-line

arguments [1-32](#)

-baudrate # loader switch [5-18](#)

- BEST_FIT LDF identifier [2-49](#)
 - block address [5-38](#)
 - BMODE pin setting
 - ADSP-BF531/32/33 processors [5-12](#)
 - ADSP-BF535 processor [5-3](#)
 - No Boot mode [5-28](#)
 - boot file format.
 - specifying [5-19](#)
 - boot kernel
 - files [5-26](#)
 - output file [5-19](#)
 - user-specified [5-20](#)
 - boot kernel settings
 - ADSP-BF531/32/33 processors [5-25](#)
 - ADSP-BF535 processor [5-24](#)
 - boot mode
 - specifying [5-18](#)
 - boot mode selection
 - ADSP-BF531 processor [5-12](#)
 - ADSP-BF532 processor [5-12](#)
 - ADSP-BF533 processor [5-12](#)
 - ADSP-DM102 processor [5-12](#)
 - boot source
 - selecting [5-5](#)
 - boot stream
 - 10-byte header [5-38](#)
 - ADSP-BF531/32/33 processors [5-37](#)
 - ADSP-BF535 processor [5-31](#)
 - file formats [5-30](#)
 - booting
 - ADSP-BF531 processor [5-12](#)
 - ADSP-BF532 processor [5-12](#)
 - ADSP-BF533 processor [5-12](#)
 - ADSP-BF535 processor [5-3](#)
 - without boot kernel [5-20](#)
 - booting mode
 - No-Boot [5-3](#), [5-12](#), [5-28](#)
 - booting modes
 - ADSP-BF531/32/33 processors [5-12](#)
 - ADSP-BF535 processor [5-3](#)
 - booting sequence
 - ADSP-BF53/32/33 processors [5-13](#)
 - ADSP-BF535 processor [5-4](#)
 - ADSP-BF535 processor with second-stage loader [5-6](#)
 - bootstrap code [5-20](#)
 - bootup
 - sections [1-28](#)
 - branch expansion instruction [1-44](#)
 - branch instructions [2-39](#)
 - build files
 - description of [A-5](#)
 - built-in LDF macros [2-20](#)
- ## C
- C/C++
 - source files [A-2](#)
 - cache
 - memory [1-23](#)
 - calls
 - inter-overlay [1-65](#)
 - inter-processor [1-66](#)
 - color selection

INDEX

- Expert Linker [3-13](#)
- command-line arguments base
 - address
 - setting [1-32](#)
- commands
 - LDF [2-22](#)
 - linker [1-33](#)
- comments
 - .LDF file [2-10](#)
- constdata input section [1-28](#)
- conventions, of this manual [-xxii](#)
- converting
 - library members to source code
 - [B-3](#)
 - out-of-range short calls and jumps
 - [1-44](#)
- count
 - booting [5-38](#)
- Create LDF wizard [3-4](#)
- ctor input section [1-28](#)
- customer support [-xv](#)

D

- Darchitecture (target architecture)
 - linker command-line switch
 - [1-41](#)
- data1 input section [1-28](#)
- debugger
 - files [A-9](#)
- declaring
 - macros [2-21](#)
- DEFINED() LDF operator [2-17](#)
- development tools
 - file formats [A-1](#)

- dialog boxes
 - Legend [3-13](#)
- directories
 - supported by linker [1-36](#)
- disassembling
 - library members [B-3](#)
- disassembly
 - using archiver [B-3](#)
 - using dumper [B-3](#)
- DSPs
 - development software [1-2](#)
- dumper
 - use in disassembly [B-3](#)
- DWARF
 - references [A-10](#)

E

- e (eliminate unused symbols) linker
 - command-line switch [1-43](#)
- ELF file dumper
 - about [B-1](#)
 - command-line switches [B-1](#)
 - extracting data [B-1](#)
 - overlay library files [B-4](#)
 - references [A-10](#)
- elfar.exe
 - about [4-1](#)
 - command-line reference [4-7](#)
- elfdump.exe
 - about [B-1](#)
 - used by Expert Linker [3-35](#)
- elfloader.exe [5-1](#)
- ELIMINATE() LDF command
 - [2-24](#)

ELIMINATE_SECTIONS() LDF

command [2-25](#)

elimination

specifying properties [3-45](#)

encryption

symbol names in libraries [4-11](#)

END() LDF identifier [2-30](#)

-v [5-21](#)

errors

linker [1-21](#)

-es (eliminate listed sections) linker

command-line switch [1-43](#)

-ev (eliminate unused symbols,
verbose) linker command-line
switch [1-43](#)

executable files [A-6](#)

