VisualDSP++™ 3.5 Linker and Utilities Manual for 16-Bit Processors

Revision 1.0, October 2003

Part Number
82-000035-07

Analog Devices, Inc.
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, and the Blackfin logo are registered 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 xv
Intended Audience xvi
Manual Contents xvi
What's New in This Manual xvii
Technical or Customer Support xviii
Supported Processors xix
Product Information xx
MyAnalog.com xx
DSP Product Information xx
Related Documents xxi
Online Technical Documentation xxii
From VisualDSP++ xxii
From Windows xxiii
From the Web xxiii
Printed Manuals xxiv
VisualDSP++ Documentation Set xxiv
Hardware Manuals xxiv
Data Sheets xxiv

Contacting DSP Publications xxv
Notation Conventions xxv

INTRODUCTION

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
Linker and Assembler Preprocessor 1-7
Loading and Splitting 1-8

LINKER

Linker Operation 2-2
Directing Linker Operation 2-3
Linking Process Rules 2-4
Linker Description File — Overview 2-5
Linking Environment 2-6
Project Builds 2-6
Expert Linker 2-9
Linker Warning and Error Messages 2-10
Link Target Description 2-11
Representing Memory Architecture 2-11
ADSP-BF535 Processor Memory Architecture Overview 2-12

ADSP-218x DSP Core Architecture Overview 2-15
ADSP-219x DSP Architecture Overview 2-17
Specifying the Memory Map 2-18
Memory Usage 2-18
Memory Characteristics 2-20
Linker MEMORY{} Command in .LDF File 2-24
Placing Code on the Target 2-26
Passing Arguments for Simulation or Emulation: Blackfin Processors ONLY 2-29
Linker Command-Line Reference 2-30
Linker Command-Line Syntax 2-30
Command-Line Object Files 2-31
Command-Line File Names 2-32
Object File Types 2-34
Linker Command-Line Switches 2-34
Linker Switch Summary 2-36
@filename 2-38
-Dprocessor 2-38
-L path 2-39
-M 2-39
-MM 2-39
-Map filename 2-39
-MDmacro[=def] 2-39
-Ovcse 2-40
-S 2-40

-T filename 2-40
-Wwarn [number] 2-40
-e 2-41
-es sectionName 2-41
-ev 2-41
-flags-meminit -opt1[,-opt2... 2-41
-flags-pp -opt1[,-opt2...] 2-42
-h[elp] 2-42
-i|I directory 2-42
-ip 2-42
-jcs2l 2-43
-jcs2l+ 2-44
-keep symbolName 2-44
-meminit 2-44
-o filename 2-44
-od directory 2-45
-pp 2-45
-proc processor 2-45
-s 2-46
-save-temps 2-46
-si-revision version 2-46
-sp 2-48
-t 2-48
-v[erbose] 2-48

-version 2-48
-warnonce 2-48
-xref filename 2-49

LINKER DESCRIPTION FILE

LDF File Overview 3-3
Example 1 – Basic .LDF File for Blackfin Processors 3-4
Example 2 - Basic .LDF File for ADSP-218/9x DSPs 3-6
Notes on Basic .LDF File Examples 3-7
LDF Structure 3-11
Command Scoping 3-12
LDF Expressions 3-13
LDF Keywords, Commands, and Operators 3-14
Miscellaneous LDF Keywords 3-15
LDF Operators 3-16
ABSOLUTE() Operator 3-16
ADDR() Operator 3-17
DEFINED() Operator 3-18
MEMORY_SIZEOF() Operator 3-18
SIZEOF() Operator 3-19
Location Counter (.) 3-19
LDF Macros 3-20
Built-In LDF Macros 3-21
User-Declared Macros 3-22
LDF Macros and Command-Line Interaction 3-22

LDF Commands 3-23
ALIGN() 3-24
ARCHITECTURE() 3-24
ELIMINATE() 3-25
ELIMINATE_SECTIONS() 3-26
INCLUDE() 3-26
INPUT_SECTION_ALIGN() 3-26
KEEP() 3-27
LINK_AGAINST() 3-28
MAP() 3-29
MEMORY{} 3-29
Segment Declarations 3-30
MPMEMORY{} 3-32
OVERLAY_GROUP{} 3-33
PACKING() 3-33
Packing in ADSP-218x and ADSP-219x DSPs 3-34
Efficient Packing 3-35
Inefficient Packing: Null Bytes 3-36
Overlay Packing Formats 3-37
Trivial Packing: No Reordering 3-37
PAGE_INPUT() 3-37
PAGE_OUTPUT() 3-38
PLIT{} 3-38
PROCESSOR{} 3-39

RESOLVE() 3-40
SEARCH_DIR() 3-41
SECTIONS{} 3-42
INPUT_SECTIONS() 3-45
expression 3-45
FILL(hex number) 3-46
PLIT{plit_commands} 3-46
OVERLAY_INPUT{overlay_commands} 3-46
SHARED_MEMORY{} 3-48

EXPERT LINKER

Expert Linker Overview 4-2
Launching the Create LDF Wizard 4-4
Step 1: Specifying Project Information 4-5
Step 2: Specifying System Information 4-6
Step 3: Completing the LDF Wizard 4-9
Expert Linker Window Overview 4-10
Input Sections Pane 4-12
Input Sections Menu 4-12
Mapping an Input Section to an Output Section 4-14
Viewing Icons and Colors 4-14
Sorting Objects 4-17
Memory Map Pane 4-18
Context Menu 4-20
Tree View Memory Map Representation 4-22

Graphical View Memory Map Representation 4-23
Specifying Pre- and Post-Link Memory Map View 4-27
Zooming In and Out on the Memory Map 4-28
Inserting a Gap into a Memory Segment 4-31
Working With Overlays 4-33
Viewing Section Contents 4-35
Viewing Symbols 4-39
Profiling Object Sections 4-40
Adding Shared Memory Segments and Linking Object Files ... 4-45
Managing Object Properties 4-50
Managing Global Properties 4-51
Managing Processor Properties 4-52
Managing PLIT Properties for Overlays 4-54
Managing Elimination Properties 4-55
Managing Symbols Properties 4-57
Managing Memory Segment Properties 4-61
Managing Output Section Properties 4-62
Managing Packing Properties 4-64
Managing Alignment and Fill Properties 4-65
Managing Overlay Properties 4-67
Managing Stack and Heap in Processor Memory 4-69

MEMORY OVERLAYS AND ADVANCED LDF COMMANDS

Overview 5-2

Memory Management Using Overlays 5-4
Introduction to Memory Overlays 5-5
Overlay Managers 5-7
Breakpoints on Overlays 5-7
Memory Overlay Support 5-8
Example – Managing Two Overlays 5-12
Linker-Generated Constants 5-15
Overlay Word Sizes 5-15
Storing Overlay ID 5-16
Overlay Manager Function Summary 5-17
Reducing Overlay Manager Overhead 5-17
Using PLIT{} and Overlay Manager 5-22
Inter-Overlay Calls 5-24
Inter-Processor Calls 5-24
Advanced LDF Commands 5-27
MPMEMORY{} 5-28
OVERLAY_GROUP{} 5-29
Ungrouped Overlay Execution 5-31
Grouped Overlay Execution 5-33
PLIT{} 5-34
PLIT Syntax 5-34
Command Evaluation and Setup 5-35
Allocating Space for PLITs 5-36
PLIT Example 5-37

PLIT – Summary 5-37
SHARED_MEMORY{} 5-38

ARCHIVER

Archiver Guide 6-2
Creating a Library From VisualDSP++ 6-3
Making Archived Functions Usable 6-3
Writing Archive Routines: Creating Entry Points 6-4
Accessing Archived Functions From Your Code 6-5
Archiver File Searches 6-6
Tagging an Archive with Version Information 6-6
Basic Version Information 6-6
User-Defined Version Information 6-7
Printing Version Information 6-8
Removing Version Information from an Archive 6-9
Checking Version Number 6-9
Adding Text to Version Information 6-10
Archiver Command-Line Reference 6-11
elfar Command Syntax 6-11
Archiver Parameters and Switches 6-12
Command-Line Constraints 6-14
Archiver Symbol Name Encryption 6-15

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
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 (.XML) 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

LDF PROGRAMMING EXAMPLES FOR BLACKFIN PROCESSORS

Linking for a Single-Processor System C-2
Linking Large Uninitialized or Zero-initialized Variables C-4

Linking for Assembly Source File C-6
Linking for C Source File – Example 1 .... C-8
Linking for Complex C Source File – Example 2 .... C-11
Linking for Overlay Memory C-17

LDF PROGRAMMING EXAMPLES FOR ADSP-21XX DSPS

Linking for a Single-Processor ADSP-219x System D-3
Linking Large Uninitialized or Zero-initialized Variables D-5
Linking an Assembly Source File D-7
Linking a Simple C-Based Source File D-9
Linking Overlay Memory for an ADSP-2191 System D-16
Linking an ADSP-219x MP System With Shared Memory D-19
Overlays Used With ADSP-218x DSPs D-23

INDEX

PREFACE

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

Purpose of This Manual

The *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* contains information about the linker and utilities programs for 16-bit fixed-point Blackfin® processors and ADSP-21xx DSPs.

The Blackfin processors are 16-bit fixed-point embedded processors that support 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.

The ADSP-218x and ADSP-219x DSPs are low-cost 16-bit fixed-point DSPs for use in computing, communications, and consumer applications.

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 embedded processors and 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, "Introduction"
  This chapter provides an overview of the linker and utilities.
- Chapter 2, "Linker"
  This chapter describes how to combine object files into reusable library files to link routines referenced by other object files.
- Chapter 3, "Linker Description File"
  This chapter describes how to write an `.LDF` file to define the target.
- Chapter 4, "Expert Linker"
  This chapter describes Expert Linker, which is an interactive graphical tool to set up and map DSP memory.
- Chapter 5, "Memory Overlays and Advanced LDF Commands"
  This chapter describes how overlays and advanced LDF commands are used for memory management.

- Chapter 6 "Archiver"
  This chapter describes the `elfar` archiver utility used to combine object files into library files, which serve as reusable resources for code development.
- 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 utilities that provide legacy and file conversion support.
- Appendix C, "LDF Programming Examples for Blackfin Processors"
  This appendix provides code examples of `.LDF` files used with Blackfin processors.
- Appendix D, "LDF Programming Examples for ADSP-21xx DSPs"
  This appendix provides code examples of `.LDF` files used with ADSP-21xx DSPs.

What's New in This Manual

This is a new manual that documents support for 16-bit fixed-point Blackfin processors and ADSP-21xx DSPs.

This manual now combines linker-related information for all ADI 16-bit fixed-point processors. The manual provides information for Blackfin processors, ADSP-218x DSPs and ADSP-219x DSPs.

Loader/splitter information is now available in separate Loader manuals for appropriate target processor families.

Refer to *VisualDSP++ 3.5 Product Bulletin for 16-Bit Processors* for information on all new and updated features and other release information.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

- Visit the DSP Development Tools website at `http://www.analog.com/technology/dsp/developmentTools/index.html`
- 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.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

Blackfin Processors

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

- ADSP-BF532 (formerly ADSP-21532)
- ADSP-BF535 (formerly ADSP-21535)
- ADSP-BF531
- ADSP-BF533
- ADSP-BF561
- AD6532
- AD90747

The ADSP-BF531 and ADSP-BF533 processors are memory variants of the ADSP-BF532 processor as well as a dual-core ADSP-BF561 processor.

ADSP-218x and ADSP-219x DSPs

The name "*ADSP-21xx*" refers to two families of Analog Devices 16-bit, fixed-point processors. VisualDSP++ currently supports the following processors:

- **ADSP-218x DSPs**: ADSP-2181, ADSP-2183, ADSP-2184/84L/84N, ADSP-2185/85L/85M/85N, ADSP-2186/86L/86M/86N, ADSP-2187L/87N, ADSP-2188L/88N, and ADSP-2189M/89N

- **ADSP-219x DSPs**: ADSP-2191, ADSP-2192-12, ADSP-2195, ADSP-2196, ADSP-21990, ADSP-21991, and ADSP-21992

Product Information

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

Analog Devices is online at `www.analog.com`. Our website 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 website that allows customization of a webpage 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 webpages 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, simply log on. Your user name is your e-mail address.

DSP Product Information

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

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

- Email 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 website 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++ 3.5 Getting Started Guide for 16-Bit Processors

VisualDSP++ 3.5 User's Guide for 16-Bit Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.5 C Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.5 Product Bulletin for 16-Bit Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-21xx DSPs

VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x DSPs

VisualDSP++ 3.5 Loader Manual for 16-Bit Processors

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

For hardware information, refer to your DSP's *Hardware Reference* manual and datasheet.

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.

Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or Analog Devices website.

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** 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 the VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.
- Access the `.PDF` files by clicking the **Start** button and choosing **Programs**, **Analog Devices**,, **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 website. 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 website. 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 website. Data sheets without the suffix can be downloaded from the website only—no hard copies are available. You can ask for the data sheet by a part name or by product number.

If you want to have a data sheet faxed to you, the fax 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 @ `dsp.techpubs@analog.com`.

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 `this` or `that`.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[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 `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.

Example	Description
(note symbol)	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.
(caution 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 INTRODUCTION

This chapter provides an overview of VisualDSP++ development tools and their use in DSP project development process.

This chapter includes:

- "Software Development Flow" on page 1-2
  Shows how linking, loading, and splitting fit into the DSP project development process.
- "Compiling and Assembling" on page 1-3
  Shows how compiling and assembling the code fits into the DSP project development process.
- "Linking" on page 1-6
  Shows how linking fits into the DSP project development process.
- "Loading and Splitting" on page 1-8
  Shows how loading and splitting fit into the DSP project development process.

Software Development Flow

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

The linker tool (`linker.exe`) consumes object and library files to produce executable files, which can be loaded onto a 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 a simulator or emulator. Refer to the *VisualDSP++ User's Guide* of your target 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. Separate *Loader Manual for 16-Bit Processors* describes loader/splitter functionality for the target processors.

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 (`.DXE`) file. If specified, shared memory (`.SM`) and overlay (`.OVL`) files are also produced.

3. Loading or splitting – The executable (`.DXE`), as well as shared memory (`.SM`) and overlay (`.OVL`) files, are processed to yield output file(s). For Blackfin processors, these are boot-loadable (`LDR`) files or non-bootable PROM image files, which execute from the processor's external memory.

Compiling and Assembling

The process starts with source files written in C, C++, or assembly. The compiler (or a code developer who writes 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 file 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).

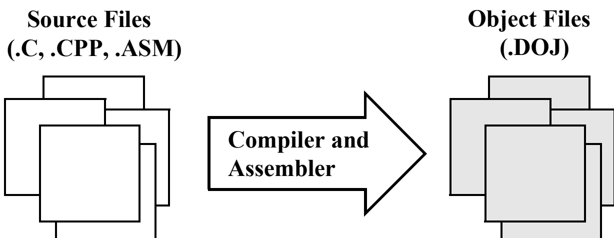


Figure 1-1. Compiling and Assembling

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 Chapter 3, "Linker Description File").

Input Section Directives in Assembly Code

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

Example (for Blackfin processors)

```
.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 1

While processing the following code, the compiler stores the temp variable in the ext_data input section of the .DOJ file 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 2

In the following example, `section ("extern")` is optional. Note the new function (`funct2`) does not require `section ("extern")`. For more information on LDF sections, refer to "Specifying the Memory Map" on page 2-18.

```
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++ 3.5 C/C++ Compiler and Library Manual* for appropriate target processors and "Placing Code on the Target" on page 2-26.

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, in some situations you may want to use non-default names. One such situation is 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).

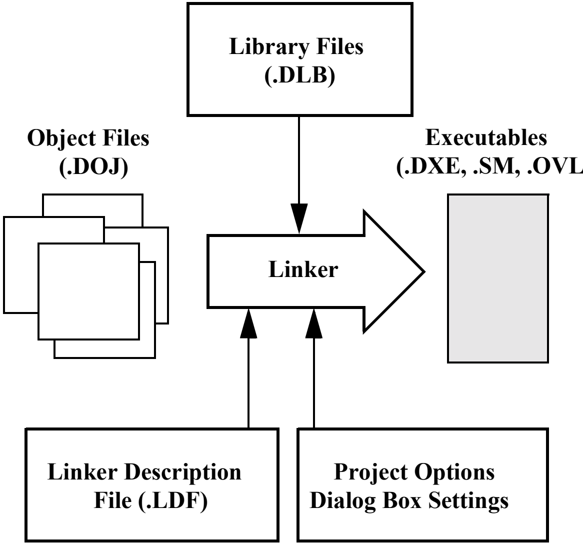


Figure 1-2. Linking Diagram

Linking enables your code to run efficiently in the target environment. Linking is described in detail in Chapter 2, "Linker".

When developing a new project, use the Expert Linker to generate the project's .LDF file. See Chapter 4, "Expert Linker".

Linker and Assembler Preprocessor

The linker and assembler preprocessor program (`pp.exe`) evaluates and processes preprocessor commands in source files. With these commands, you direct the preprocessor to define macros and symbolic constants, include header files, test for errors, and control conditional assembly and compilation.

The `pp` preprocessor is run by the assembler or linker from the operating system's command line or within the VisualDSP++ environment. These tools accept and pass this command information to the preprocessor. The preprocessor can also operate from the command line using its own command-line switches.

The preprocessor supports ANSI C standard preprocessing with extensions but differs from the ANSI C standard preprocessor in several ways. For more information on rhe `pp` preprocessor, see the *VisualDSP++ 3.5 Assembler & Preprocessor Manual* for appropriate target archtecture.

The compiler has it own preprocessor that allows you to use preprocessor commands within your C/C++ source. The compiler preprocessor automatically runs before the compiler. For more information, see the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual* for the target processors.

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. The file(s) may reside on another processor (host) or may be burned into a PROM.

The *VisualDSP++ 3.5 Loader Manual for 16-Bit Processors* provides detailed descriptions of the processes and options to generate boot-loadable .LDR (loader) files for the approprate target processors. This manual also describes splitting (when used), which creates non-bootloadable files that execute from the processor's external memory.

In general:

- The Blackfin processors use the loader (elfloader.exe) to yield a bootloadable image (.LDR file), which resides in memory external to the processor (PROM or host processor).
- The ADSP-218x loader/splitter (elfspl21.exe) is used to convert executable files into boot-loadable (or non-bootable) files for ADSP-218x DSPs.
- The ADSP-219x loader/splitter (elfloader.exe) is used to create bootstream, non-boot-stream, or combinational output for ADSP-219x DSPs (ADSP-2191/2195/2196 DSPs as well as ADSP-21990/21991/21992 DSPs).
- The ADSP-2192 loader utility (elfloader.exe) is used to convert executable files (.DXE) into a boot-loadable file (.H) for the ADSP-2192-12 DSP.

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).

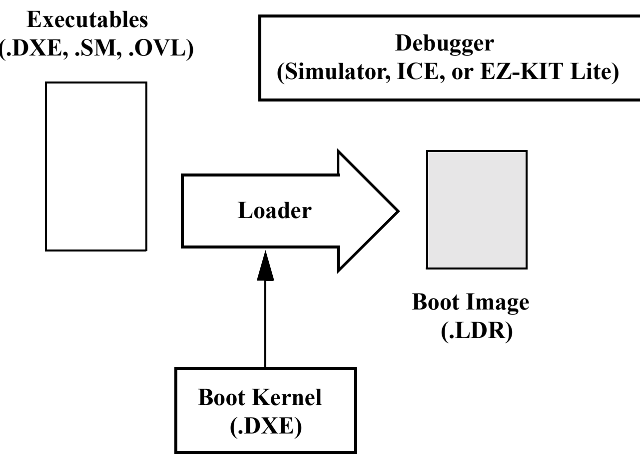


Figure 1-3. Loading Diagram

For example, when a Blackfin processor is reset, the boot kernel portion of the image is transferred to the processor's core. Then, the instruction and data portion of the image are loaded into the processor's internal RAM (as shown in Figure 1-4) by the boot kernel.

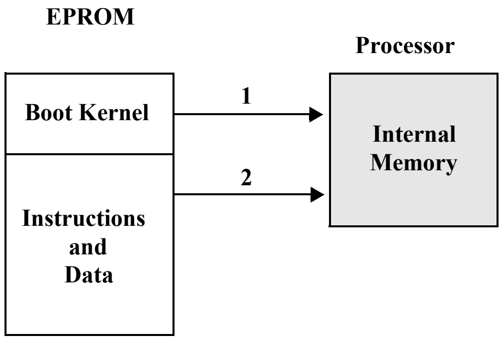


Figure 1-4. Booting from a Bootloadable (.LDR) File

VisualDSP++ includes boot kernel files (`.DXE`), which are automatically used when you run the loader. You can also customize boot kernel source files (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`). This example illustrate the generation of a loader file for a multiprocessor architecture.

The `.SM` and `.OVL` files must reside in the same directory that contains 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.

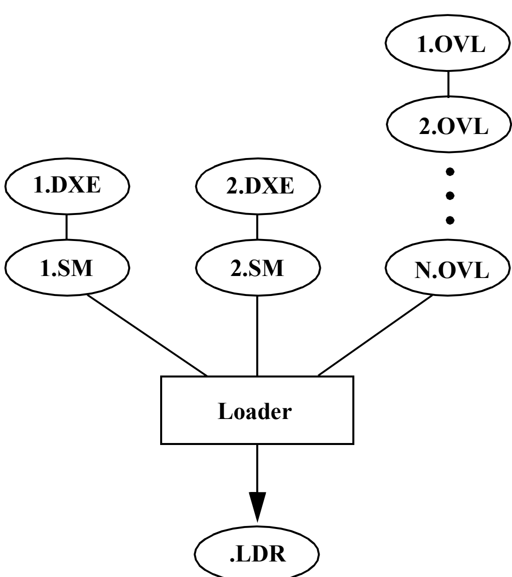


Figure 1-5. Input Files for a Multiprocessor System

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

2 LINKER

Linking assigns code and data to processor memory. For a simple single-processor architecture, a single `.DXE` file is generated. A single invocation of the linker may 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.

You can run the linker from a command line or from the VisualDSP++ Integrated Development and Debugging Environment (IDDE).

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

This chapter includes:

- "Linker Operation" on page 2-2
- "Linking Environment" on page 2-6
- "Link Target Description" on page 2-11
- "Passing Arguments for Simulation or Emulation: Blackfin Processors ONLY" on page 2-29
- "Linker Command-Line Reference" on page 2-30

Linker Operation

Figure 2-1 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.

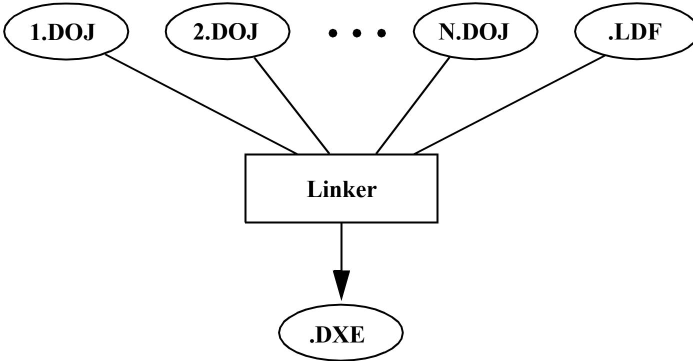


Figure 2-1. Linking Object Files to Produce an Executable File

When developing a new project, use the Expert Linker to generate the project's LDF. See Chapter 4, "Expert Linker" for more information.

In a multiprocessor system, a .DXE file for each processor is generated. For example, for a two-processor system, you must generate two .DXE files. The processors in a multiprocessor architecture may 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.

Overlay files (.OVL), another linker output, support applications that require more program instructions and data than the processor's internal memory can accommodate. Refer to "Memory Management Using Overlays" on page 5-4 for more information.

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

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) (`.DXE`) 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). Refer to "Linker Command-Line Reference" on page 2-30.
- Settings (options) on the **Link** page of the **Project Options** dialog box. See "Project Builds" on page 2-6.
- LDF commands. Refer to "LDF Commands" on page 3-23 for a detailed description of the 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 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 provides the information needed to link your code.

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 file. 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 one or more portions 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 — Overview

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, as described in Chapter 4, "Expert Linker".

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

Refer to Chapter 3, "Linker Description File" 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 (`.DXE`) file, or an image file (`.LDR` or others). Error and warning messages appear 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 2-2 shows the VisualDSP++ environment with the **Project** window and an `.LDF` file open in an editor window.

The VisualDSP++ IDDE provides an intuitive interface for processor programming. When you open 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.

For information about the VisualDSP++ environment, refer to the *VisualDSP++ User's Guide* for the appropriate target architecture or online Help. 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.

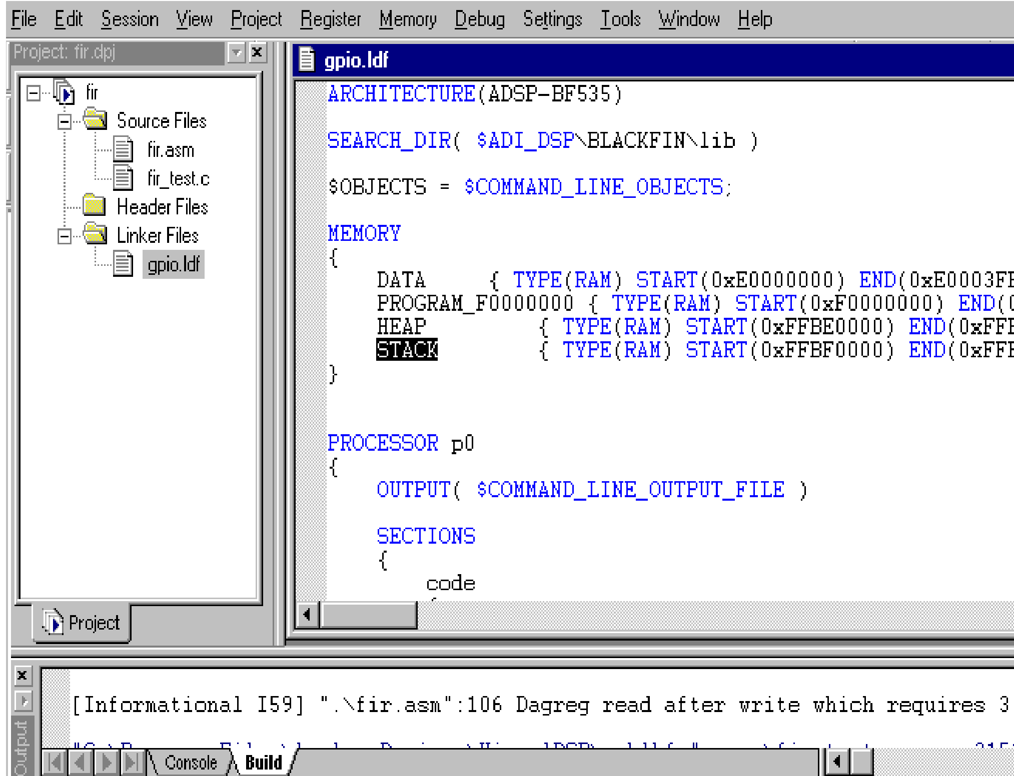


Figure 2-2. VisualDSP++ Environment

Within VisualDSP++, specify tool settings for project builds. Modify linker options via the **Link** tab of the **Project Options** dialog box (Figure 2-3). Choosing a **Category** from the pull-down list at the top of the **Link** tab presents different panes of options.

There are four sub-pages you can access—**General**, **LDF Preprocessing**, **Elimination**, and **Processor**. Almost every setting option has a corresponding compiler command-line switch described in "Linker Command-Line Switches" on page 2-34. The **Additional options** field in

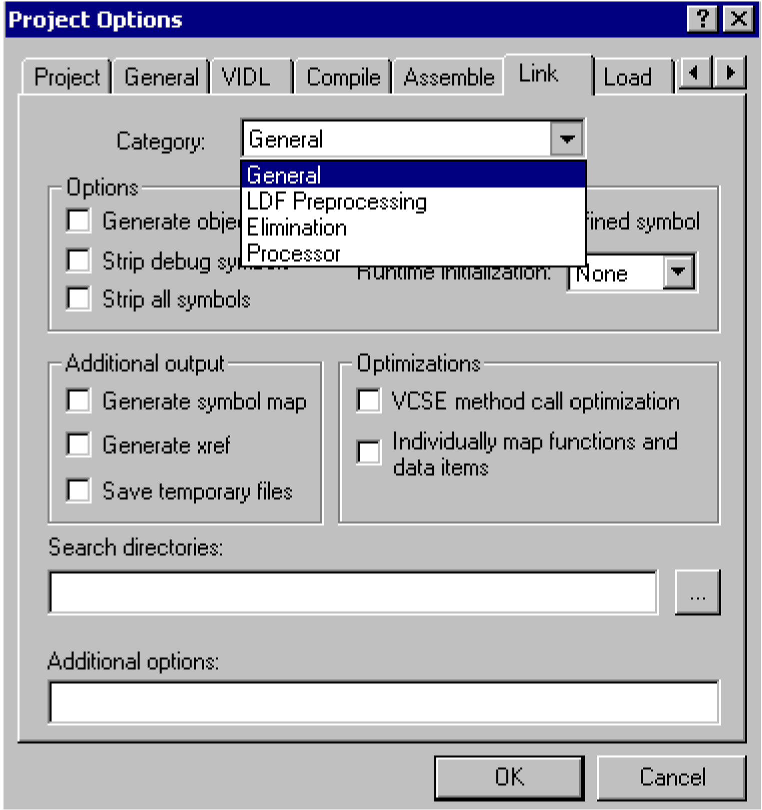


Figure 2-3. Main Link Tab with Category Selections

each sub-page is used to enter the appropriate file names and options that do not have corresponding controls on the **Link** sub-page but are available as compiler switches.

Due to different processor architectures, Blackfin processors, ADSP-218x DSPs and ADSP-219x DSPs provide different **Link** tab selection options. Use the VisualDSP++ context-sensitive online Help for each target architecture to select information on linker options you can specify in VisualDSP++.

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 (`.DXE`).

Figure 2-4 shows the Expert Linker window (for Blackfin processors), which comprises two panes: **Input Sections** and **Memory Map** (output sections). Refer to Chapter 4, "Expert Linker", for detailed information.

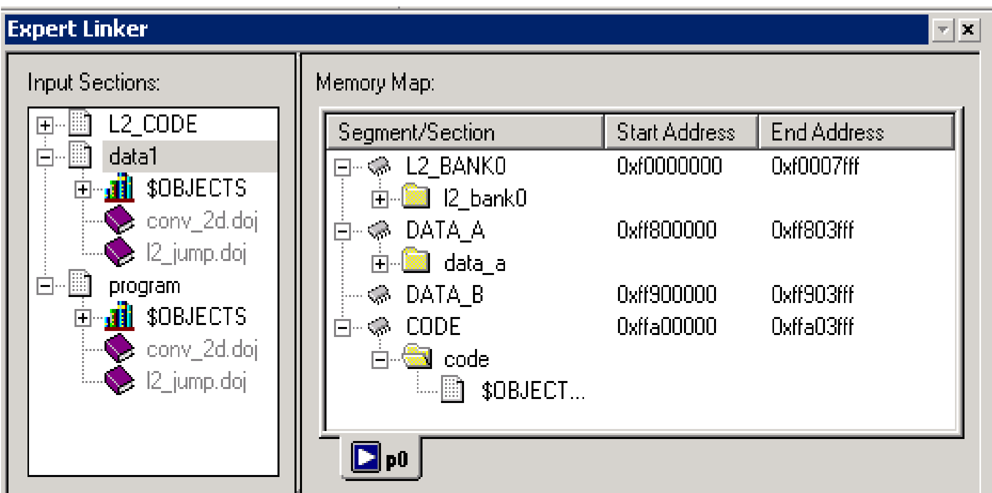


Figure 2-4. Expert Linker Window

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 unused symbols 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** tab displays project build status and error messages. In most cases, double-click a message displays the line in the source file causing the problem. You can access descriptions of linker messages from VisualDSP++ online Help by selecting a six-character code (for example, `li1002`) and pressing the **F1** key.

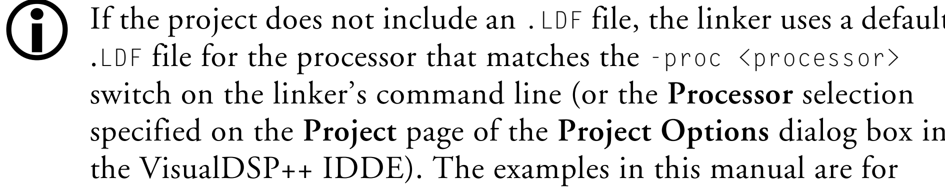
Some build errors, such as a reference to an undefined symbol, do not correlate directly to source files. These errors often stem from omissions in the `.LDF` file.

For example, if an input section from the object file is not placed by the `.LDF` file, a cross-reference error occurs at every object that refers to labels in the missing section. Fix this problem by reviewing the `.LDF` file and specifying all sections that need placement. For more information, refer to the *VisualDSP++ 3.5 User's Manual for 16-Bit Processors* or online Help.

Link Target Description

Before defining the system's memory and program placement with linker commands, analyze the target system to ensure 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 file for the processor that matches the -proc <processor> switch on the linker's command line (or the **Processor** selection specified on the **Project** page of the **Project Options** dialog box in the VisualDSP++ IDDE). 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 2-18.

- **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 3-29.

ADSP-BF535 Processor Memory Architecture Overview

As an example, this section describes the Blackfin ADSP-BF535 memory architecture and memory map organization.

Other processors in the Blackfin family (ADSP-BF531/2/3 and ADSP-561) have very different memory architectures. Refer to *Hardware Reference* manuals of target processors for appropriate information.

The ADSP-BF535 processor includes the L1 memory subsystem with a 16Kbyte instruction SRAM/cache, a dedicated 4Kbyte data scratchpad, and a 32Kbyte data SRAM/cache configured as two independent 16Kbyte 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.

Figure 2-5 shows the ADSP-BF535 system block diagram.

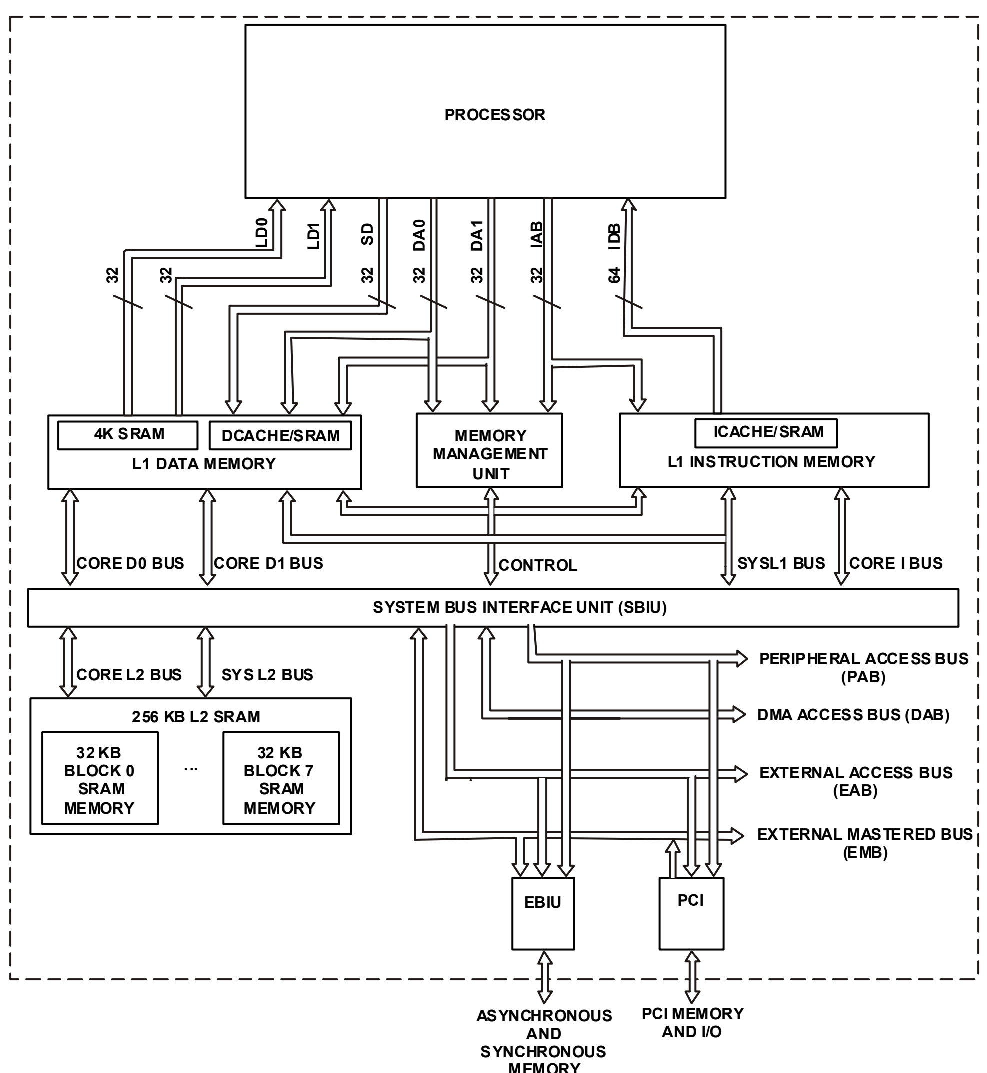


Figure 2-5. ADSP-BF535 System Block Diagram

The device has two ports to 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.

Memory ranges are listed in Table 2-1. Address ranges that are not listed are reserved.

Table 2-1. ADSP-BF535 Processor Memory Map Addresses

Memory Range	Range Description
`0xFFE00000 —0xFFFFFFFF`	Core MMR registers (2MB)
`0xFFC00000 —0xFFDFFFFF`	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)
`0xF0040000 —0xFF7FFFFF`	Reserved
`0xF0000000 —0xF003FFFF`	L2 Memory Bank SRAM (256K)
`0xEF000400 —0xEFFFFFFF`	Reserved
`0xEF000000 —0xEF0003FF`	Boot ROM (1K)
`0x00000000 —0xEEFFFFFF`	External memory

The `MEMORY` section in Listing 2-1 on page 2-24 assumes that only L1 and L2 SRAMs are available and that L1 is unused. Refer to the *VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors* and the appropriate *Hardware Reference* for information about cache configuration.

See the *Memory* chapter in an appropriate *Hardware Reference* for information about your target processor's memory organization.

ADSP-218x DSP Core Architecture Overview

Figure 2-6 shows the ADSP-218x DSP core architecture.

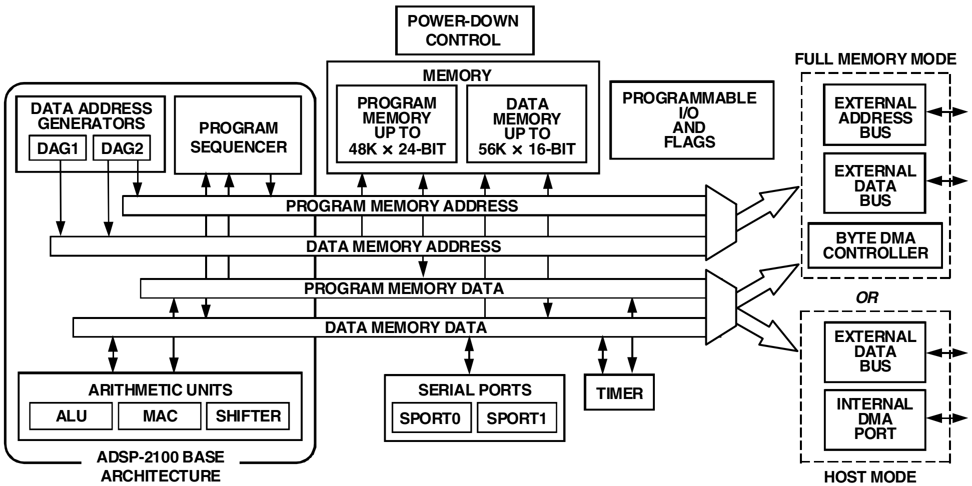


Figure 2-6. ADSP-218x DSP Functional Block Diagram

ADSP-218x DSPs use a modified Harvard architecture in which Data Memory stores data and Program Memory stores both instructions and data. All ADSP-218x processors contain on-chip RAM that comprises a portion of the Program Memory space and Data Memory space. (Program Memory and Data Memory are directly addressable off-chip.) The speed of the on-chip memory allows the processor to fetch two operands (one from Data Memory and one from Program Memory) and an instruction (from Program Memory) in a single cycle. In each ADSP-218x processor, five on-chip buses connect internal memory with the other functional units. A single external address bus (14 bits) and a single external data bus (24 bits) are extended off-chip; these buses can be used for either Program or Data Memory accesses.

All ADSP-218x DSPs (except for the ADSP-2181 and ADSP-2183 DSPs) can be configured in either a Host Mode or a Full Memory Mode. In Host Mode, each processor has an Internal DMA (IDMA) port for connection to external host systems. The IDMA port provides transparent, direct access to the DSP's on-chip Program and Data RAM. Since the ADSP-2181 and ADSP-2183 DSPs have complete address, data, and IDMA busses, these two processors provide both IDMA and BDMA functionality concurrently to provide greater system functionality without additional external logic.

In Full Memory Mode, ADSP-218x processors have complete use of the external address and data buses. In this mode, the processors behave as ADSP-2181 and ADSP-2183 processors with the IDMA port removed.

Program Memory (Full Memory Mode) is a 24-bit-wide space for storing instruction opcodes and data. The ADSP-218x DSPs have up to 48K words of Program Memory RAM on chip, and the capability of accessing up to two 8K external memory overlay spaces by means of the external data bus. Program Memory (Host Mode) allows access to all internal memory. External overlay access is limited by a single external address line (A0). External program execution is not available in the host mode because of a restricted data bus that is only 16 bits wide.

Data Memory (Full Memory Mode) is a 16-bit-wide space used for storing data variables and memory-mapped control registers. For example, ADSP-218xN DSPs have up to 56K words of Data Memory RAM on-chip. Part of this space is used by 32 memory-mapped registers. Support also exists for up to two 8K external memory overlay spaces through the external data bus. All internal accesses complete in one cycle. Data Memory (Host Mode) allows access to all internal memory. External overlay access is limited by a single external address line (A0).

The ADSP-218x processors support memory-mapped peripherals with programmable wait state generation through a dedicated 2048 location I/O Memory space.

ADSP-219x DSP Architecture Overview

Figure 2-7 shows the ADSP-219x DSP core architecture. The ADSP-219x architecture is code-compatible with ADSP-218x DSPs. However, the ADSP-219x architecture has several enhancements over the ADSP-218x architecture, including single or dual-core architecture, three computational units, two data address generators, a program sequencer, a JTAG port, a 24-bit address reach, and an instruction cache. These enhancements make ADSP-219x DSPs more flexible and easier to program.

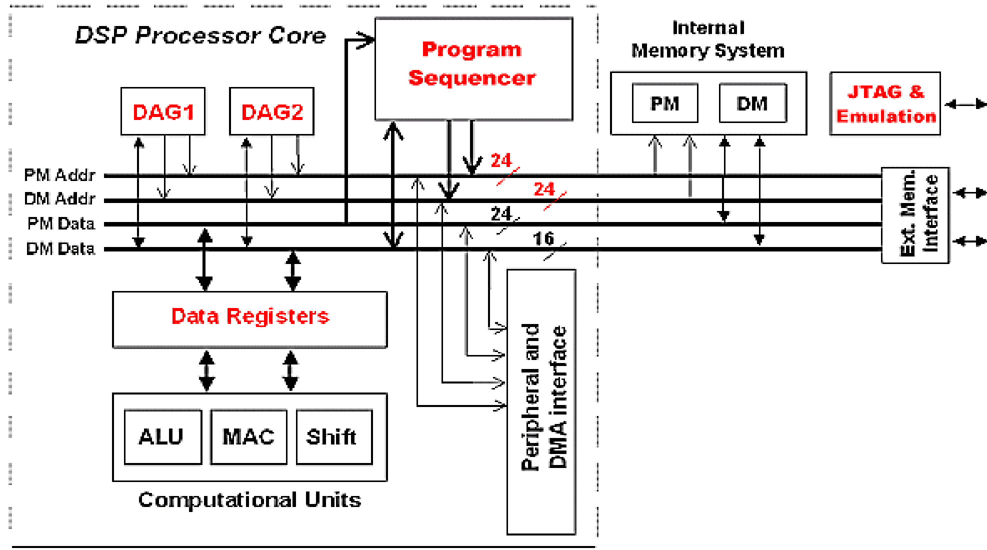


Figure 2-7. ADSP-219x DSP Basic Functional Block Diagram

For example, the ADSP-2191M DSP has a single-core architecture that integrates 64K words of on-chip memory configured as 32K words (24-bit) of program RAM, and 32K words (16-bit) of data RAM. Power-down circuitry is also provided to reduce power consumption.

The two address buses (PMA and DMA) share a single external address bus to allow memory to be expanded off-chip, and the two data buses (PMD and DMD) share a single external data bus. Boot memory space and I/O memory space also share the external buses. Program memory can store both instructions and data to enable the ADSP-2191M DSP to fetch two operands in a single cycle, one from program memory and one from data memory. The DSP's dual memory buses also let the ADSP-219x core fetch an operand from data memory and the next instruction from program memory in a single cycle.

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, as shown in "SECTIONS{}" on page 3-42.

The three steps involved in allocating memory are:

- "Memory Usage" on page 2-18
- "Memory Characteristics" on page 2-20
- "Linker MEMORY{} Command in .LDF File" on page 2-24

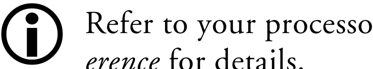
Memory Usage

Input section names are generated automatically by the compiler or are specified in the assembly source code. The `.LDF` file 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 tables below). The produced `.DXE` 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.

The following tables show how input sections, output sections, and memory segments correspond in the default LDFs for appropriate target processor architectures.

- Table 2-2 shows section mapping in the default `.LDF` file for ADSP-2191 DSPs (as an example for the ADSP-218x/9x DSPs)
- Table 2-3 shows section mapping in the default `.LDF` file for ADSP-BF535 processor (as an example for Blackfin processors)



Refer to your processor's default `.LDF` file and to the *Hardware Reference* for details.