Expert Linker

about [3-1](#)

color selection [3-13](#)

launching [3-3](#)

mapping sections in [3-12](#)

Memory Map pane [3-16](#)

object properties [3-40](#)

overview [1-20](#), [3-1](#)

resize cursor [3-26](#)

window [3-10](#)

extracting data

ELF executable files [B-1](#)

F

-f hex/ASCII/binary loader switch
[5-19](#)

file extensions

loader [5-17](#)

files

.ASM [A-3](#)

.DAT [A-3](#)

.DLB [A-6](#)

.DOJ [A-5](#)

.DXE [A-6](#)

.H [A-4](#)

.LDR (ASCII-format) [A-8](#)

.LDR (hex format) [A-6](#)

.MAP [A-6](#)

.OVL [A-6](#)

.SM [A-6](#)

.TXT [A-5](#)

assembler [A-5](#)

build [A-5](#)

C/C++ [A-2](#)

debugger [A-9](#)

dumping contents of [B-1](#)

executable [A-6](#)

format references [A-10](#)

formats [A-1](#)

header [A-4](#)

input [A-2](#)

library [A-6](#)

linker command-line [1-35](#), [1-37](#)

linker command-line (.TXT) [A-5](#)

output [1-6](#)

FILL() LDF command [2-25](#), [2-48](#)

FIRST_FIT LDF identifier [2-49](#)

G

gaps

inserting [3-32](#)

global properties

INDEX

setting [3-41](#)

H

-h (command-line help) linker
command-line switch [1-43](#)

header files [A-4](#)

heap

graphic representation [3-60](#)

managing in memory [3-60](#)

program section [1-28](#)

-help (command-line help) linker
command-line switch [1-43](#)

hex-format files

.LDR [A-6](#)

-HoldTime # loader switch [5-19](#)

hold-time cycle selection [5-19](#)

I

-i (include search directory) linker
command-line switch [1-44](#)

icons

Expert Linker [3-13](#)

unmapped icon [3-12](#)

Ignore Block indicator [5-38](#)

INCLUDE() LDF command [2-25](#)

-init filename loader switch [5-19](#)

initialization

block [5-38](#)

code [5-38](#), [5-39](#)

initialization code

for loader [5-19](#)

input sections

adding [3-10](#)

directives [1-4](#)

names [1-27](#)

source code [1-3](#)

Input Sections pane [3-10](#)

menu selections [3-10](#)

INPUT_SECTION_ALIGN()

LDF command [2-25](#)

INPUT_SECTIONS() LDF

identifier [2-9](#), [2-47](#)

inserting

gaps [3-32](#)

inter-overlay calls [1-65](#)

inter-processor calls [1-66](#)

J

-jcs21 (convert out-of-range short
calls) linker command-line
switch [1-44](#)

-jcs21+ (convert indirect
calls/jumps) linker
command-line switch [1-44](#)

K

-kb KernelBootMode loader switch
[5-19](#)

-keep (keep unused symbols) linker
command-line switch [1-44](#)

KEEP() LDF command [2-27](#)

kernel code

output address [5-20](#)

-kf KernelFormat loader switch
[5-19](#)

-kp # loader switch [5-20](#)

-kWidth # loader switch [5-20](#)

L

-L path (libraries and objects) linker
command-line switch [1-42](#)

-l userkernel loader switch [5-20](#)

L1 memory [1-23](#), [5-6](#)

segment alignment [5-10](#)

start address [5-12](#)

L2 memory [1-23](#)

reserved for booting [5-10](#)

start address [5-4](#)

Last Block indicator [5-38](#)

LDF

see .LDF files [1-6](#)

LDF commands

about [1-12](#), [2-22](#)

ALIGN() [2-23](#)

ARCHITECTURE() [2-24](#)

ELIMINATE() [2-24](#)

ELIMINATE_SECTIONS()
[2-25](#)

FILL() [2-48](#)

INCLUDE() [2-25](#)

INPUT_SECTION_ALIGN()
[2-25](#)

KEEP() [2-27](#)

LINK_AGAINST() [2-27](#)

MAP() [2-28](#)

MEMORY{} [2-28](#)

MPMEMORY{} [2-31](#)

OVERLAY_GROUP{} [2-33](#)

OVERLAY_INPUT{} [2-48](#)

PLIT{} [2-37](#), [2-48](#)

PROCESSOR{} [2-42](#)

RESOLVE() [2-44](#)

SEARCH_DIR() [2-44](#)

SECTIONS{} [2-45](#)

LDF macros

about [2-19](#)

adding [3-10](#)

list of [2-20](#)