Typical uses for memory segments include interrupt tables, initialization data, program code, data, heap space, and stack space.

Table 2-2. Section Mapping in the Default ADSP-2191 LDF

Input Section	Output Section	Memory Segment
program	program_dxe	mem_code
data1	data1_dxe	mem_data1
data2	data2_dxe	mem_data2
N/A	sec_stack	mem_stack
N/A	sec_heap	mem_heap

Table 2-3. Section Mapping in the Default 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

For Blackfin processors, you can modify your .LDF file to place objects into L1 memories when they are configured as SRAM.

Memory Characteristics

This section provides an overview of basic memory information (including addresses and ranges) for target architectures.

Blackfin Processors

Table 2-4 lists memory ranges for the ADSP-BF535 processors. Address ranges that are not listed are reserved. Blackfin processors have a 32-bit address range to support memory addresses from 0x0 to 0xFFFF FFFF. Figure 2-8 on page 2-22 shows the ADSP-BF535 processor memory architecture. Other Blackfin processors have different memory architectures. Refer to *Hardware References* of target processors for appropriate information.

Table 2-4. ADSP-BF535 Processor Memory Map Addresses

Memory Range	Range Description
0xFFE00000 —0xFFFFFFFF	Core MMR registers (2MB)
0xFFC00000 —0xFFDFFFFF	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)
0xF0040000 —0xFF7FFFFF	Reserved
0xF0000000 —0xF003FFFF	L2 Memory Bank SRAM (256K)
0xEF000400 —0xEFFFFFFF	Reserved
0xEF000000 —0xEF0003FF	Boot ROM (1K)
0x00000000 —0xEEFFFFFF	Unpopulated

The `MEMORY` section in Listing 2-1 on page 2-24 assumes that only L1 and L2 SRAMs are available and that L1 is unused. Refer to the *VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors* and the appropriate *Hardware Reference* for information about cache configuration.

ADSP-218x and ADSP-219x DSPs

ADSP-218x and ADSP-219x DSPs have a 24-bit address range to support memory addresses from `0x0` to `0xFFFFFF`. Memory type is defined by two characteristics: `PM` or `DM`, and `RAM` or `ROM`. Some portions of the DSP memory are reserved. Refer to your DSP's *Hardware Reference* for details.

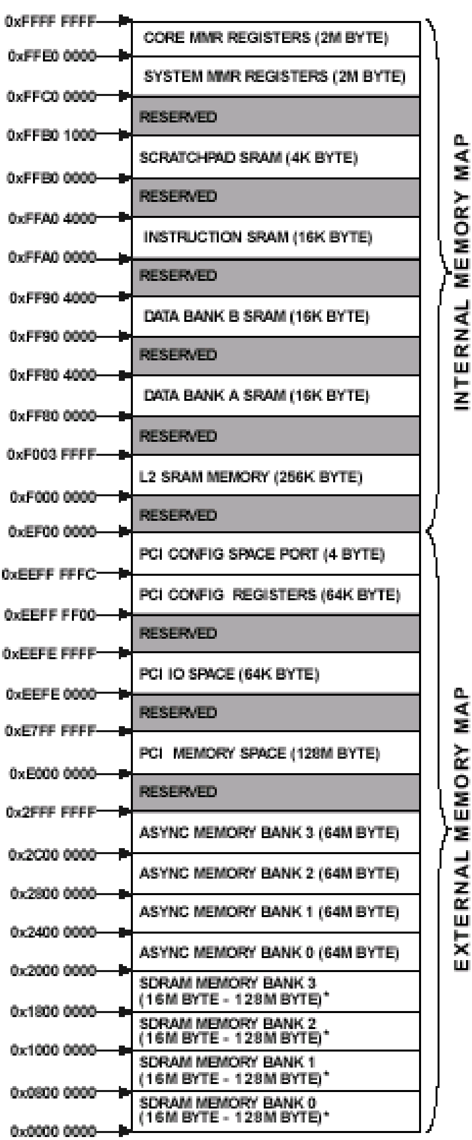


Figure 2-8. ADSP-BF535 Processor Memory Architecture

ADSP-218x DSPs provide a variety of memory and peripheral interface options. The key functional groups are Program Memory, Data Memory, Byte Memory, and I/O. Refer to Figure 2-9 for PM and DM memory allocations in the ADSP-2186 DSP.

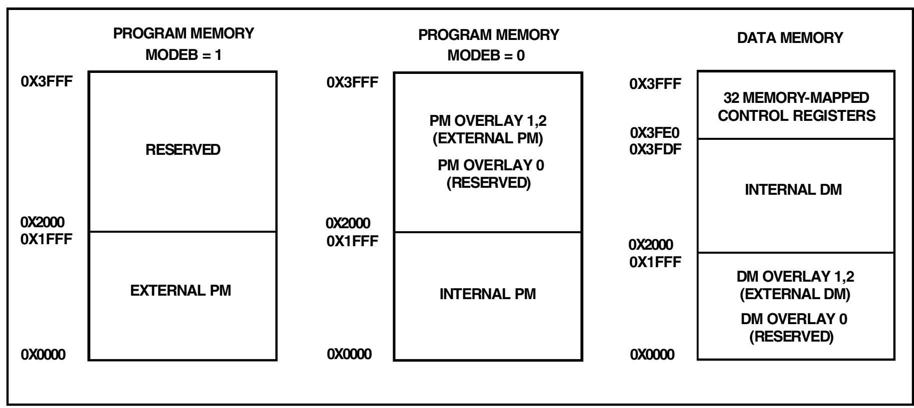


Figure 2-9. ADSP-2186 DSP Memory Architecture

ADSP-2191x DSPs provide various memory allocations. Refer to Figure 2-10 for PM and DM memory allocations in the ADSP-2191M DSP. It provides 64K words of on-chip SRAM memory. This memory is divided into four 16K blocks located on memory Page 0 in the DSP's memory map. In addition to addressing internal and external memory space, ADSP-2191M DSPs can address two additional and separate off-chip memory spaces: I/O space and boot space.

As shown, the DSP's two internal memory blocks populate all of Page 0. The entire DSP memory map consists of 256 pages (Pages 0–255), and each page is 64K words long. External memory space consists of four memory banks (banks 0–3) and supports a wide variety of SRAM memory devices. Each bank is selectable with the memory select pins (MS3–0) and has configurable page boundaries, wait states, and wait state modes. The 1K word of on-chip boot ROM populates the top of Page 255 while the

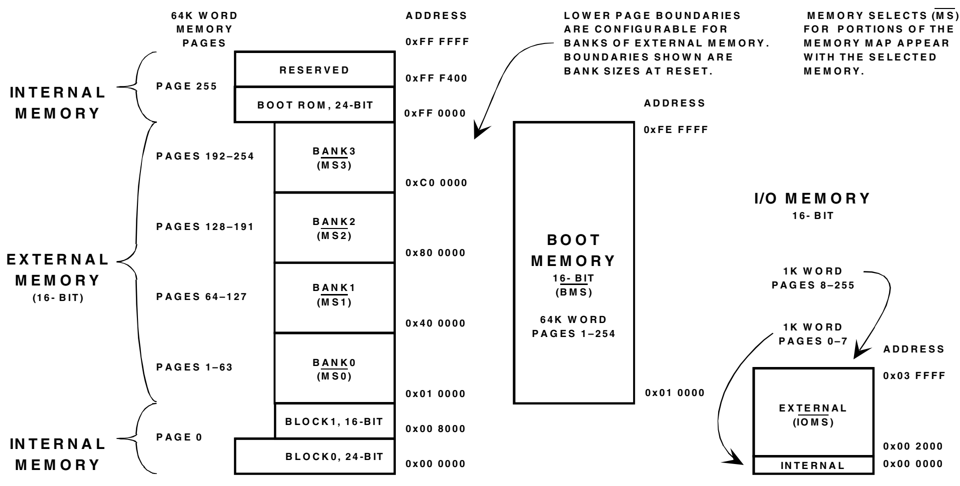


Figure 2-10. ADSP-2191M DSP Memory Architecture

remaining 254 pages are addressable off-chip. I/O memory pages differ from external memory pages in that I/O pages are 1K word long, and the external I/O pages have their own select pin (IOMS). Pages 0–7 of I/O memory space reside on-chip and contain the configuration registers for the peripherals. Both the core and DMA-capable peripherals can access the DSP's entire memory map.

Linker MEMORY{} Command in .LDF File

Referring to information in sections "Memory Usage" and "Memory Characteristics", you can specify the target's memory with the `MEMORY{}` command for any of the four target processor architectures (Listing 2-1).

Listing 2-1. Blackfin Processors -- MEMORY{} Command Code

```
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) }
}
```

Listing 2-2. ADSP-2191 DSPs -- MEMORY{} Command

```
MEMORY /* Define and label system memory  */
{      /* List of global memory segments  */
seg_rth   {TYPE(PM RAM) START(0x000000) END(0x000241) WIDTH(24)}
seg_code  {TYPE(PM RAM) START(0x000242) END(0x007fff) WIDTH(24)}
seg_data1 {TYPE(DM RAM) START(0x008000) END(0x00ffff) WIDTH(16)}
}
```

The above examples apply 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.

These example define 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.

To write a `SECTIONS{}` command:

1. List all input sections defined in the source files.
   - **Assembly files.** List each assembly code `.SECTION` directive, identify its memory type (`PM` or `CODE`, or `DM` or `DATA`), and note 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`" or "`code`" 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 2-3 presents a `SECTIONS{}` command that would work with the `MEMORY{}` command in Listing 2-1.

Listing 2-3. Blackfin SECTIONS{} Command in the .LDF File

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

      dxe_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 */
```

Listing 2-4. ADSP-218x/9x SECTIONS{} Command in the .LDF File

```
SECTIONS
   {         /* List of sections for processor P0 */
   sec_rth   {INPUT_SECTIONS ( $OBJECTS(rth))}     > seg_rth
   sec_code  {INPUT_SECTIONS ( $OBJECTS(code)}     > seg_code
   sec_code2 {INPUT_SECTIONS ( $OBJECTS(y_input)}  > seg_code
   sec_data1 {INPUT_SECTIONS ( $OBJECTS(data1))}   > seg_data1
   }
}
```

Passing Arguments for Simulation or Emulation: Blackfin Processors ONLY

To support simulation and emulation in Blackfin processors, 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 1 – Basic .LDF File for Blackfin Processors" on page 3-4).

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.

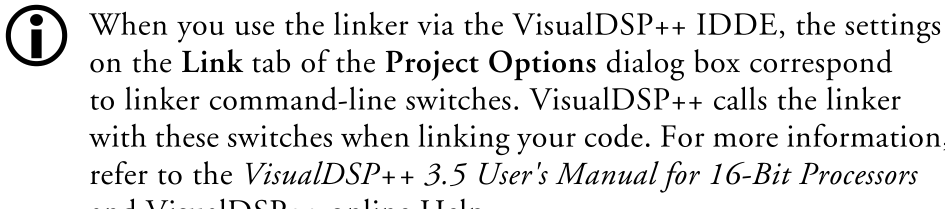
Refer to the *VisualDSP++ 3.5 User's Manual for 16-Bit Processors* or online Help for information about the simulator and command-line arguments.

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-Line Syntax" on page 2-30
- "Linker Command-Line Switches" on page 2-34



When you use the linker via the VisualDSP++ IDDE, the settings on the **Link** tab 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++ 3.5 User's Manual for 16-Bit Processors* and VisualDSP++ online Help.

Linker Command-Line Syntax

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

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

The linker command requires `-proc` *`processor`* or a `-T <ldf name>` for the link to proceed. If the command line does not include `-proc` *`processor`*, the `.LDF` file following the `-T` switch must contain a `-D`*`architecture`* 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 -proc ADSP-BF535 p0.doj -T target.ldf -t -o program.dxe
```

Use `-proc processor` instead of the deprecated `-Darchitecture` on the command line to select the target processor. See Table 2-6 on page 2-36 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" on page 2-31
- "Command-Line File Names" on page 2-32
- "Object File Types" on page 2-34

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 Chapter 6, "Archiver".
- An executable (`.DXE`) file to be linked against. Refer to `$COMMAND_LINE_LINK_AGAINST` in "Built-In LDF Macros" on page 3-21.

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
- The directory path may be an absolute path or a path relative 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 2-5 lists valid file extensions used by the linker.

Command-Line File Names

Some linker switches take a file name as a parameter. Table 2-5 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 2-5.

Table 2-5. File Extension Conventions

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

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` file.

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 specifying `-proc <processor>`), 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 3-21.

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 file names that contain spaces 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.

Object File Types

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 is given an `.XML` extension. An executable is written in the ELF format and is 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 3-21.

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

Linker Command-Line Switches

This section describes the linker's command-line switches. Table 2-6 on page 2-36 briefly describes each switch with regard to case sensitivity, equivalent switches, switches overridden or contradicted by the one described, and naming and spacing constraints for parameters.

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` file 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. Using no switches is the same as typing `linker -help`.

Linker Switch Summary

Table 2-6 briefly describes each linker switch. Each individual switch is described in detail following this table. See "Project Builds" on page 2-6 for information on the VisualDSP++ **Project Options** dialog box.

Table 2-6. Linker Command-Line Switches – Summary

Switch	Description	More Info
`@file`	Uses the specified file as input on the command line.	on page 2-38
`-DprocessorID`	Specifies the target processor ID. The use of `-proc processorID` is recommended.	on page 2-38
`-L path`	Adds the path name to search libraries for objects	on page 2-39
`-M`	Produces dependencies.	on page 2-39
`-MM`	Builds and produces dependencies.	on page 2-39
`-Map file`	Outputs a map of link symbol information to a file	on page 2-39
`-MDmacro[=def]`	Defines and assigns value `def` to a preprocessor macro.	on page 2-39
`-Ovcse`	Enables VCSE method call optimization	on page 2-40
`-S`	Omits debugging symbols from the output file.	on page 2-40
`-T filename`	Names the LDF	on page 2-40
`-Wwarn number`	Demotes the specified error message to a warning	on page 2-40
`-e`	Eliminates unused symbols from the executable.	on page 2-41
`-es secName`	Names input sections (`secName` list) to which elimination algorithm is being applied.	on page 2-41
`-ev`	Eliminates unused symbols verbosely.	on page 2-41
`-flag-meminit`	Passes each comma-separated option to the Meminit utility.	on page 2-42
`-flag-pp`	Passes each comma-separated option to the preprocessor.	on page 2-42

Table 2-6. Linker Command-Line Switches – Summary (Cont'd)

Switch	Description	More Info
`-h` `-help`	Outputs the list of command-line switches and exits.	on page 2-42
`-i` *path*	Includes search directory for preprocessor include files.	on page 2-42
`-ip`	Fills fragmented memory with individual data objects that fit and requires that objects have been assembled with the assembler's `-ip` switch. **Note:** ADSP-21xx DSPs only.	on page 2-42
`-jcs21`	Converts out-of-range short calls and jumps to the longer form. **Note:** Blackfin processors and ADSP-219x DSPs only.	on page 2-43
`-jcs21+`	Enables `-jcs21` and allows the linker to convert out-of-range branches to indirect calls and jumps sequences **Note:** Blackfin processors only.	on page 2-44
`-keep` *symName*	Retains unused symbols.	on page 2-44
`-meminit`	Cause post-processing of the executable file.	on page 2-44
`-o` *filename*	Outputs the named executable file.	on page 2-44
`-od` *filename*	Specifies the output directory.	on page 2-45
`-pp`	Stops after preprocessing.	on page 2-45
`-proc` *processor*	Selects a target processor.	on page 2-45
`-s`	Strips symbol information from the output file	on page 2-46
`-save-temps`	Saves temporary output files	on page 2-46
`-si-revision` *version*	Specifies silicon revision of the specified processor.	on page 2-46
`-sp`	Skips preprocessing.	on page 2-48
`-t`	Outputs the names of link objects.	on page 2-48
`-v` `-verbose`	Verbose——Outputs status information.	on page 2-48

Table 2-6. Linker Command-Line Switches – Summary (Cont'd)

Switch	Description	More Info
`-version`	Outputs version information and exits.	on page 2-48
`-warnonce`	Warns only once for each undefined symbol.	on page 2-48
`-xref` *`filename`*	Produces a cross reference (`ProjectName.xrf` file)	on page 2-49

The following sections provide the detailed descriptions of the linker's command-line switches.

@filename

Uses *`filemname`* 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.

-Dprocessor

The `-D`*`processor`* (define processor) switch specifies the target processor (architecture); for example, `-DADSP-BF535` or `-DADSP-2191`.

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

White space is not permitted between `-D` and *`processor`*. 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.

-L *path*

The `-Lpath` (search directory) switch 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 `SEARCH_DIR{}` command.

-M

The `-M` (generate make rule only) switch directs the linker to check a dependency and to output the result to `stdout`.

-MM

The `-MM` (generate make rule and build) switch directs the linker to output a rule, which is suitable for the make utility, describing the dependencies of the source file. The linker check for a dependency, outputs the result to `stdout`, and performs the build. The only difference between `-MM` and `-M` actions is that the linking continues with `-MM`. See "-M" for more information.

-Map *filename*

The `-map filename` (generate a memory map) switch directs the linker to output a memory map of all symbols. The map file name corresponds to the *filename* argument. For example, if the file name argument is `test`, the map file name is `test.xml`. The`.xml` extension is added where necessary.

-MD*macro*[=*def*]

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

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

-Ovcse

The -Ovcse (VCSE optimization) switch directs the linker to optimize VCSE method calls.

-S

The -S (strip debug symbol) switch directs the linker to omit debugging symbol information (*not* all symbol information) from the output file. Compare this switch with the -s switch on page 2-46.

-T *filename*

The -T *filename* (linker description file) switch directs the linker to use *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 -proc <*processor*>. The linker requires the -T switch when linking for a processor for which no VisualDSP++ support has been installed. In such cases, 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). White space must appear 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.

-Wwarn [number]

The -Wwarn (override error message) switch directs the linker to demote the specified error message to a warning. The *number* argument specifies the message to demote.

-e

The `-e` (eliminate unused symbols) switch directs the linker to eliminate unused symbols from the executable.

In order for the C and C++ run-time libraries to work properly, the following symbols should be retained with “KEEP()” (described on page 3-27):
`___ctor_NULL_marker` and `___lib_end_of_heap_descriptions`

-es *sectionName*

The `-es` *`sectionName`* (eliminate listed section) switch specifies a section to which the elimination algorithm is to be applied. This switch restricts elimination to the named input sections. The `-es` switch may be used on a command line more than once. Both this switch and the `ELIMINATE_SECTIONS()` LDF command (see on page 3-26) may be used to specify sections from which unreferenced code and data are to be eliminated.

In order for the C and C++ run-time libraries to work properly, the following symbols should be retained with “KEEP()” (described on page 3-27):
`___ctor_NULL_marker` and `___lib_end_of_heap_descriptions`

-ev

The `-ev` switch directs the linker to eliminate unused symbols and verbose, and provides reports on each eliminated symbol.

-flags-meminit -opt1[,-opt2...

The `-flags-meminit` switch passes each comma-separated option to the MemInit (Memory Initializer) utility.

-flags-pp -opt1[,-opt2...]

The `-flags-pp` switch passes each comma-separated option to the preprocessor.

Use `-flags-pp` with caution. For example, if the `pp` legacy comment syntax is enabled, the comment characters become unavailable for non-comment syntax.

-h[elp]

The `-h` or `-help` switch directs the assembler to output to `<stdout>` a list of command-line switches with a syntax summary.

-i|I *directory*

The `-idirectory` or `-Idirectory` (include directory) switch directs the linker to append the specified directory or a list of directories separated by semicolons (;) to the search path for included files.

-ip

ADSP-21xx DSPs only

The `-ip` (individual placement) switch directs the linker to fill in fragmented memory with individual data objects that fit. When the `-ip` switch is specified on the linker's command line or via the VisualDSP++ IDDE, the default behavior of the linker—placing data blocks in consecutive memory addresses—is overridden. The `-ip` switch allows individual placement of a grouping of data in DSP memory to provide more efficient memory packing.

The `-ip` switch works only with objects assembled using the assembler's `-ip` switch.

Absolute placements take precedence over data/program section placements in contiguous memory locations. When remaining memory space is not sufficient for the entire section placement, the link fails. The `-ip` switch allows the linker to extract a block of data for individual placement and fill in fragmented memory spaces.

The assembler's `-noip` option turns off individual placement option. See the *VDSP++ 3.5 Assembler and Preprocessor Manual* for target processors.

-jcs2l

Blackfin processors and ADSP-219x DSPs only

The `-jcs2l` switch directs the linker to convert out-of-range short calls and jumps to the longer or indirect form. Refer to **Branch expansion instruction** on the **Link** page. Any jump/call is subject to expansion to *`indirect`* if the linker is invoked with the `-jcs2l` switch (default for C programs).

The following table shows how the Blackfin linker handles jump/call conversions.

Instruction	Without -jcs2l	With -jcs2l
`JUMP.S`	short	short
`JUMP`	short or long	short or long
`JUMP.L`	long	long
`JUMP.X`	short or long	short, long or indirect
`CALL`	CALL	CALL
`CALL.X`	CALL	CALL or indirect

Refer to the *Instruction Set Reference* for target architecture for more information on jump and call instructions.

-jcs2l+

Blackfin processors only.
This is a deprecated switch equivalent to the -jcs2l switch.

The `-jcs2l+` switch enables the `-jcs2l` switch and allows the linker to convert out-of-range branches (`0x800000` to `0x7FFFFF`) 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*

The `-keep` *`sectionName`* (keep unused symbols) switch directs the linker to retain unused symbols. It directs the linker (when `-e` or `-ev` is enabled) to retain listed symbols in the executable even if they are unused.

-meminit

The `-meminit` (post-processing executable file) switch directs the linker to post-process the `.DXE` file through the MemInit (Memory Initializer) utility. This will cause the sections specified in the `.LDF` file to be "run-time" initialized by the C run-time library. By default, if this flag is not specified, all sections are initialized at "load" time (for example, via the VisualDSP++ IDDE or the boot loader).

-o *filename*

The `-o` *`filename`* (output file) switch directs the linker to output 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 `OUTPUT()` command in the `.LDF` file to name the output file.

-od *directory*

The `-od directory` switch directs the linker to specify the value of the `$COMMAND_LINE_OUTPUT_DIRECTORY LDF` macro. This switch 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 3-21.

-pp

The `-pp` (end after preprocessing) switch directs the linker to stop after the preprocessor runs without linking. The output (preprocessed LDF) prints to standard output.

-proc processor

The `-proc processor` (target processor) switch specifies that the linker should produce code suitable for the specified processor.

For example,

```
linker -proc ADSP-BF535 p0.doj p1.doj p2.doj -o program.dxe
```

If the processor identifier is unknown to the linker, it attempts to read required switches for code generation from the file `<processor>.ini`. The linker searches for the `.ini` file in the VisualDSP ++ System folder. For custom processors, the linker searches the section "`proc`" in the `<processor>.ini` for key "`architecture`". The custom processor must be based on an architecture key that is one of the known processors. Therefore, `-proc Custom-xxx` searches the `Custom-xxx.ini` file. For example,

```
[proc]
Architecture: ADSP-BF535
```

See also "-si-revision version" on page 2-46 for more information on silicon revision of the specified processor.