LDF operators

about [2-18](#)

ABSOLUTE() [2-15](#)

ADDR() [2-16](#)

DEFINED() [2-17](#)

MEMORY_SIZEOF() [2-17](#)

SIZEOF() [2-18](#)

Legend dialog box [3-13](#)

legends

Expert Linker [3-11](#)

LENGTH() LDF identifier [2-30](#)

librarian

VisualDSP++ [4-1](#)

libraries

members [4-1](#)

symbol name encryption [4-11](#)

library files

about [A-6](#)

adding [3-10](#)

creating [4-3](#)

searching [4-2](#)

library members [4-1](#)

converting to source code [B-3](#)

disassembling [B-3](#)

library routines

using [4-5](#)

Link page

options [1-16](#)

INDEX

LINK_AGAINST() LDF

command 2-27

linker

about 1-10

command-line files (.TXT) A-5

command-line switches 1-38

command-line syntax 1-33

commands 1-33

describing the target 1-22

error messages 1-21

executable files A-6

file name conventions 1-36

memory map files (.MAP) A-6

objects 1-37

outputs 1-6

overlay constants generated by
1-51

selecting a target processor 1-45

switches 1-12

warning messages 1-21

linker command-line switches

-Darchitecture 1-41

-e 1-43

-es secName 1-43

-ev 1-43

-help 1-43

-i path 1-44

-jcs21 1-44

-jcs21+ 1-44

-keep symbolName 1-44

-L path 1-42

-M 1-42

-Map file 1-42

-MDmacro 1-42

-MM 1-42

-o filename 1-44

-pp 1-45

-proc processor 1-45

-S 1-43

-s 1-45

-sp 1-45

-t 1-45

-T filename 1-43

-v 1-45

-version 1-45

-warnonce 1-46

-xref filename 1-46

Linker Description Files

see .LDF files A-4

linker.exe 1-2

linking

about 1-10

controlling 1-12

file with large uninitialized
variables 2-55

overlay memory system 2-68

single-processor system 2-54

Load page

basic features 5-22

specifying second-stage loader
5-24

loader 5-1

command-line switches 5-18

guide 5-3

hex-format files A-6

initialization code 5-19

output file without boot kernel,
5-20

- selecting output files [5-19](#)
- specifying output files [5-26](#)
- loader file
 - creating [5-15](#)
 - No-Boot mode [5-29](#)
 - with second-stage loader [5-26](#)
- loader settings
 - basic [5-16](#)
 - selections [5-22](#)
- loader switch [5-21](#)
- loader switches
 - b prom|flash|spi [5-18](#)
 - baudrate # [5-18](#)
 - f (format) [5-19](#)
 - HoldTime # [5-19](#)
 - init filename [5-19](#)
 - kb KernelBootMode [5-19](#)
 - kf KernelFormat [5-19](#)
 - kp # [5-20](#)
 - kWidth # [5-20](#)
 - l userkernel [5-20](#)
 - M [5-20](#)
 - maskaddr [5-20](#)
 - MM [5-20](#)
 - Mo filename [5-20](#)
 - Mt filename [5-20](#)
 - no2kernel [5-20](#)
 - o filename [5-20](#)
 - o2 (two output files) [5-21](#)
 - p # [5-21](#)
 - proc processorID [5-21](#)
 - romsplitter [5-21](#)
 - waits # [5-21](#)
 - width # (word width) [5-21](#)

- location counter [2-18](#)
 - definition of [2-18](#)

M

- M (dependency check and output)
 - linker command-line switch [1-42](#)
- M loader switch [5-20](#)
- macros
 - LDF [2-19](#)
 - user-declared [2-21](#)
- Map (file) linker command-line switch [1-42](#)
- MAP() LDF command [2-28](#)
- mapping
 - input sections to output sections [3-12](#)
- maskaddr # loader switch [5-20](#)
- Masking EPROM address bits [5-20](#)
- MDmacro (macro value) linker command-line switch [1-42](#)
- MEM_ARGV memory section [1-28](#)
- MEM_BOOTUP memory section [1-28](#)
- MEM_HEAP memory section [1-28](#)
- MEM_PROGRAM section [1-28](#)
- MEM_STACK memory section [1-28](#)
- MEM_SYSSSTACK memory section [1-28](#)
- memory
 - allocation [1-23](#), [1-27](#)
 - architecture [1-23](#)

INDEX

- architecture representation [1-22](#)
- managing heap/stack [3-60](#)
- overlays [1-47](#)
- partitions [3-16](#)
- segment declaration [1-23](#)
- segments [3-16](#)
- types [1-23](#)
- memory map
 - files [A-6](#)
 - specifying [1-27](#)
- Memory Map pane [3-16](#), [3-17](#)
 - overlays [3-33](#)
- memory maps
 - graphical view [3-22](#)
 - highlighted objects in [3-26](#)
 - tree view [3-21](#)
 - viewing [3-16](#)
- memory overlays [1-48](#)
- memory segments
 - about [1-3](#)
 - changing size of [3-25](#)
 - gap [3-32](#)
 - MEMORY{} command [3-16](#)
 - rules [1-13](#)
 - size [3-21](#)
 - specifying properties [3-51](#)
 - start address [3-21](#)
- memory spaces [1-48](#)
- MEMORY_SIZEOF() LDF
 - operator [2-17](#)
- MEMORY{} LDF command [1-23](#), [2-7](#), [2-28](#)
- MM (dependency check, output and build) linker command-line switch [1-42](#)
- MM loader switch [5-20](#)
- Mo filename loader switch [5-20](#)
- modes
 - booting [5-3](#)
- MPMEMORY{} LDF command [2-31](#)
- Mt filename loader switch [5-20](#)
- multiple overlays [3-33](#)
-
- N
 - no2kernel loader switch [5-20](#)
- No-Boot mode
 - selecting [5-23](#), [5-28](#)
- non-bootable image
 - creating [5-21](#)
- non-bootable mode
 - setting [5-21](#)
-
- O
 - o (output file) linker command-line switch [1-44](#)
 - o filename loader switch [5-20](#)
 - o2 (two output files) loader switch [5-21](#)
 - object files
 - adding [3-10](#)
 - object properties
 - managing with Expert Linker [3-40](#)
 - objects
 - deleting [3-11](#)

- sorting [3-15](#)
- on-chip boot ROM
 - ADSP-BF531/32/33 processors [5-12](#)
 - ADSP-BF535 processor [5-4](#)
- operators
 - LDF [2-15](#)
- output directory
 - specifying [1-44](#)
- output loader file [5-30](#)
- output sections
 - about [1-11](#)
 - dumping [1-27](#)
 - rules [1-13](#)
 - specifying properties [3-53](#)
- OUTPUT() LDF command [2-8](#), [3-16](#)
- ov_id_loaded buffer [1-59](#)
- overlay algorithm
 - ALL_FIT [3-59](#)
- overlay library files
 - viewing
 - archive files [B-4](#)
- overlay manager
 - about [1-47](#), [1-49](#)
 - assembly code [2-39](#)
 - constants [1-57](#)
 - major functions [1-50](#)
 - placing constants [1-58](#)
 - PLIT table [1-53](#)
 - routine steps [1-59](#)
- overlay memory
 - linking for [2-68](#)
- OVERLAY_GROUP{} LDF
 - command [2-33](#)
- OVERLAY_ID LDF identifier [2-49](#)
- OVERLAY_INPUT{} LDF
 - command [2-48](#)
- overlays
 - address [1-52](#)
 - constants [1-51](#), [1-57](#)
 - dumping library files [B-4](#)
 - grouped [2-33](#)
 - grouping [2-33](#)
 - in Memory Map pane [3-33](#)
 - live space [3-33](#)
 - loading instructions with PLIT [2-41](#)
 - managing properties [3-58](#)
 - memory [1-47](#), [1-48](#)
 - multiple [3-33](#)
 - reducing overhead [1-60](#)
 - run space [3-34](#)
 - symbols [1-64](#)
 - ungrouped [2-33](#)
 - word size [1-52](#)
- P**
 - p # loader switch [5-21](#)
 - packing
 - specifying properties [3-55](#)
 - pinning
 - to output section [3-19](#)
 - pins
 - selecting 16-bit external memory booting [5-3](#), [5-12](#)
 - selecting 16-bit SPI booting [5-3](#),