-s

The `-s` (strips all symbols) switch directs the linker to omit 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 file. This switch removes these symbols.

-save-temps

The `-save-temps` switch directs the linker to save temporary (intermediate) output files and place them in the **/temp** directory.

-si-revision version

The `-si-revision` *`version`* (silicon revision) switch directs the linker to provides a silicon revision of the specified processor. For example,

```
linker -proc ADSP- BF535 -si-revision 0.1
```

The parameter *`version`* represents a silicon revision of the processor specified by the `-proc` switch (on page 2-45).

The revision version takes one of two forms:

- One or more decimal digits, followed by a point, followed by one or two decimal digits. Examples of revisions are: 0.0; 0.1; 1.12; 23.1. Version 0.1 is distinct from and "lower" than version 0.10. The digits to the left of the point specify the chip tapeout number; the digits to the right of the point identify the metal mask revision number. The number to the right of the point cannot exceed decimal 255.
- A version value of *`none`* is also supported to indicate that the linker should not concern itself with silicon errata.

The `-si-revision` switch without a valid version value—that is, `-si-revision` alone or with an invalid parameter—generates an error.

This switch enables the linker to:

- Generate a warning about any “potential” anomalous conditions
- Generate errors if any anomalous conditions are detected

In the absence of silicon revision, the linker selects the largest silicon revision it “knows” about, if any.

The linker defines a macro, `__SILICON_REVISION__`, prior to preprocessing. The value assigned to this macro corresponds to the chip tapeout number converted to hexadecimal value and shifted left eight bits plus the metal mask revision number. Thus, revision 0.0 is `0x0`, 0.1 is `0x1`, 1.0 is `0x100`, and 10.21 is `0xa15`, etc. If the silicon revision is specified as “`none`”, the macro is not defined.

When the silicon revision number specified is greater than the largest number known to the linker, it will perform revision processing for the greatest known revision, and then emits a warning that it is defaulting to the earlier revision.

When a linker has no embedded support for silicon revisions of a processor, no warning is generated when the silicon revision is specified. When no silicon revision is specified, no warning is generated and the `__SILICON_REVISION__` macro is not set.

A linker “passes along” the appropriate `-si-revision` switch setting when invoking another VisualDSP++ tool; for example, when the linker invokes the assembler to process PLITs. When no switch was specified, the invoking tool passes no switch parameters. When the input is larger than the latest known parameter, the linker passes along the input value. These pass-through rules apply to all situations in which one tool that accepts this switch invokes another that also accepts the switch.

Example:

The Blackfin linker invoked as

```
linker -proc ADSP-BF535 -si-revision 0.1 …
```

invokes the assembler with

```
easmbkfn -proc ADSP-BF535 -si-revision 0.1
```

-sp

The `-sp` (skip preprocessing) switch directs the linker to link without preprocessing the `.LDF` file.

-t

The `-t` (trace) switch directs the linker to output the names of link objects to standard output as the linker processes them.

-v[erbose]

The `-v` or `-verbose` (verbose) switch directs the linker to display version and command-line information for each phase of linking.

-version

The `-version` (display version) switch directs the linker to display version information for the linker and preprocessor programs.

-warnonce

The `-warnonce` (single symbol warning) switch directs the linker to warn only once for each undefined symbol, rather than once for each reference to that symbol.

-xref *filename*

The `-xref` *`filename`* (external reference file) switch directs the linker to produce a cross-reference file (`ProjectName.xrf` file).

3 LINKER DESCRIPTION FILE

Every DSP project requires one Linker Description File (.LDF). The .LDF file specifies precisely how to link projects. Chapter 2, "Linker", describes the linking process and how the .LDF file ties into the linking process.

When generating a new .LDF file, use the Expert Linker to generate an .LDF file. Refer to Chapter 4, "Expert Linker" for details.

The .LDF file allows development of code for any processor system. It defines your system to the linker and specifies how the linker creates executable code for your system. This chapter describes .LDF file syntax, structure and components. Refer to Appendix C, "LDF Programming Examples for Blackfin Processors" and Appendix D, "LDF Programming Examples for ADSP-21xx DSPs" for the LDF examples for typical systems.

This chapter contains:

- "LDF File Overview" on page 3-3
- "LDF Structure" on page 3-11
- "LDF Expressions" on page 3-13
- "LDF Keywords, Commands, and Operators" on page 3-14
- "LDF Operators" on page 3-16
- "LDF Macros" on page 3-20
- "LDF Commands" on page 3-23

The linker runs the preprocessor on the `.LDF` file, so you can use preprocessor commands (such as `#defines`) within the file. For information about preprocessor commands, refer to a *VisualDSP++ 3.5 Assembler and Preprocessor Manual* for an appropriate target processor architecture.

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

Refer to example DSP programs shipped with VisualDSP++ for sample `.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), and assigns an address to every symbol, where:

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

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

You can use an .LDF file written from scratch. However, modifying an existing LDF (or a default .LDF file) is often the easier alternative when there are no large changes in your system's hardware or software. See "Example 1 – Basic .LDF File for Blackfin Processors", "Example 2 - Basic .LDF File for ADSP-218/9x DSPs", and "Notes on Basic .LDF File Examples"for basic information on LDF structure.

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

The linker may output warning messages and error messages. You must resolve the error messages to enable the linker to produce valid output. See "Linker Warning and Error Messages" on page 2-10 for more information.

Example 1 – Basic .LDF File for Blackfin Processors

Listing 3-1 is an example of a basic .LDF file for ADSP-BF535 processors (formatted for readability). Note the MEMORY{} and SECTIONS{} commands and refer to "Notes on Basic .LDF File Examples". Other .LDF file examples are provided in "LDF Programming Examples for Blackfin Processors" and "LDF Programming Examples for ADSP-21xx DSPs".

Listing 3-1. Example .LDF File for ADSP-BF535 Processor

```
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_SYSSTACK
        { 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 */

      dxe_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 */
```

Example 2 - Basic .LDF File for ADSP-218/9x DSPs

Listing 3-1 is an example of a basic `.LDF` file for ADSP-2191 DSPs (formatted for readability). Note the MEMORY{} and SECTIONS{} commands and refer to "Notes on Basic .LDF File Examples" on page 3-7. Other examples for assembly and C source files are in "LDF Programming Examples for Blackfin Processors" and "LDF Programming Examples for ADSP-21xx DSPs".

Listing 3-2. Example .LDF File for ADSP-2191 DSP

```
ARCHITECTURE(ADSP-2191)
SEARCH_DIR($ADI_DSP\219x\lib)
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY      /* Define and label system memory  */
{           /* List of global memory segments  */
seg_rth     {TYPE(PM RAM) START(0x000000) END(0x000241) WIDTH(24)}
seg_code    {TYPE(PM RAM) START(0x000242) END(0x007fff) WIDTH(24)}
seg_data1   {TYPE(DM RAM) START(0x008000) END(0x00ffff) WIDTH(16)}
}
PROCESSOR p0   /* the first (only?) processor in the system */
{
LINK_AGAINST ( $COMMAND_LINE_LINK_AGAINST )
OUTPUT ( $COMMAND_LINE_OUTPUT_FILE )
SECTIONS
   {         /* List of sections for processor P0 */
   sec_rth    {INPUT_SECTIONS ( $OBJECTS(rth))}      > seg_rth
   sec_code   {INPUT_SECTIONS ( $OBJECTS(code)}      > seg_code
   sec_code2  {INPUT_SECTIONS ( $OBJECTS(y_input)}   > seg_code
   sec_data1  {INPUT_SECTIONS ( $OBJECTS(data1))}    > seg_data1
   }
}
```

Notes on Basic .LDF File Examples

In the following description, the `MEMORY{}` and `SECTIONS{}` commands connect the program to the target DSP. For complete syntax information on LDF commands, see "LDF Commands" on page 3-23.

These notes describe features of the `.LDF` file presented in Listing 3-1.

- `ARCHITECTURE(ADSP-BF535)` specifies the target architecture (processor). This architecture 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 (on page 3-41). This example's argument (`$ADI_DSP\Blackfin\lib`) specifies one search directory.

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

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

- `$COMMAND_LINE_OBJECTS` (on page 3-21) is an LDF *command-line macro*, which expands at the linker command line into the list of input files. Each linker invocation from the VisualDSP++ IDDE has a command-line equivalent. In the VisualDSP++ IDDE, `$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)`.

  The VisualDSP++ IDDE generates 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.

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

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

- The `MEMORY{}` command (on page 3-29) 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 sec-

tion 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 3-39) generates a single executable file.
- The OUTPUT() command (on page 3-39) 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 3-21). The linker names the executable file 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++ IDDE).

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

  SECTIONS{} (on page 3-42) 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 3-1, two input sections (program and data1) are mapped into one memory segment (L2), as shown below.

```
    dxe_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 the output section 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 and define 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 command specifies system resources available as shared resources in a multi-processor environment.

  Declare common (shared) memory definitions in the global scope before the PROCESSOR{} commands. See "Command Scoping" 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 2-11
- "Placing Code on the Target" on page 2-26
- Appendix C, "LDF Programming Examples for Blackfin Processors"
- Appendix D, "LDF Programming Examples for ADSP-21xx DSPs"

Command Scoping

The two LDF scopes are *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 3-20).

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 3-1 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 is restricted to those scopes.

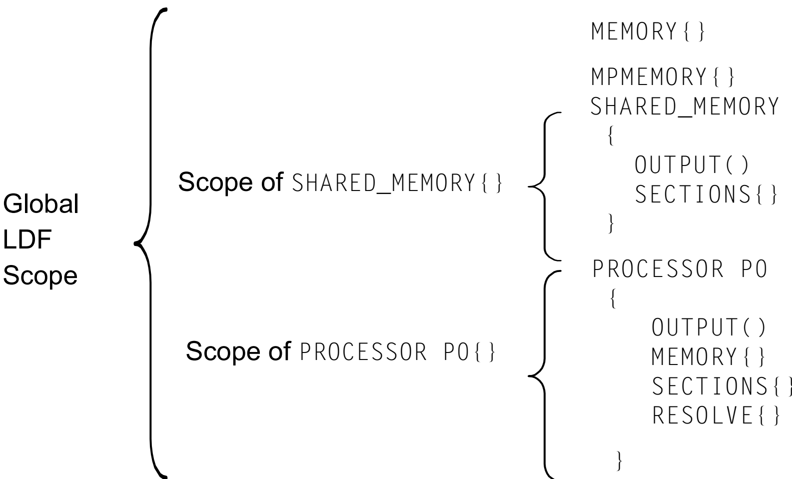


Figure 3-1. LDF Command Scoping Example

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 delimited by white space
    - Do not conflict with any keywords
    - Are unique

Table 3-1. Valid Items in Expressions

Convention	Description
.	Current location counter (a period character in an address expression). See "Location Counter (.)" on page 3-19.
`0x`*`number`*	Hexadecimal number (a 0x prefix)
`number`	Decimal number (a number without a prefix)
number`k` or *number*`K`	A decimal number multiplied by 1024
`B#`number or `b#`number	A binary number

LDF Keywords, Commands, and Operators

Table 3-2 lists .LDF file keywords. Descriptions of LDF keywords, operators, macros, and commands are provided in the following sections.

- "Miscellaneous LDF Keywords" on page 3-15
- "LDF Operators" on page 3-16
- "LDF Macros" on page 3-20
- "LDF Commands" on page 3-23

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

Table 3-2. LDF File Keywords Summary

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BM[1]	BOOT
DEFINED	DM[2]	ELIMINATE
ELIMINATE_SECTIONS	END	FALSE
FILL	FIRST_FIT	INCLUDE
INPUT_SECTION_ALIGN	INPUT_SECTIONS	KEEP
LENGTH	LINK_AGAINST	MAP
MEMORY	MEMORY_SIZEOF	MPMEMORY
NUMBER_OF_OVERLAYS	OUTPUT	OVERLAY_GROUP
OVERLAY_ID	OVERLAY_INPUT	OVERLAY_OUTPUT
PACKING	PAGE_INPUT[2]	PAGE_OUTPUT[2]

Table 3-2. LDF File Keywords Summary (Cont'd)

PLIT	PLIT_SYMBOL_ADDRESS	
PLIT_SYMBOL_OVERLAYID	PM[2]	PROCESSOR
RAM	RESOLVE	RESOLVE_LOCALLY
ROM	SEARCH_DIR	SECTIONS
SHARED_MEMORY	SHT_NOBITS	SIZE
SIZEOF	START	TYPE
VERBOSE	WIDTH	XREF

1 Supported on ADSP-218x DSPs only.
2 These keywords apply only to ADSP-218x/9x LDFs.

Miscellaneous LDF Keywords

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

Table 3-3. 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 2-49.

For more information about other .LDF file keywords, see "LDF Operators" on page 3-16, "LDF Macros" on page 3-20, and "LDF Commands" on page 3-23.

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 3-19).

  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,

```
dxe_L2_code
{
  INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
}> mem_L2

dxe_L2_data
{
  INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
}> mem_L2
```

the .LDF file may contain the command:

```
ldf_start_L2 = ADDR(dxe_L2_code)
```

The linker generates the constant `ldf_start_L2` and assigns it the start address of the `dxe_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.

```
test_present = DEFINED(test);
```

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 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 and leaves 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 simply declare them 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` file, regardless of where they are defined. For more information, see "Command Scoping" on page 3-12 and "LDF Macros and Command-Line Interaction" on page 3-22.

LDF macros are independent of preprocessor macro support, which is also available in the `.LDF` file. 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++ 3.5 Compiler and Library Manual* and the *VisualDSP++ 3.5 Assembler and Preprocessor Manual* for appropriate target processors for more information.

Built-In LDF Macros

The linker provides the following built-in 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. 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++ **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 of 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 LDF.

Base your decision on 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 3-12.

The linker supports these LDF commands (not all commands are used with specific processors):

- "ALIGN()" on page 3-24
- "ARCHITECTURE()" on page 3-24
- "ELIMINATE()" on page 3-25
- "ELIMINATE_SECTIONS()" on page 3-26
- "INCLUDE()" on page 3-26
- "INPUT_SECTION_ALIGN()" on page 3-26
- "KEEP()" on page 3-27
- "LINK_AGAINST()" on page 3-28
- "MEMORY{}" on page 3-29
- "MPMEMORY{}" on page 3-32
- "OVERLAY_GROUP{}" on page 3-33
- "PACKING()" on page 3-33
- "PAGE_INPUT()" on page 3-37
- "PAGE_OUTPUT()" on page 3-38
- "PLIT{}" on page 3-38

- "PROCESSOR{}" on page 3-39
- "RESOLVE()" on page 3-40
- "SEARCH_DIR()" on page 3-41
- "SECTIONS{}" on page 3-42
- "SHARED_MEMORY{}" on page 3-48

ALIGN()

The `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(processor)
```

The `ARCHITECTURE()` command is case sensitive. For example, valid entries may be `ADSP-BF535`, `ADSP-2189`, `ADSP-21990`, etc. 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` *`processor`* ...). 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` *`processor`*

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

ELIMINATE()

The `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 `-e` command-line switch (see on page 2-41).

When using either the linker's data elimination feature (via the Expert Linker or command-line switches) or the `ELIMINATE()` command in an `.LDF` file, it is essential that certain objects are kept using the `KEEP()` command so that the C/C++ run-time libraries function properly. The safest way to do this is to copy the `KEEP()` command from the default `.LDF` file into your own `.LDF` file.

For the C and C++ run-time libraries to work properly, the following symbols should be retained with "KEEP()" (see on page 3-27):
`___ctor_NULL_marker` and `___lib_end_of_heap_descriptions`

ELIMINATE_SECTIONS()

The `ELIMINATE_SECTIONS(sectionList)` command instructs the linker to remove unreferenced code and data from listed sections only.

The *sectionList* is a comma-delimited list of input sections. Both this LDF command and the linker's `-es` command-line switch (see on page 2-41) may be used to specify sections from which unreferenced code and data are to be eliminated.

INCLUDE()

The `INCLUDE()` command specifies additional `.LDF` files that 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.

Only one of these additional `.LDF` files is obligated to specify a target architecture. Normally, the top-level `.LDF` file includes the other `.LDF` files.

INPUT_SECTION_ALIGN()

The `INPUT_SECTION_ALIGN(number)` command aligns each input section (data or instruction) in an output section to an address satisfying *number*. The *number* argument, 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 3-42.

The INPUT_SECTION_ALIGN() command is valid only within the scope of an output section. For more information, see "Command Scoping" on page 3-12. For more information on output sections, see the syntax description for "SECTIONS{}" on page 3-42.

Example

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

When utilizing the linker's data elimination capabilities, it is essential that certain objects are kept using the `KEEP()` command so that the C/C++ run-time libraries function properly. The safest way to do this is to copy the `KEEP()` command from the default `.LDF` file into your own `.LDF` file.

For the C and C++ run-time libraries to work properly, the following symbols should be retained with "KEEP()" (see on page 3-27): `___ctor_NULL_marker` and `___lib_end_of_heap_descriptions`. A symbol specified in *`keepList`* must be a global symbol.

LINK_AGAINST()

The `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 the `PROCESSOR{}` and `SHARE_MEMORY{}` 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 3-40), which directs the linker to use the specified resolver, thus ignoring LINK_AGAINST() for a specific symbol. LINK_AGAINST() for other symbols still applies.

MAP()

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

The MAP(*filename*) command corresponds to and may be overridden by the linker's -Map <*filename*> command-line switch (see on page 2-39). In VisualDSP++, if a project's options (**Link** tab of **Project Options** dialog box) specify the generation of a symbol map, the linker runs with -Map <*projectname*>.xml asserted and the LDF's MAP() command generates a warning.

MEMORY{}

The 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 3-12.

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 3-2, followed by a description of each part of a *`segment declaration`*.

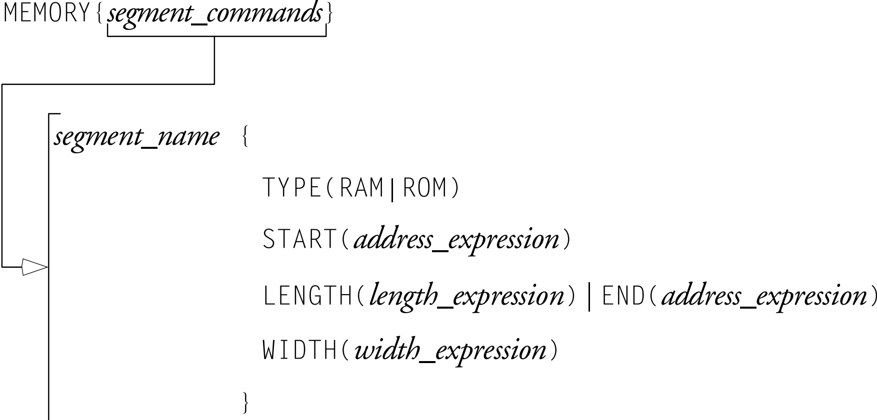


Figure 3-2. Syntax Tree of the MEMORY{} Command

Segment Declarations

A *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()`. Parts of a segment declaration are described as:

- *`segment_name`*
  Identifies the memory region. The *`segment_name`* must start with a letter, underscore, or point, and may include any letters, underscores, digits, and points, and must not conflict with LDF keywords.

- `START(`*`address_number`*`)`
  Specifies the memory segment's start address. The *`address_number`* must be an absolute address.

- `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 file for use by other development tools.

  - For Blackfin processors, use `TYPE()` to 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.

  - For ADSP-218x/9x DSPs, use `TYPE()` to specify two parameters: memory usage (PM for program memory or DM for data memory), and functional or hardware locus (RAM or ROM, as described above). In addition, in ADSP-218x LDFs, the PM/DM declarator specifies the memory type; and in ADSP-219x LDFs, it is used only to distinguish between a 16-bit or 24-bit logical data width.

`LENGTH(length_number)`
or
`END(address_number)`
Identifies the length of the memory segment (in words) or specifies the segment's end address. When you state the length, *length_number* is the number of addressable words within the region or an expression that evaluates to the number of words. When you state the end address, *address_number* is an absolute address

- `WIDTH(width_number)`
  Specifies the physical width (number of bits) of the on-chip or off-chip memory interface. The *width_number* paramter must be a whole number.

MPMEMORY{}

The `MPMEMORY{}` command is used with DSPs that implement physical shared memory, such as Blackfin processors and ADSP-2192-12 DSPs.

The `MPMEMORY{}` command specifies the offset of each processor's physical memory in a multiprocessor target system. After you declare the processor names and memory segment offsets with the `MPMEMORY{}` command, the linker uses the offsets during multiprocessor linking.

Refer to "MPMEMORY{}" on page 5-28 for a detailed description of the `MPMEMORY{}` command.

OVERLAY_GROUP{}

The `OVERLAY_GROUP{}` command is deprecated. It provides support for defining a set of overlays that share a block of runtime memory.

Refer to "OVERLAY_GROUP{}" on page 5-29 for a detailed description of the `OVERLAY_GROUP{}` command. Refer to "Memory Management Using Overlays" on page 5-4 for a detailed description of overlay functionality.

PACKING()

The `PACKING()` command is used with ADSP-218x and ADSP-219x DSPs (as described in detail in "Packing in ADSP-218x and ADSP-219x DSPs" on page 3-34).

The DSPs exchange data with their environments (on-chip or off-chip) through several buses. Refer to your DSP's data sheet and *Hardware Reference* manual.

The linker places data in memory according to the constraints imposed by your system's architecture. For this purpose, the `.LDF` file's `PACKING()` command specifies the placement order of bytes in memory. This ordering places data in the same sequence required by the DSP as it transfers data.

The `PACKING()` command allows the linker to structure its executable output to comport with your target's memory organization. `PACKING()` can be applied (scoped) on a segment-by-segment basis within the `.LDF` file, with adequate granularity to handle heterogeneous memory configurations. Divide a memory segment that requires more than one `PACKING()` command into multiple memory segments.

Non-trivial use of `PACKING()` commands reorder bytes in the executable (`.DXE`, `.SM`, or `.OVL`) files, so they arrive at the target in the correct number, alignment, and sequence. To accomplish this task, the command must be informed of the size of the reordered group, the byte order within the

group, and whether and where null bytes are to be inserted to preserve alignment on the target. In this case, null refers to usage; the target ignores the null byte. Coincidentally, the linker sets these bytes to 0s.

Syntax

The command syntax is:

```
PACKING (number_of_bytes byte_order_list)
```

where:

- `number_of_bytes` is an integer specifying the number of bytes to pack (reorder) before repeating the pattern
- `byte_order_list` is the output byte ordering—what the linker writes into memory. Each list entry consists of `B` followed by the byte's number (in a group) at the storage medium (memory) and follows these rules.
  - Parameters are delimited by whitespace characters
  - The total number of non-null bytes is `number_of_bytes`
  - If null bytes are included, they are labeled `B0`

Packing in ADSP-218x and ADSP-219x DSPs

In ADSP-21xx DSPs, some buses (PMA, DMA, PMD, ...) are 24 bits wide, and some (DMD) are 16 bits wide. Each device's configuration, placement, and amount of memory is implementation-specific.

Example

If the processor unpacks three 16-bit memory words to build two 24-bit instruction words, the following unpacked and storage orders could apply the requisite transfer order. The linker must reorder the bytes into their unpacked order.

Table 3-4. Unpacked Order vs. Native Storage Order

Unpacked Order: Two 24-Bit Internal Words	Native Storage Order 16-bit External Memory
B3, B2, B1 B6, B5, B4 word1 word2	B2, B1 B4, B3 B6, B5 addr[n] addr[n+1] addr[n+2]

Because the order of unpacked bytes does not match the requisite transfer order, the linker must reorder the bytes into their unpacked order.

Depending on how the memory interfaces to the PMD (program memory bus) is programmed, one command for doing so is:

```
PACKING(6 B2 B6 B1 B4 B6 B5);
```

Each set of 6 bytes would then be reordered as shown above.

The `PACKING()` order must match how the target is programmed to transfer data. In the previous example, the target must handle each 16-bit memory read (set of three) differently. This task is computationally expensive.

Efficient Packing

The packing (in Table 3-4 on page 3-35) is efficient. Each 16-bit memory location is used in its entirety to build two-thirds of an instruction word on the target. As a benefit, this scheme limits program size at the target and, perhaps, decreases download time.

Conversely, it implies programming overhead and transfer latency constraints as each address in a set of three memory reads must be handled differently. For this reason, the ordering shown above is possible rather than obligatory. You can program the port to handle any reordering.

Inefficient Packing: Null Bytes

Given the target byte order requirements of packing in Table 3-4 on page 3-35, it is much simpler to program memory access by directing the linker to add unused (null) bytes to your executable file. This method simplifies alignment and sequencing.

Consider an executable file that places bytes (in the following order) into three 16-bit memory addresses (MSByte on the left).

```
B2, B1,    B4, B3,    B6, B5
```

Suppose you want to load them as two 24-bit instructions.

```
B3, B2, B1,          B6, B5, B4
```

Reordering them with `PACKING(6 B3 B2 B1 B6 B5 B4)` leaves alignment and programming overhead, as shown above. However, the addition of null bytes after the third byte and the sixth byte simplifies things considerably.

```
PACKING(6 B3 B2 B1 B0 B6 B5 B4 B0)
```

This order defines four 16-bit memory reads, which generate two 24-bit addresses. Reads 1 and 3 are copied to the two MSBytes on the PMD. Reads 2 and 4 are copied to the LSByte on the PMD. The lower byte is ignored.

The same number of bytes are reordered as in the efficient packing example. Hence, the byte count parameter (6) to the `PACKING()` command is the same.

The byte designated as `B0` in the `PACKING()` syntax acts as a place holder. The linker sets that byte to zero in external memory.

Overlay Packing Formats

The PACKING() command also packs code and data for overlays, which, by definition, reside in a "live" external memory. Use an explicit PACKING() command when the width or byte order of stored data (executables or overlays in a "live" location) differs from its run-time organization.

Trivial Packing: No Reordering

If your memory organization matches its run-time environment and loads and runs without reordering, the linker uses (implicit) trivial packing. You need only to specify nontrivial packing in the .LDF file, though memory segments without reordering may be labeled as such to retain segment-specific packing order visibility, and to provide convenient locations to change the .LDF file when you change your target or memory configuration.

PAGE_INPUT()

The PAGE_INPUT() command supports paged memory architectures, such as ADSP-218x DSPs.

Use PAGE_INPUT() to specify overlays for memory pages in ADSP-218x DSPs. This command can be used instead of OVERLAY_INPUT (see "OVERLAY_GROUP{}" on page 3-33) in any location in the .LDF file. Refer to "Memory Management Using Overlays" on page 5-4 for a detailed description of overlay functionality.

The linker creates an additional symbol in each page overlay object. The symbol name identifies the overlay as one created from a PAGE_INPUT() command, identifies the register name used to activate the page, and specifies the page value. The linker determines the register name and page value. The register name is based on the type of memory selected for "run" and "live" space for the overlay. The page value is based on the description of the "live" space specified in the PAGE_INPUT() command.

Page overlays that appear in the PAGE_INPUT() commands as "live" memory must first be described in a MEMORY{} command. The specified address must contain a leading digit to indicate the page number.

For example, the memory corresponding to PMOVLAY 4 on an ADSP-2187 DSP could appear in the MEMORY{} command as:

```
seg_page4 { TYPE(PM RAM) START (0x42000) END(0x43FFF) WIDTH(24)}
```

According to this definition, to operate as both the "run-time" memory for all of the paged overlays and the "live" memory for PMOVLAY 0, the memory segment has page value 0, and the start address for this section is 0x2000. In implementations where no internal memory is at that address (for example, in the ADSP-2186 DSP), the linker generates an error for the page specified to "live" at that address.

PAGE_OUTPUT()

The PAGE_OUTPUT() command supports paged memory architectures, such as ADSP-218x DSPs.

Use the PAGE_OUTPUT() command with the PAGE_INPUT() command to specify overlays for memory pages. PAGE_OUTPUT() can be used instead of OVERLAY_OUTPUT (see "OVERLAY_GROUP{}" on page 5-29) in any location in the .LDF file. Refer to "Memory Management Using Overlays" on page 5-4 for a detailed description of overlay functionality.

PLIT{}

The PLIT{} (procedure linkage table) command in an .LDF file inserts assembly instructions that handle calls to functions in overlays. The PLIT{} commands provide a template from which the linker generates assembly code when a symbol resolves to a function in overlay memory. Refer to "PLIT{}" on page 5-34 for a detailed description of the PLIT{} command. Refer to "Memory Management Using Overlays" on page 5-4 for a detailed description of overlay and PLIT functionality.

PROCESSOR{}

The 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 3-3.

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

Figure 3-3. 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 3-13.

- 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 3-12 and "MEMORY{}" on page 3-29.

- `PLIT{plit_commands}`

  Defines procedure linkage table (PLIT) commands that apply only to this processor. For more information, see "PLIT{}" on page 3-38.

- `SECTIONS{section_commands}`

  Defines sections for placement within the executable (.DXE). For more information, see "SECTIONS{}" on page 3-42.

- `RESOLVE{symbol, resolver}`

  Ignores any `LINK_AGAINST()` command. For details, see the "RESOLVE()" command.

RESOLVE()

Use the `RESOLVE(symbol_name, resolver)` command to ignore a `LINK_AGAINST()` command for a specific symbol. This command overrides the search order for a specific variable or label. Refer to the "LINK_AGAINST()" on page 3-28 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.

For the `RESOLVE(symbol_name, resolver)` command:

- When the symbol is not defined in the current processor scope, the `<resolver>` supplies a filename, overriding any `LINK_AGAINST()`.
- When the symbol is defined in the current processor scope, the `<resolver>` supplies an address at which the linker should locate the symbol.

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

SEARCH_DIR()

The `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` file to ensure that 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 SECTIONS{} command uses memory segments (defined by MEMORY{} commands) to specify the placement of output sections into memory. Figure 3-4 shows syntax for the SECTIONS{} command.

Figure 3-4. Syntax Tree of the SECTIONS{} Command

An .LDF file may contain one SECTIONS{} command within each of the PROCESSOR{} commands. 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 file may contain only one SECTIONS{} command within each processor 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 3-13.

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

Parts of a `SECTION` declaration are:

- *`section_name`* – Must start with a letter, underscore, or period and may include any letters, underscores, digits, and points. A section_name must not conflict with any LDF keywords.

  The special section name `.PLIT` 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.

  The special section name `.MEMINIT` indicates where to place the "run-time" initialization structures to be used by the C run-time library. The linker will "place" this section into the largest available unused memory at the specified memory segment. The MemInit post-process will fill this space with the data needed by the C run-time library for run-time initialization. The `.MEMINIT` section should be placed in non-overlay memory.

- *`init_qualifier`* – Specifies run-time initialization type (optional).

  The qualifiers are:

  - NO_INIT – The section type contains un-initialized data. There is no data stored in the .DXE file for this section (equivalent to SHT_NOBITS legacy qualifier).
  - ZERO_INIT – The section type contains only "zero-initialized" data. If invoked with the -meminit switch (on page 2-44), the "zeroing" of the section is done at runtime by the C run-time library. If -meminit is not specified, the "zeroing" is done at "load" time.
  - RUNTIME_INIT – If the linker is invoked with the -meminit switch, this section will be filled at runtime. If -meminit is not specified, the section will be filled at "load" time.

- *section_commands* – May consist of any combination of such commands and/or expressions as:

  "INPUT_SECTIONS()" on page 3-45,

  "expression" on page 3-45,

  "FILL(hex number)" on page 3-46,

  "PLIT{plit_commands}" on page 3-46

  "OVERLAY_INPUT{overlay_commands}" on page 3-46

- *memory_segment* – Declares that the output section is placed in the specified memory segment.

  The *memory_segment* 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 file.

INPUT_SECTIONS()

The `INPUT_SECTIONS()` portion of a *`section_command`* identifies the parts of the program to place in the executable file. 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`*.

The syntax is:

```
INPUT_SECTIONS(library.dlb [ member.doj (input_label) ])
```

Note that spaces are significant in this syntax.

In the `INPUT_SECTIONS()` of the LDF command:

- *`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 (for example. `program`). Delimit the list of names with commas.

Example:

To place section "`program`" of object "`foo.doj`" in library "`myLib.dlb`":

```
INPUT_SECTIONS(myLib.dlb [ foo.doj (program) ])
```

expression

In a *`section_command`*, an *`expression`* manipulates symbols or positions the current location counter. See "LDF Expressions" on page 3-13 for details.

FILL(hex number)

In a *section_command*, the `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 5-34.

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 “Memory Management Using Overlays” on page 5-4 and “OVERLAY_GROUP{}” on page 5-29. For overlay code examples, see “Linking for Overlay Memory” on page C-17 and “Overlays Used With ADSP-218x DSPs” on page D-23.

The *overlay_commands* item consist of at least one of the following commands: `INPUT_SECTIONS()`, `OVERLAY_ID()`, `NUMBER_OF_OVERLAYS()`, `OVERLAY_OUTPUT()`, `ALGORITHM()`, `RESOLVE_LOCALLY()`, or `SIZE()`.

The *overlay_memory_segment* item (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 file, 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 3-45.
- OVERLAY_ID() – Returns the overlay ID.
- NUMBER_OF_OVERLAYS() – Returns the number of overlays that the current link generates when the FIRST_FIT or BEST_FIT overlay placement for ALGORITHM() is used.

  Note: Not currently available.
- 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.

  (FIRST_FIT – 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.

(BEST_FIT – 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, this command controls whether the linker generates PLIT entries for function calls that are resolved within the overlay.

  The default RESOLVE_LOCALLY(TRUE) 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 only ADSP-218x/9x DSP that supports SHARED_MEMORY{} is the ADSP-2192 DSP. All Blackfin processors support this function.

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. The SHARED_MEMORY{} command is used to produce .SM files. For more information, see "SHARED_MEMORY{}" on page 5-38.

4 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 executable file(s). The linker is described in Chapter 2, "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 adjustment. 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 4-2
- "Launching the Create LDF Wizard" on page 4-4
- "Expert Linker Window Overview" on page 4-10
- "Input Sections Pane" on page 4-12
- "Memory Map Pane" on page 4-18
- "Managing Object Properties" on page 4-50

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 you load and run 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. Freeing up memory enables you 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.

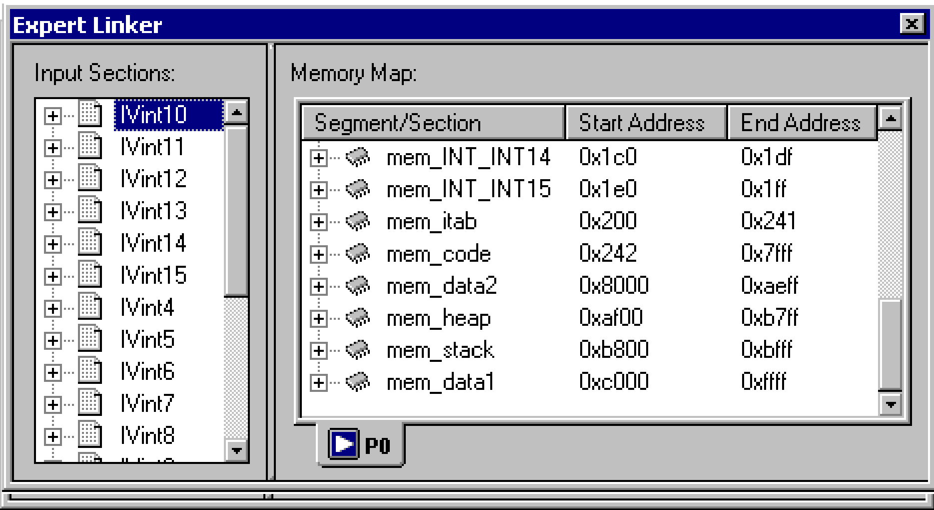


Figure 4-1. Expert Linker Window

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. You use the **Create LDF** option when you create a new project.

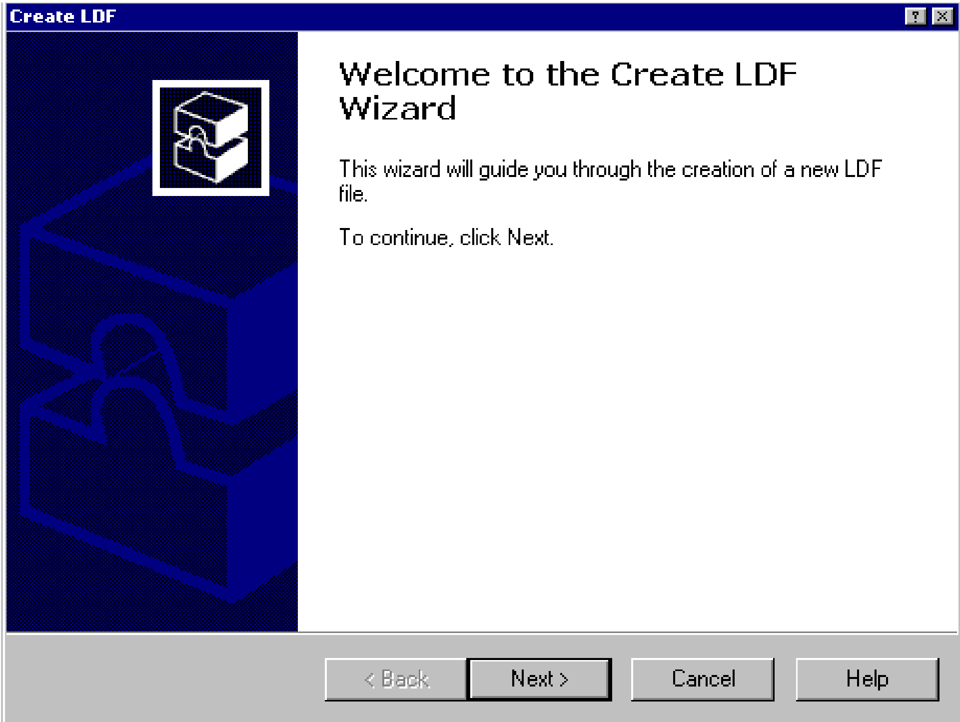


Figure 4-2. Welcome Page of the Create LDF Wizard

If an `.LDF` file is already 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.

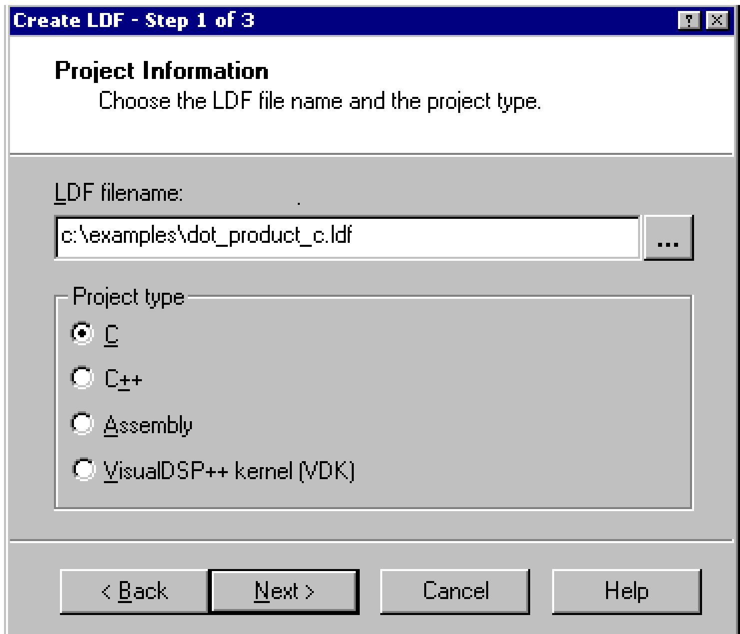


Figure 4-3. Selecting File Name and Project Type

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.

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 .C files are in the project, the default is **C**; if a VDK.H file is in the project, the default is **VDK**, and so on. This setting determines which template is used as a starting point.

Note that in case a mix of assembly and C files, or any other combination is used, the most abstract programming language should be selected. For example, for a project with C and assembly files, a C LDF should be selected. Similarly, for a C++ and C project the C++ LDF should be selected.

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.

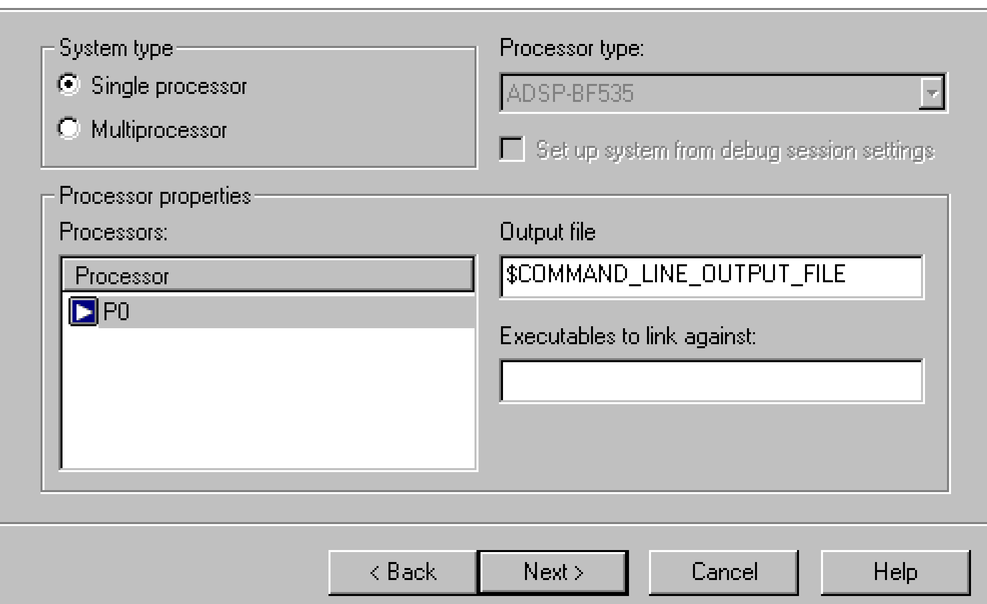


Figure 4-4. Selecting System and Processor Types

By default, the `.LDF` file is set for single processors. Under **System type**, select **Single processor** or **Multiprocessor**

- For a single-processor system, the **Processors** list shows only one processor and the MP address columns do not appear.
- For a multiprocessor system, right-click in the **Processor Properties** box to add the desired number of processors to be included in the `.LDF` file, 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 setting 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 grayed out 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, and so on).

When you select a processor in the **Processors** list, the system displays the output file name and the list of executable files to link against for that processor appear. You can change these files by typing a new file name. The file name may include a relative path, an LDF macro, or both. In addition, if the processor's ID is detected, the processor is placed in the correct position in the processor list.

For multiprocessor (MP) systems, the window (Figure 4-5) shows the list of processors in the project. The Expert Linker automatically displays MP address range for each processor space providing specific MP addresses and multiprocessor memory space (MMS) offsets which makes the use of MP commands much easier. This is an automatic replacement for the `MPMEMORY` linker command used in the LDF source file.

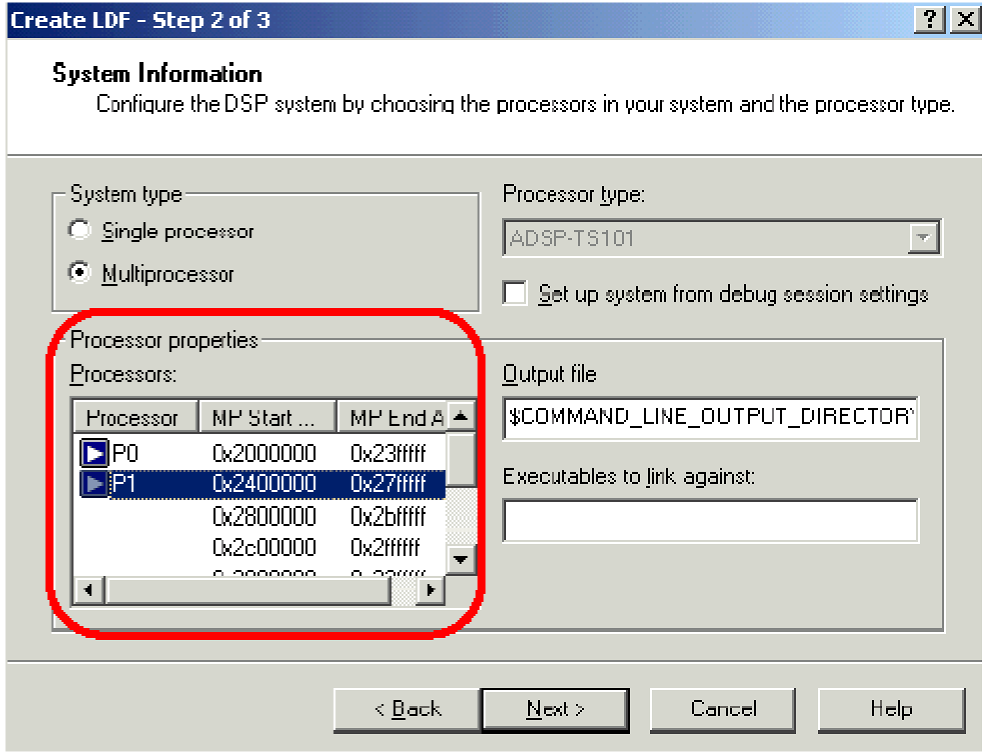


Figure 4-5. Processors and MMS Offset

The MP address range is available only for processors that have MP memory space, such as ADSP-2192 DSPs or ADSP-BF561 processors.

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

Step 3: Completing the LDF Wizard

From the **Wizard Completed** page, you can go back and verify 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 that contains the project file and adds it to the current project. The Expert Linker window appears and displays the contents of the new `.LDF` file.