INDEX

- 5-12
 - selecting 8-bit FLASH memory
 - booting 5-3, 5-12
 - selecting 8-bit SPI booting 5-3, 5-12
 - PLIT
 - about 1-52
 - allocating space for 2-39
 - constants 2-39
 - executing user-defined code 1-52
 - overlay management 1-49
 - resolving inter-overlay calls 1-65
 - specifying properties 3-44
 - summary 2-41
 - syntax 2-38
 - PLIT commands
 - PLIT_SYMBOL_ADDRESS
 - 2-38
 - PLIT_SYMBOL_OVERLAYID
 - 2-38
 - PLIT_SYMBOL_ADDRESS 2-38
 - PLIT_SYMBOL_OVERLAYID
 - 2-38
 - PLIT{} LDF command
 - about 2-37
 - in SECTIONS{} 2-48
 - pp (end after preprocessing) linker
 - command-line switch 1-45
 - preprocessor
 - running from linker 1-45
 - proc (processor)
 - linker command-line switch 1-45
 - proc processorID loader switch
 - 5-21
 - procedure linkage table (PLIT)
 - about 1-52
 - PLIT{} 2-37
 - using 1-64
 - processor
 - selection 1-41
 - PROCESSOR{} LDF command
 - 2-42
 - processors
 - specifying properties 3-43
 - program
 - sections 1-28
 - program counter 1-57
 - project builds
 - linker 1-15
 - Project Options dialog box
 - Link page 1-16
- ## Q
- qualifier 5-10
- ## R
- references
 - file formats A-10
 - resize cursor 3-26
 - RESOLVE() LDF command 2-28, 2-44
 - RESOLVE_LOCALLY() LDF
 - command 2-50
 - ROM splitter
 - mask address field 5-29
 - settings 5-28
 - romsplitter loader switch 5-21

S

- s (strip all symbols) linker
 - command-line switch [1-45](#)
- S (strip debug symbols) linker
 - command-line switch [1-43](#)
- SDRAM memory [5-6](#)
 - configuring [5-10](#)
 - initializing [5-9](#), [5-14](#), [5-38](#)
- SEARCH_DIR() LDF command [2-44](#)
- second-stage loader [5-5](#)
 - booting sequence [5-6](#)
 - restrictions [5-10](#)
 - selecting [5-23](#)
 - setting boot kernel options [5-24](#), [5-26](#)
- second-stage loaders
 - ADSP-BF535 processor [5-26](#)
- SECTIONS{} LDF command [1-30](#), [2-8](#), [2-45](#)
- selecting
 - target processor [1-45](#)
- setting
 - command-line arguments base address [1-32](#)
 - second-stage loader [5-26](#)
- SHT_NOBITS
 - keyword [2-56](#)
 - section qualifier [2-55](#), [2-56](#)
- SIZE() LDF command [2-50](#)
- SIZEOF() LDF operator [2-18](#)
- software
 - development [1-2](#)
- sorting
 - objects [3-15](#)
- source code
 - in input sections [1-3](#)
- source files
 - assembly instructions [A-3](#)
 - C/C++ [A-2](#)
 - fixed-point data [A-3](#)
- sp (skip preprocessing) linker
 - command-line switch [1-45](#)
- SPI booting
 - baudrate [5-18](#)
- splitter
 - ASCII-format files (.LDR) [A-8](#)
 - specifying [5-21](#)
- SPORT data files [A-9](#)
- stack
 - sections [1-28](#)
- stacks
 - graphic representation [3-60](#)
 - managing in memory [3-60](#)
- symbol declaration [2-6](#)
- symbols
 - adding [3-48](#)
 - deleting [3-50](#)
 - encryption of names [4-11](#)
 - managing properties [3-47](#)
 - removal [1-45](#)
 - viewing [3-39](#)
- SYSCR register
 - ADSP-BF531/32/33 processors [5-13](#)
 - ADSP-BF535 processor [5-4](#)
- sysstack
 - sections [1-28](#)

T

- t (trace) linker command-line switch [1-45](#)
- T file (executable program placement)
linker command-line switch [1-43](#)
- target processor
specifying [1-41](#)
- tree view
memory map [3-21](#)

U

- uninitialized variables [2-55](#)
- unmapped object icon [3-12](#)
- user-declared macros [2-21](#)
- utilities
 - archiver [4-1](#)
 - elfar.exe [4-1](#)
 - elfdump.exe [B-1](#)
 - file dumper [B-1](#)
 - linker.exe [1-2](#)

V

- v (verbose) linker command-line switch [1-45](#)
- version (linker version) linker
command-line switch [1-45](#)
- viewing
.OVL file content [B-4](#)
- VisualDSP++
 - archiver [4-1](#)
 - creating library files [4-3](#)
 - Expert Linker [3-2](#)
 - librarian [4-1](#)

W

- wait states [5-21](#)
- waits # loader switch [5-21](#)
- warnings
 - linker [1-21](#)
- warnonce (single symbol warning)
linker command-line switch [1-46](#)
- width # (word) loader switch [5-21](#)
- wizards
 - Create LDF [3-4](#)
- word width used in loader output file
[5-21](#)

X

- xref (external reference file) linker
command-line switch [1-46](#)

Z

- zero-fill block [5-38](#)