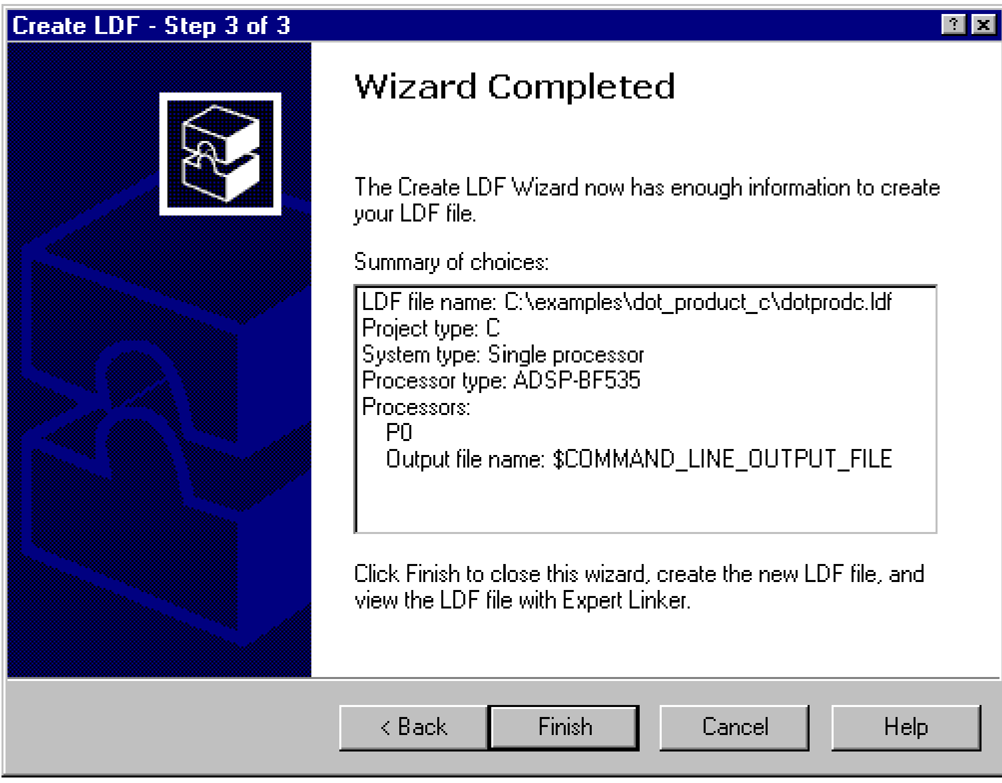


Figure 4-6. Wizard Completed Page of the Create LDF Wizard

Expert Linker Window Overview

The Expert Linker window contains two panes:

- The **Input Sections** pane (Figure 4-7) provides a tree display of the project's input sections (see "Input Sections Pane" on page 4-12).
- The **Memory Map** pane displays each memory map in a tree or graphical representation (see "Memory Map Pane" on page 4-18).

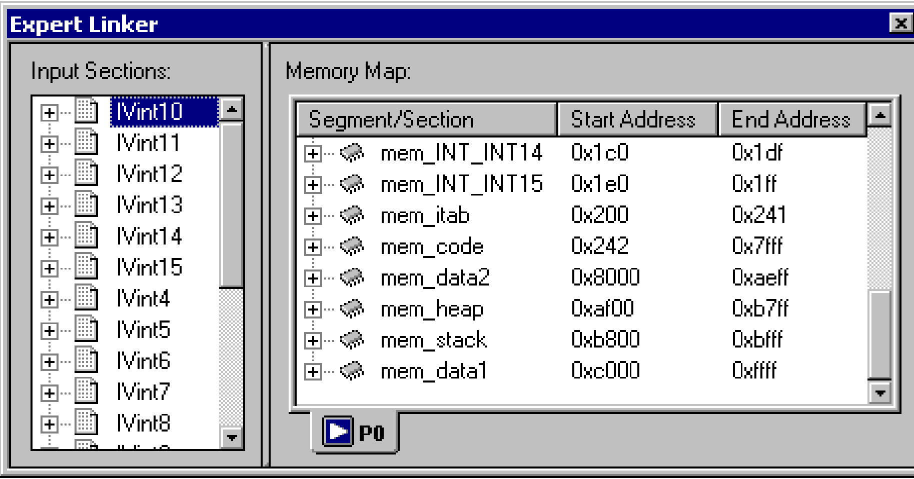


Figure 4-7. Expert Linker Window

Using commands in the LDF, the linker reads the input sections from object files (`.DOJ`) 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-

tion are displayed under that output section. For more information, refer to "Input Sections Pane" on page 4-12 and "Memory Map Pane" on page 4-18.

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

Input Sections Pane

The **Input Sections** pane (Figure 4-7) 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, a list of LDF macros, libraries, and object files may be contained in that input section. You can add or delete input sections, LDF macros, or objects/library files in this pane.

Input Sections Menu

Right-click an object in the **Input Sections** pane, and a menu appears as shown in Figure 4-8.

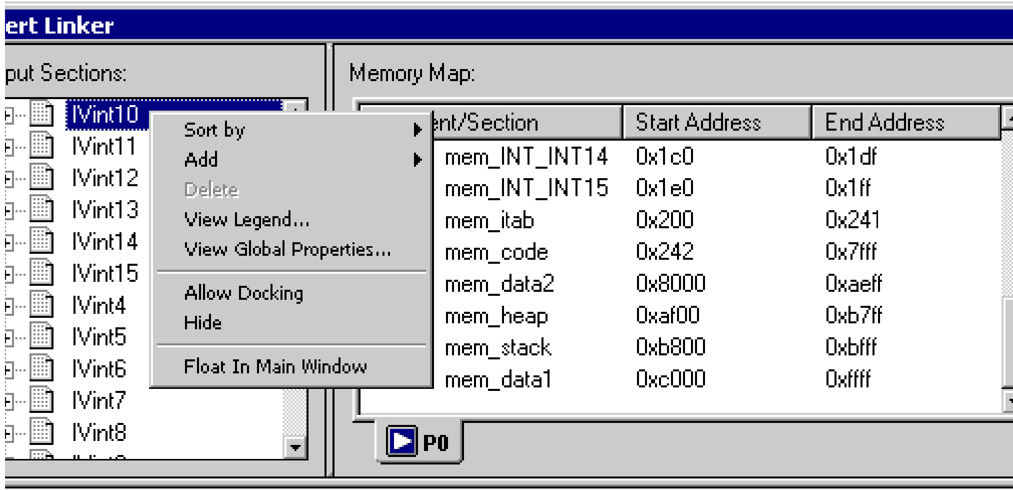


Figure 4-8. Input Sections Right-Click Menu

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)

- **Remove** – Removes an LDF macro from another LDF macro but does not delete the input section mappings that contain the removed macro. The difference between **Delete** menu and **Remove** is that **Delete** deletes the input section macros that contain the deleted macro. The **Remove** option becomes available only if you right-click on an LDF macro that is part of another LDF macro

- **Expand All LDF Macros** – Expands all the LDF macros in the input sections pane so that the contents of all the LDF macros are visible.

- **View Legend** – Displays the **Legend** dialog box which shows icons and colors used by the Expert Linker

- **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 which 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 4-41 on page 4-51).

Mapping an Input Section to an Output Section

Using the Expert Linker, you can map an input section to an output section. By using 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, are mapped 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 4-9.

As you drag 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 4-9)

The red “x” on an icon indicates this object/file is not mapped.

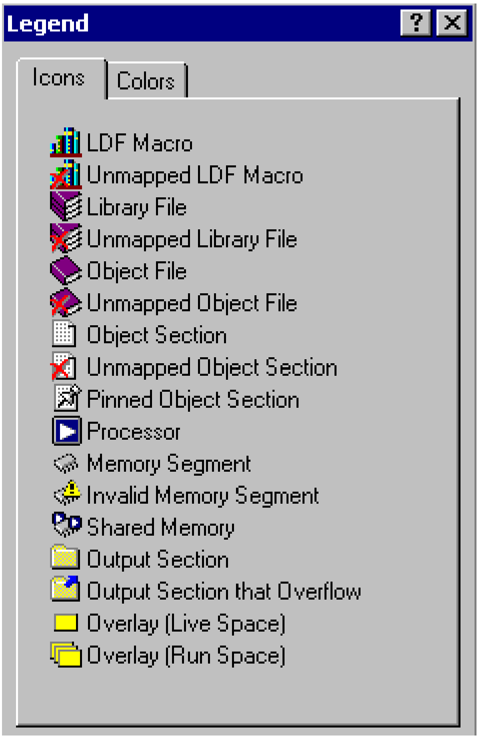


Figure 4-9. Legend Dialog Box – Icons Page

Click the **Colors** tab to view the **Colors** page (Figure 4-10). 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.

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 4-11).

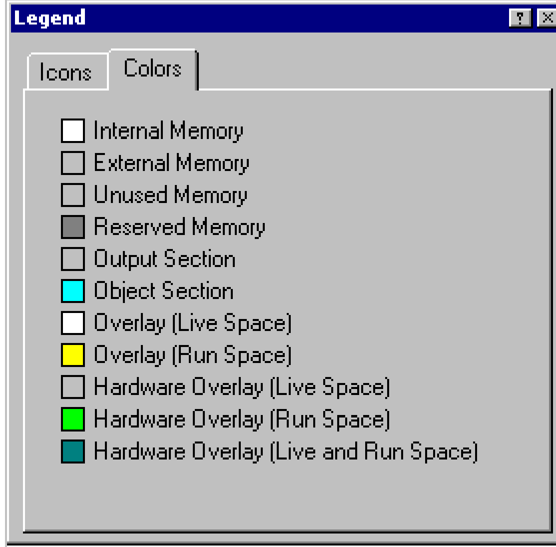


Figure 4-10. Legend Dialog Box – Colors Page

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.

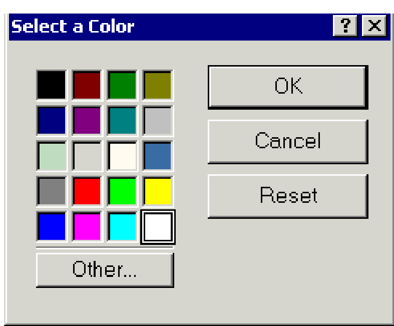


Figure 4-11. Selecting Colors

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. Refer to Figure 4-12 and Figure 4-13.

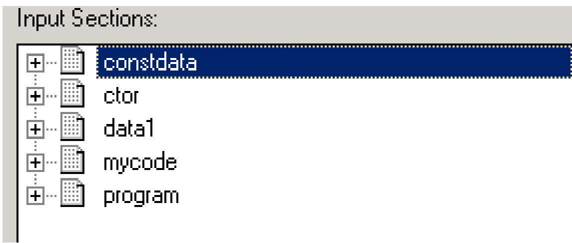


Figure 4-12. Expert Linker Window – Sorted by Input Sections

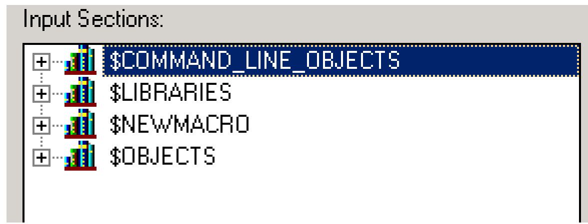


Figure 4-13. Expert Linker Window – Sorted by LDF Macros

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.

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 specify 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 (.DXE) file and specifies its file name. Figure 4-14 shows a typical memory map pane; ADSP-218x DSP is used this example.

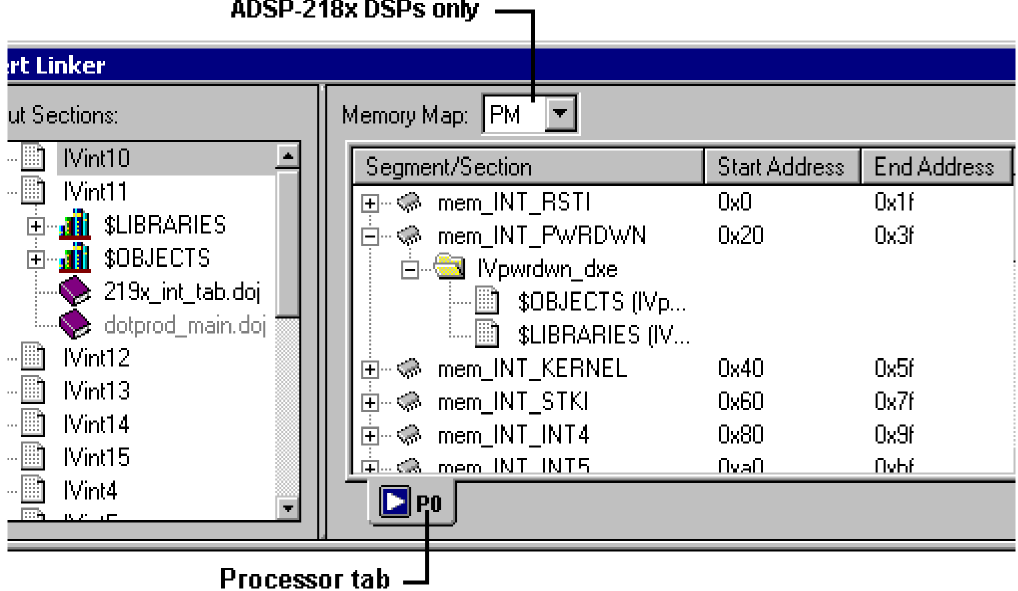


Figure 4-14. Expert Linker Window – Memory Map

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. The two viewing modes are 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.

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 4-9 on page 4-15). Move the mouse pointer over the icon and a tooltip displays a message describing why the segment is invalid.

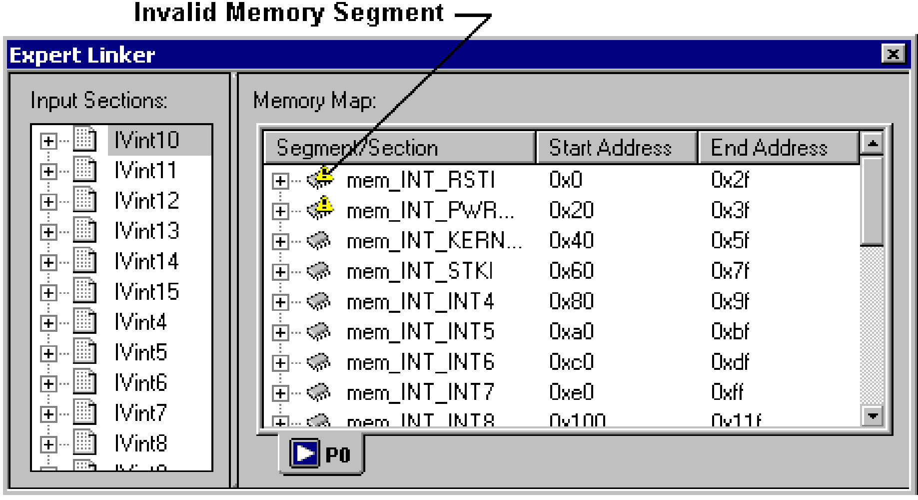


Figure 4-15. Memory Map With Invalid Memory Segments

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.

View Mode

- **Memory Map Tree** – Displays the memory map in a tree representation (see Figure 4-16 on page 4-22)
- **Graphical Memory Map** – Displays the memory map in graphical blocks (see Figure 4-17 on page 4-24)

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 4-52 on page 4-67).

Delete – Deletes the selected object

Expand All – Expands all items in the memory map tree so that their contents are visible.

Pin to Output Section – Pins an object section to an output section to prevent it from overflowing to another output section. This command appears only when you right-click on an object section that is part of an output section specified to overflow to another output section.

View Section Contents – Invokes a dialog box that displays the contents of the input or output section. It is available only after you link or build the project and then right-click on an input or object section (see Figure 4-29 on page 4-37).

View Symbols – Invokes a dialog box that displays the symbols for the project, overlay, or input section. It is available only after you link the project and then right-click on a processor, overlay, or input section (see Figure 4-41 on page 4-51).

Properties – Displays a **Properties** dialog box for the selected object. The **Properties** menu is context-sensitive; different properties are displayed 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 or 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 4-9 on page 4-15 and Figure 4-10 on page 4-16.

View Global Properties – Displays a **Global Properties** dialog box that lists the map file generated after you link the project. It also provides access to some processor and setup information (see Figure 4-42 on page 4-52).

Tree View Memory Map Representation

In the tree view (selected by right-clicking and choosing **View Mode -> Memory Map Tree**), the memory map is displayed with memory segments at the top level.

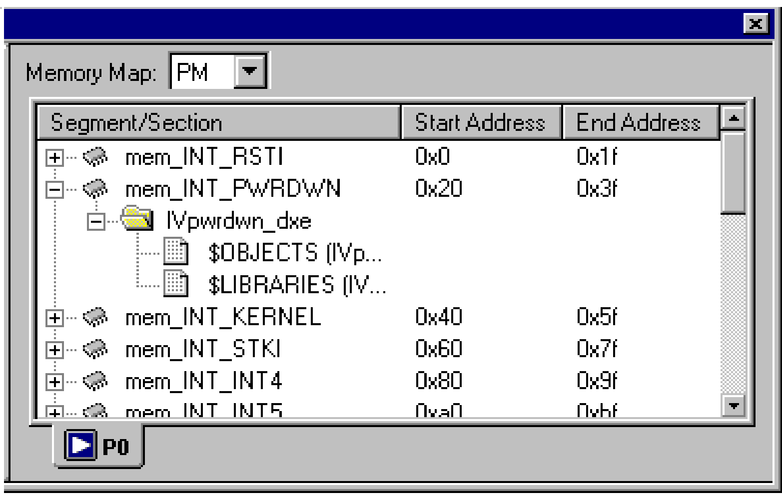


Figure 4-16. Expert Linker Window – Memory Map

Each memory segment may have one or more output sections under it. Input sections mapped to an output section appear 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 you link 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 4-27). Figure 4-17, Figure 4-18 and Figure 4-19 show examples of a graphical memory map representations.

In graphical view, the memory map comprises blocks of different colors that represent 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 are displayed in equal sizes within the memory segment. After linking, the height of the objects is scaled as a percentage of the total memory segment size. Object section names appear only when there is enough room to display them.
- Addresses are listed in ascending order from top to bottom.

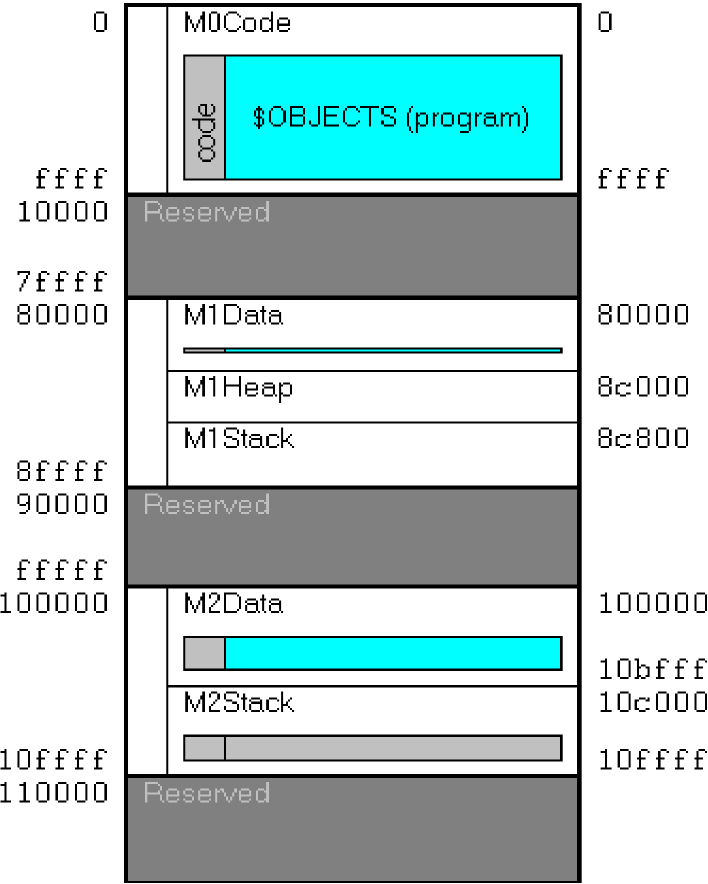


Figure 4-17. Graphical Memory Map Representation

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 4-28).

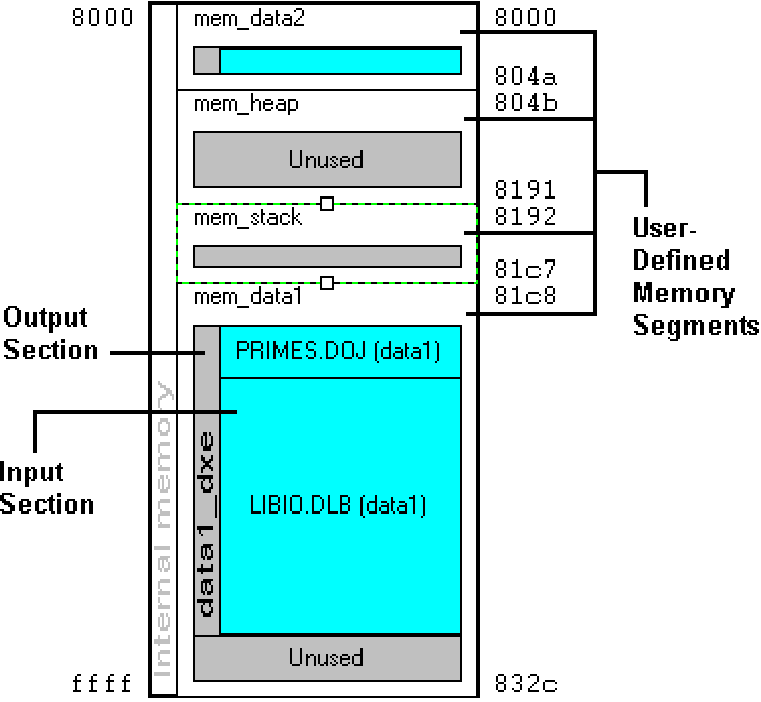


Figure 4-18. Viewing Sections and Segments in Memory Map

You can drag and drop any object except memory segments. See Figure 4-20.

Select a memory segment to display its border. Drag the border to change the memory segment's size. The size of the selected and adjacent memory segments change.

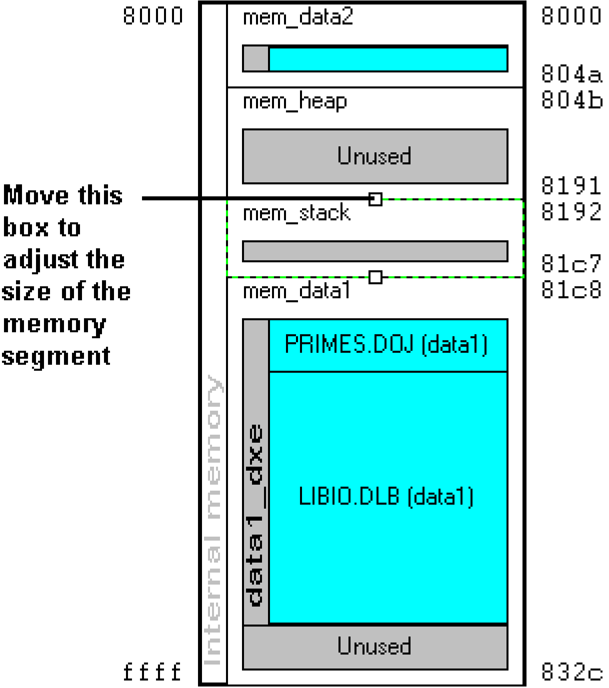


Figure 4-19. Adjusting the Size of a Memory Segment

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 is highlighted as shown in Figure 4-21 on page 4-28. If you move the mouse pointer over an object in the graphical memory map, a yellow tooltip displays the information about the object (such as name, address, and size).

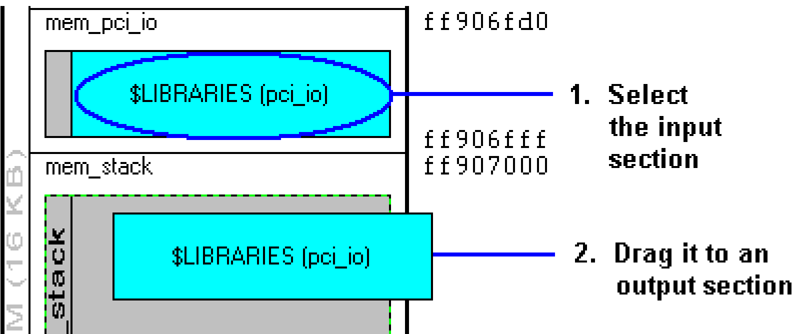


Figure 4-20. Dragging and Dropping an Object

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 are placed after you link 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 4-23 on page 4-30 illustrates a memory map before linking is performed.

- To enable post-link view from the **Memory Map** pane, right-click and choose **View** and **Link Results** (**Post-Link**). Figure 4-24 on page 4-31 illustrates a memory map after linking is performed.

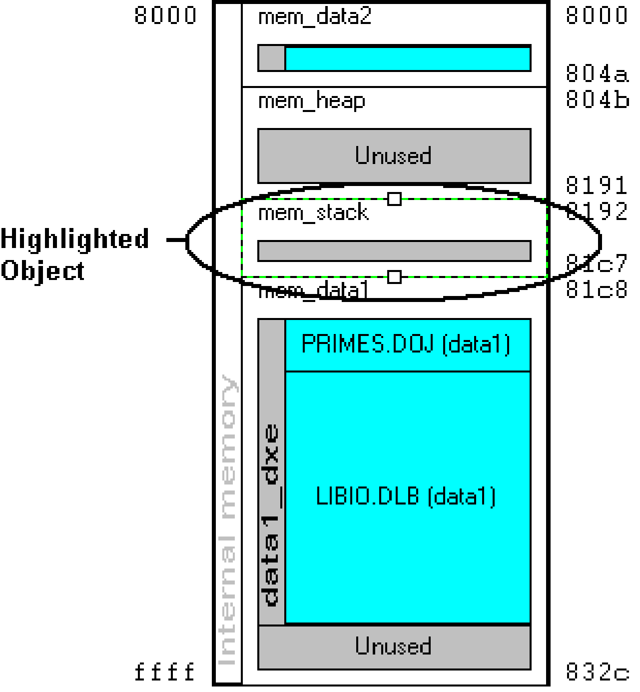


Figure 4-21. A Highlighted Memory Segment in the Memory Map

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 (see Figure 4-24 on page 4-31).

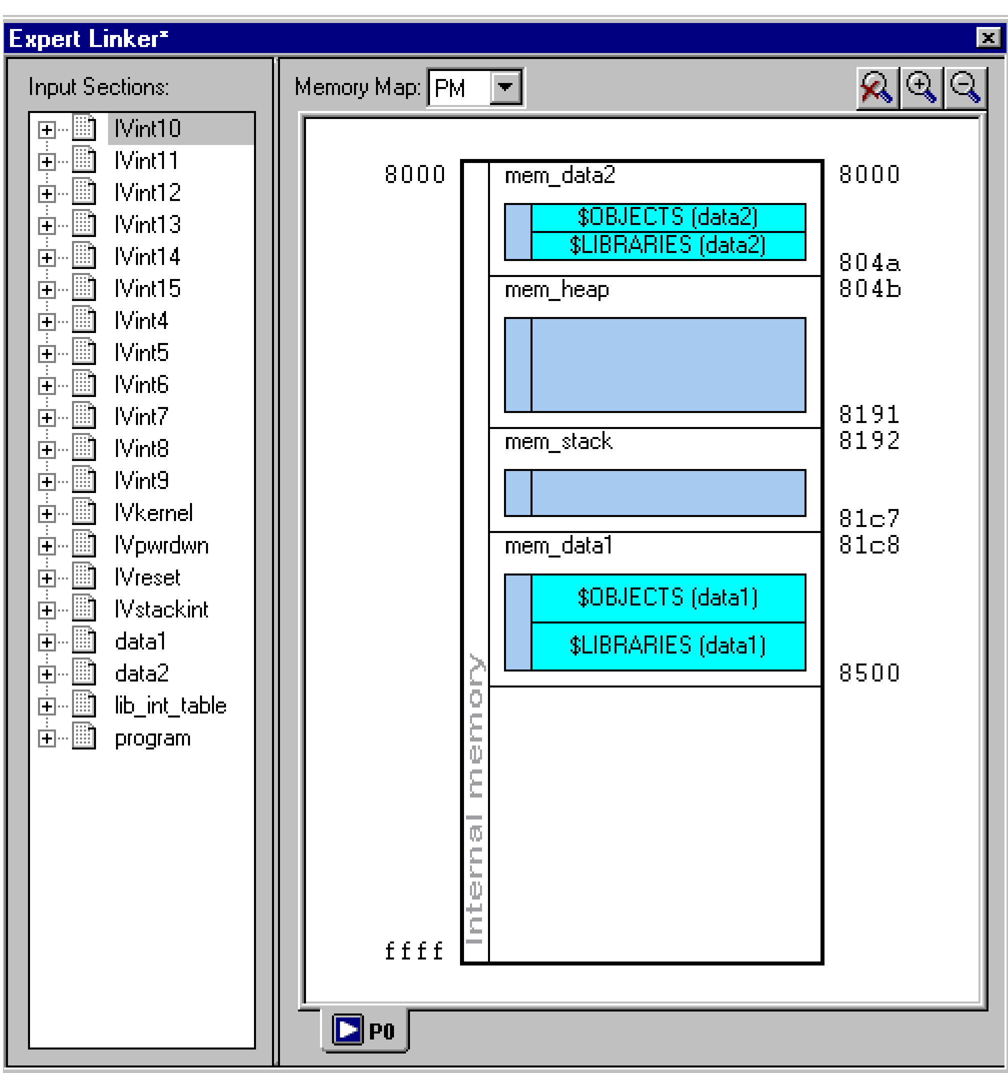


Figure 4-22. Memory Map Pane in Pre-Link View

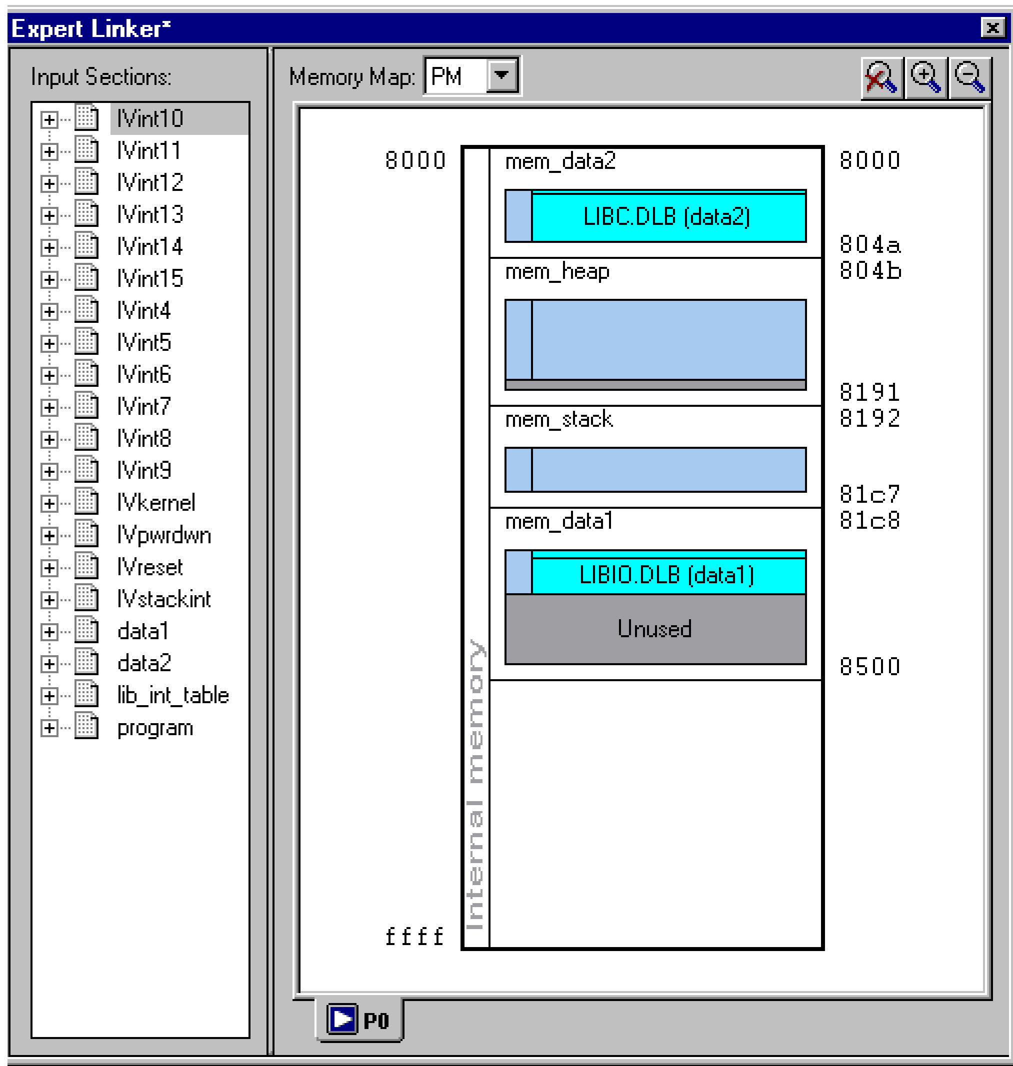


Figure 4-23. Memory Map Pane in Post-Link View

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 mode, click on the magnifying glass icon with the “**x**” above the upper right corner of the memory map window.
- 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 4-24. 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, as shown in Figure 4-25.

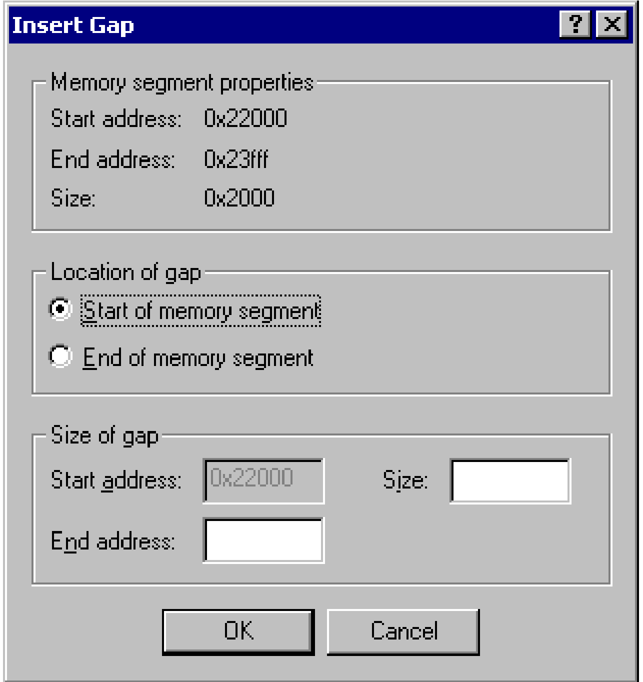


Figure 4-25. Insert Gap Dialog Box

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 4-26 shows an overlay in live space, and Figure 4-27 shows an overlay in run space.

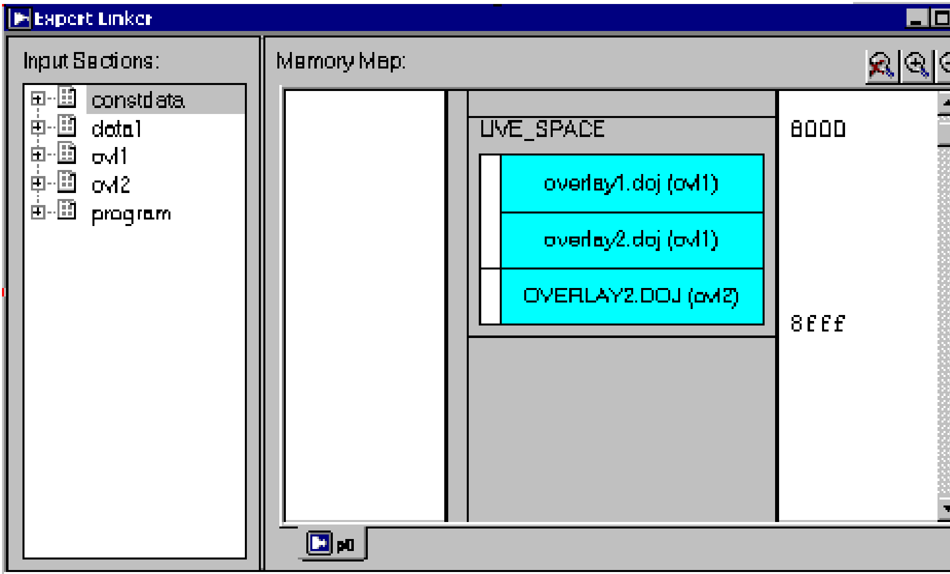


Figure 4-26. Graphical Memory Map Showing an Overlay in Live Space

Overlays in a “run” space appear 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 a 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.

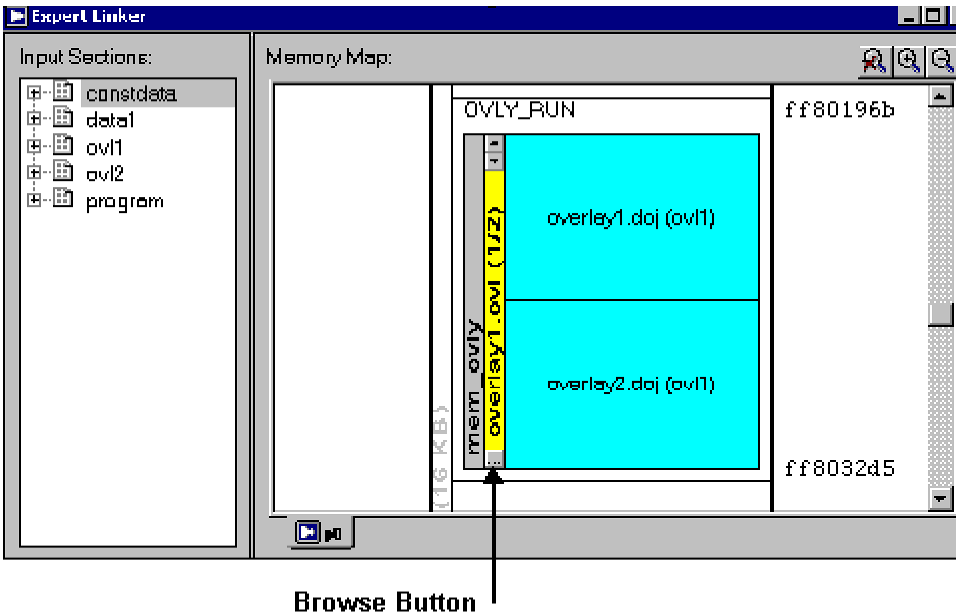


Figure 4-27. Graphical Memory Map Showing an Overlay Run Space

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 appears in the “run” and “live” spaces in two different colors in the memory map.

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` utility (`elfdump.exe`) 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. For example, Figure 4-28 shows Output Section contents in hex format.

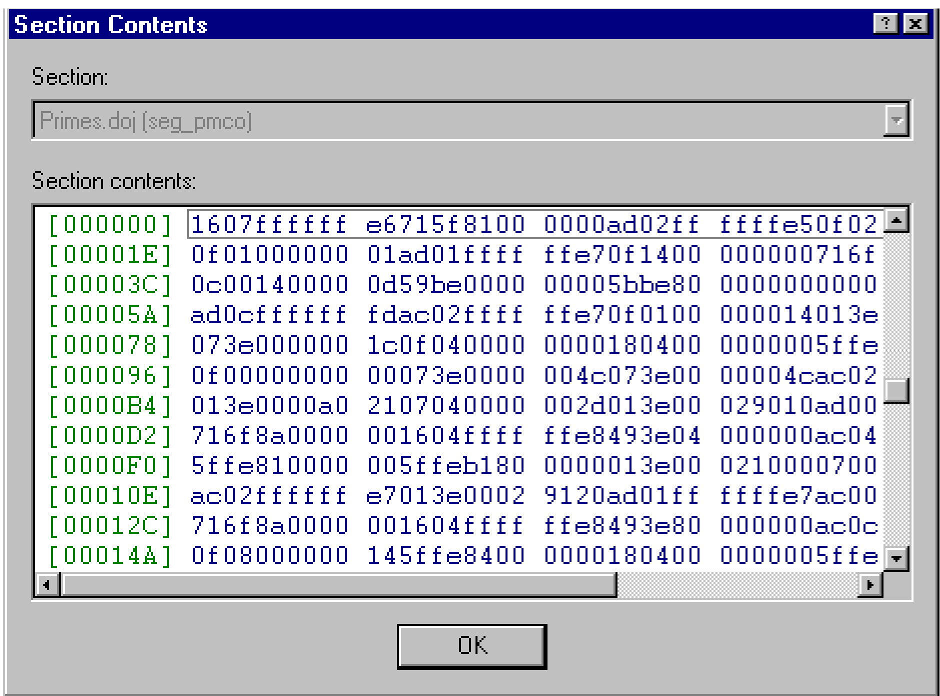


Figure 4-28. Output Section Contents in Hex Format

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 appears 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.

Figure 4-29 and Figure 4-30 illustrate memory data formats available for the selected output section.

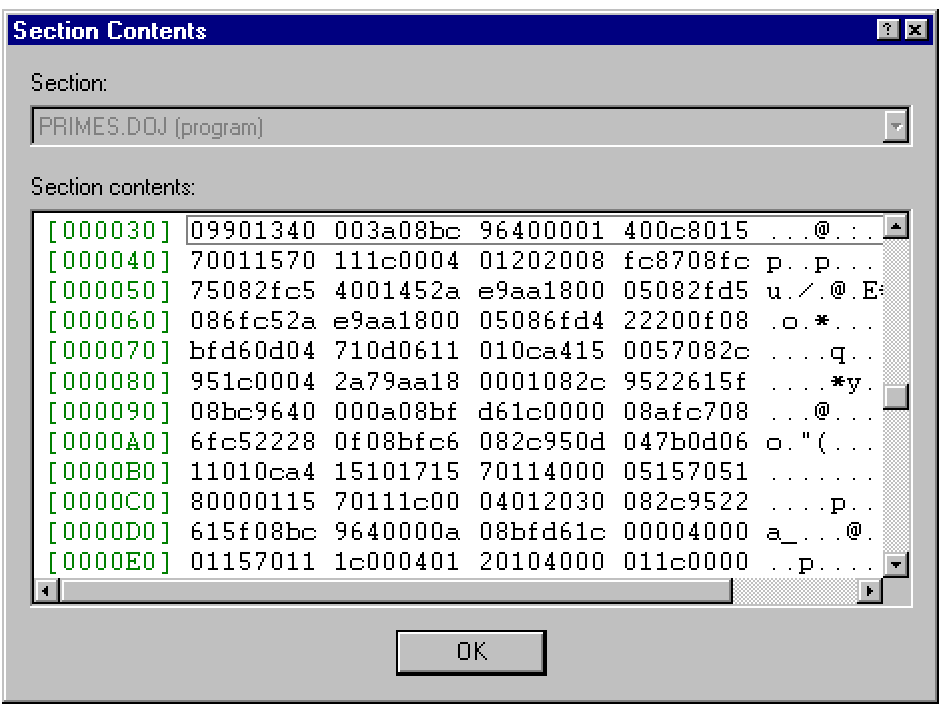


Figure 4-29. Output Section Contents in Hex and ASCII Format

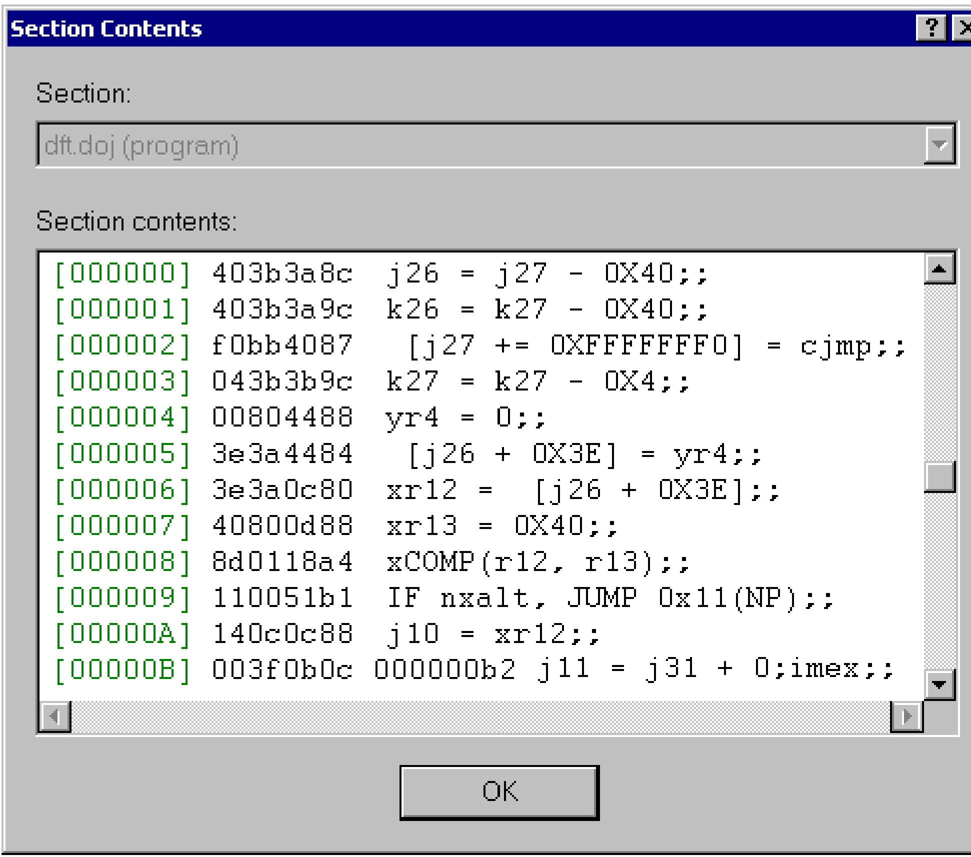


Figure 4-30. Output Section Contents in Hex and Assembly Format

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 in which it appears in the linker's map output. Sort symbols by name, address, and so on by clicking the column headings.

View Symbols

Name	Address	Size	Binding	File Name	Section
_LN$7__m...	0x286	0x0	STB_LOCAL	PRIMES.DOJ	program
_LN$8__m...	0x28e	0x0	STB_LOCAL	PRIMES.DOJ	program
_LN$9__m...	0x292	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_250002	0x28e	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_250004	0x2bd	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_250005	0x292	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_250007	0x2a7	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_250008	0x2a6	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_316000	0x2c4	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_316001	0x281	0x0	STB_LOCAL	PRIMES.DOJ	program
_L_316002	0x284	0x0	STB_LOCAL	PRIMES.DOJ	program
__EPC_text	0x273	0x0	STB_LOCAL	PRIMES.DOJ	program
_main	0x273	0x0	STB_GLOB...	PRIMES.DOJ	program
program	0x273	0x0	STB_LOCAL	PRIMES.DOJ	program

OK

Figure 4-31. View Symbols Dialog Box

To view symbols (Figure 4-31):

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.

Profiling Object Sections

You can use Expert Linker to profile object sections in your program. After doing so, Expert Linker graphically displays how much time was spent in each object section so you can locate code "hotspots" and move the code to faster, internal memory.

The following is a sample profiling procedure. Start it by selecting **Profile execution of object sections** in the **General** page of the **Global Properties** dialog box (Figure 4-32)

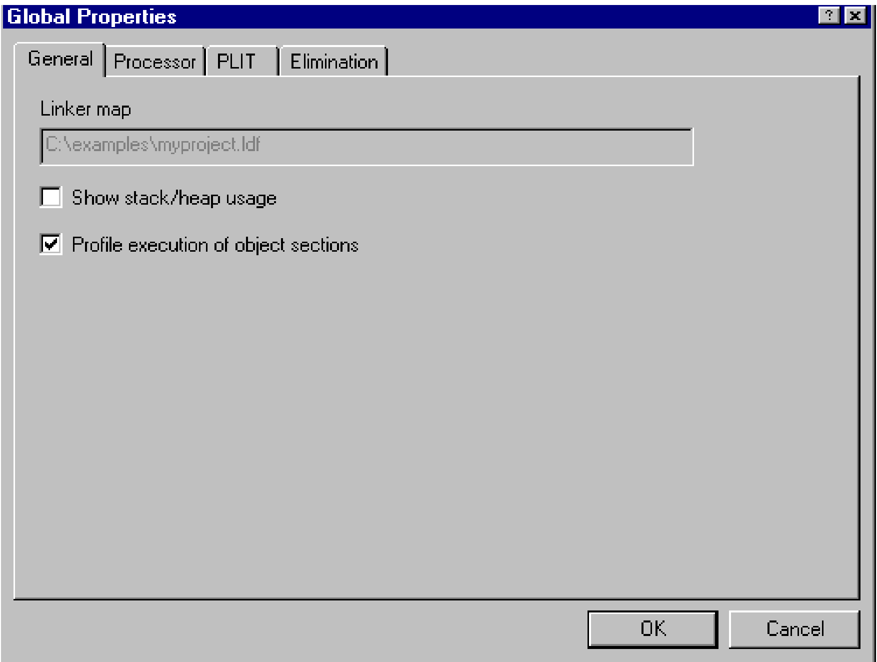


Figure 4-32. General Page of the Global Properties Dialog Box

Then build the project and load the program. After the program is loaded, Expert Linker sets up the profiling bins to collect the profiling information.

When the program run is complete, Expert Linker colors each object section with a different shade of red to indicate how much time was spent executing that section. For example, see Figure 4-33.

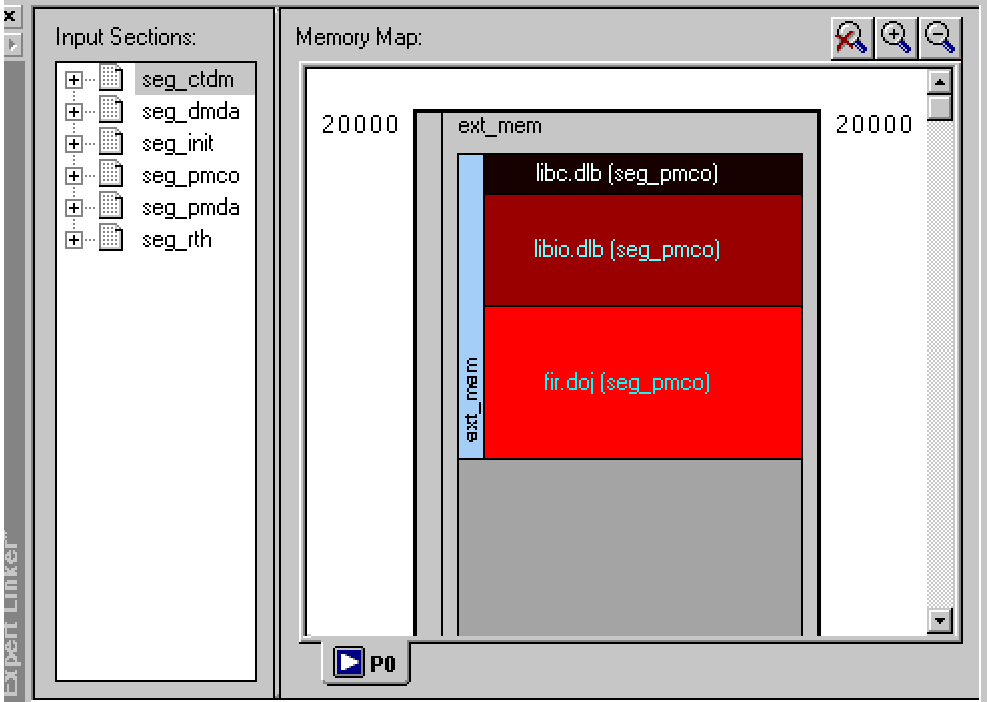


Figure 4-33. Colored Object Sections

The `fir.doj (seg_pmco)` appears in the brightest shade of red, indicating that it takes up most of the execution time. The shading of `libio.dlb (seg_pmco)` is not as bright. This indicates that it takes up less execution time than `fir.doj (seg_pmco)`. The shading of `libc.dlb (seg_pmco)` is black, indicating that it takes up a negligible amount of the total execution time.

From Expert Linker, you can view PC sample counts for object sections. To view an actual PC sample count (Figure 4-34), move the mouse pointer over an object section and view the PC sample count.

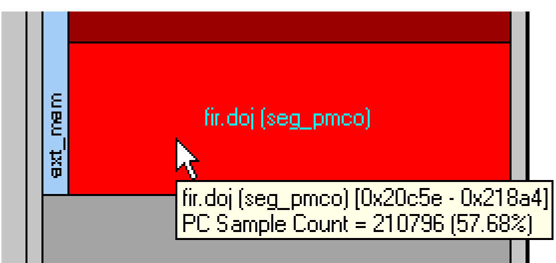


Figure 4-34. PC Sample Count

To view sample counts for functions located within an object section, double-click on the object section (Figure 4-35).

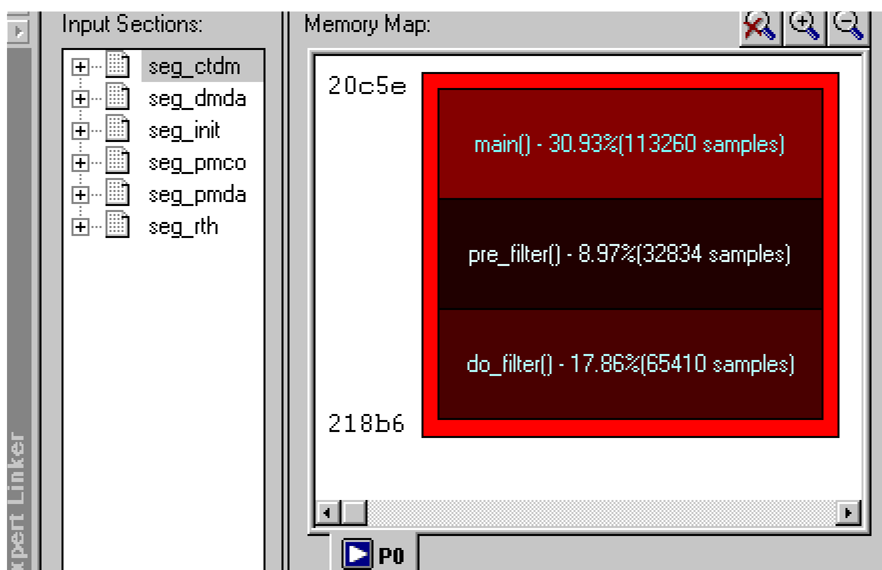


Figure 4-35. Sample Count of Functions within Object Section

Functions are available only when objects are compiled with debug information.

You can view detailed profile information such as the sample counts for each line in the function (Figure 4-36). To view profile information, double-click on a function.

Expert Linker : ADSP-21065L

Histogram	%	Execution Uni
	4.65%	xtheputcha...
	0.70%	printf_32....
	0.00%	xdecimal_d...
	0.00%	xdecimal_s...
	0.00%	xprint_e_f...
	0.00%	xprint_f_f...
	0.00%	xprint_g_f...
	10.22%	xprnt_32.d...
	57.76%	⊟ fir.doj ...
	17.86%	do_filter()
	8.97%	pre_filter(
	30.93%	main()

%	Line	C:\C-FIR\fir.c
6.64%	207	for (i =
21.15%	208	temp
	209	
	210	// compi
	211	// lcntr
	212	// mrf=m
	213	// _L$31
	214	
0.36%	215	output[k
	216	
0.70%	217	for (j =
0.52%	218	state
	219	}

Total Samples: 366165

Figure 4-36. Detailed Profile Information

To view PC samples with percent of total samples, view the memory map tree (Figure 4-37).

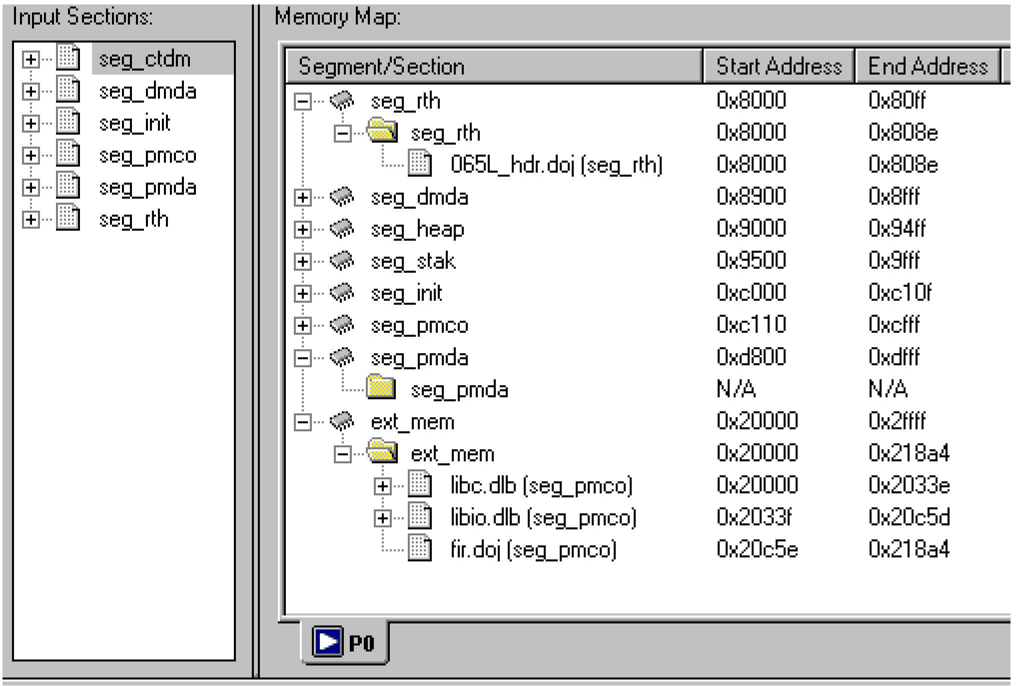


Figure 4-37. Percentage of Total PC Sample Count

Adding Shared Memory Segments and Linking Object Files

In many DSP applications where large amounts of memory for multiprocessing tasks and sharing of data are required, an external resource in the form of shared memory may be desired.

Refer to Engineer To Engineer Note EE-202 "*Using the Expert Linker for Multiprocessor LDF*" for detailed description and procedure. You can find this EE Note on Analog Devices website at `http://www.analog.com/UploadedFiles/Application_Notes`.

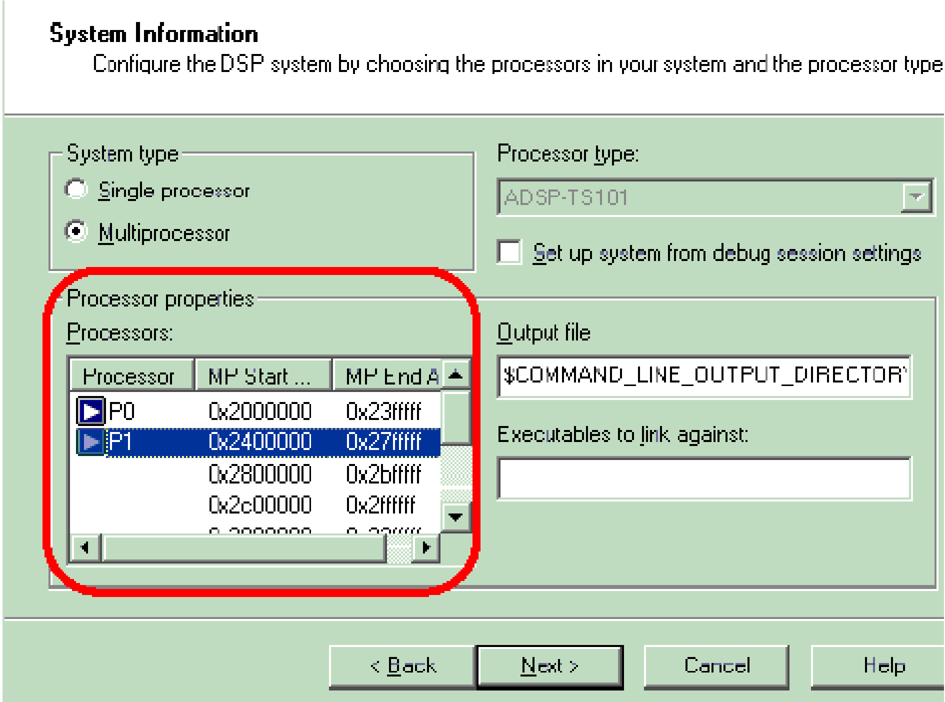


Figure 4-38. Multiprocessor LDF selection

To add a shared memory section to the .LDF file, right-click in the **Memory Map** pane and select **New/Shared Memory**. Then specify a name for the shared memory segment (.SM) and select the processors that have access to this shared memory segment.

As shown in Figure 4-39, a new shared memory segment, visible to processors P0 and P1, has been successfully added to the system. Note that variables declared in the shared memory segment will be accessed by both processors in the system. In order for the linker to be able to correctly resolve these variables, the link against command should be used once again.

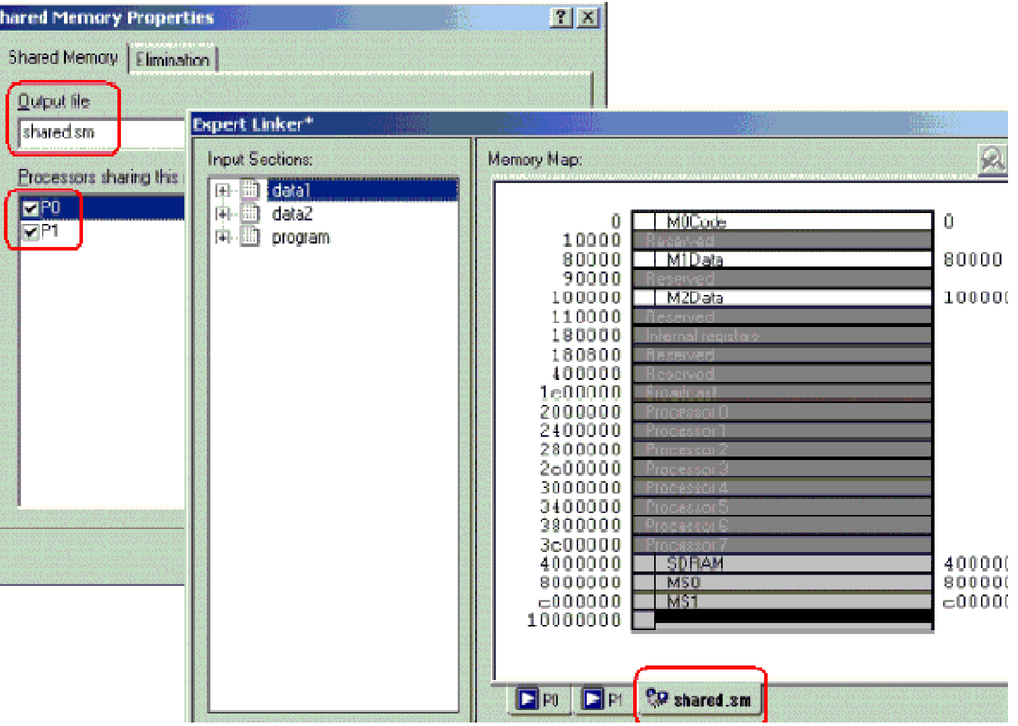


Figure 4-39. Shared Memory Segment

The Expert Linker automatically does this, and therefore you do not need to perform any additional modifications to the LDF.

You can confirm that Expert Linker has correctly added the `.sm` file to the link against command line by selecting **View Global Properties** in the **Memory Map** pane and clicking on the **Processor** tab.

The `shared.sm` file should now be contained in the **Executables to Link Against box** for each processor.

You can use Expert Linker to detect non-linked input sections, such as a variable declared in external SDRAM memory, which belongs to the shared memory segment.

When both processors and the shared memory segments have been properly configured, and Expert Linker has detected all input sections, you can link the object files from different input sections to their corresponding memory sections.

In general, the linking process consists of following steps:

1. Sort the left pane of the Expert Linker window by LDF Macros instead of Input Sections (default setting). To do that, right-click on the left pane and select **Sort by/LDF Macros**.

2. Right-click on the **LDF Macro** window and add a new macro for P0 (**Add/LDF Macro**). For example, `$OBJECTS_P0`. Repeat the same step for `P1` and `shared.sm`.

3. Add the object files (`.doj`) that correspond to each processor as well as to the shared memory segment. This is done by right-clicking on each recently created LDF macro and then selecting **Add/Object/Library File**.

   The use of LDF macros becomes extremely useful in systems where there is more than one sorted by Input Sections instead of LDF

macros..doj file per processor or shared memory segment, in which case the same step previously explained should be followed for each .doj file.

4. Delete the LDF macro $COMMAND_LINE_OBJECTS from the $OBJECTS macro to avoid duplicate of object files during the linking process. Right-click on the $COMMAND_LINE_OBJECTS macro and click **Remove**.

5. The left pane needs to be sorted by Input Sections instead of LDF macros. To do that, right-click on the left pane and select **Sort by/Input Sections**. Additionally, change in the right pane the **Memory Map View Mode** from Graphical to Tree mode. Right-click on the **Memory Map** window, select **View Mode** and then **Memory Map Tree**.

6. Map the new macros into memory. To do this, place each macro into its corresponding memory section.

7. Repeat the same steps for processor P1 ($OBJECTS_P1) and for the shared memory segment, shared.sm (place $OBJECTS_SM in the SDRAM section).

8. Press **Rebuild All**.

9. Select one of the processors by clicking on the processor's name tab. In this case, P0 is selected first. Then, place (drag and drop) the recently created LDF macro, $OBJECTS_P0, in its corresponding memory segment. The red crosses denoting the "non-linked" sections have disappeared, indicating that the input sections have been properly mapped into memory.

Also, note that the LDF macros that were moved from the **Input Sections** window (left pane) to their corresponding sections in the **Memory Map** window (right pane) have been automatically replaced during linking process with the actual object files (.doj) used by the linker.

The LDF is now complete. Figure 4-40 illustrates the generated `.LDF` file in the **Source Code View** mode.

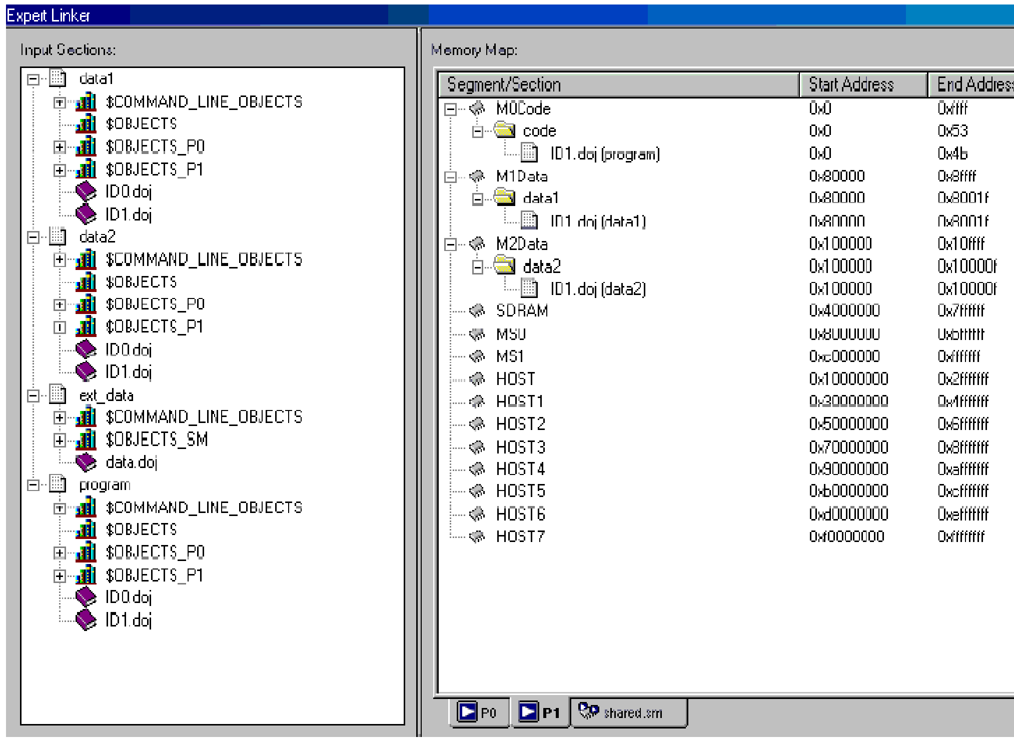


Figure 4-40. Expert Linker Multiprocessor LDF

The multiprocessor linker commands, `MPMEMORY`, `SHARED MEMORY` and `LINK AGAINST`, as well as the corresponding LDF macros, have been successfully generated by the Expert Linker in a way absolutely transparent to the user.

The complete project is now ready to be built. Once again, perform a **Rebuild All** and start debugging with the application code.

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 4-51
- "Managing Processor Properties" on page 4-52
- "Managing PLIT Properties for Overlays" on page 4-54
- "Managing Elimination Properties" on page 4-55
- "Managing Symbols Properties" on page 4-57
- "Managing Memory Segment Properties" on page 4-61
- "Managing Output Section Properties" on page 4-62
- "Managing Packing Properties" on page 4-64
- "Managing Alignment and Fill Properties" on page 4-65
- "Managing Overlay Properties" on page 4-67
- "Managing Stack and Heap in Processor Memory" on page 4-69

Managing Global Properties

The **Global Properties** dialog box provides these selections (Figure 4-41):

- **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.
- If **Profile execution of object sections** is selected, Expert Linker enables the profiling feature that allows you to see hotspots in object sections and to fine-tune the placement of object sections

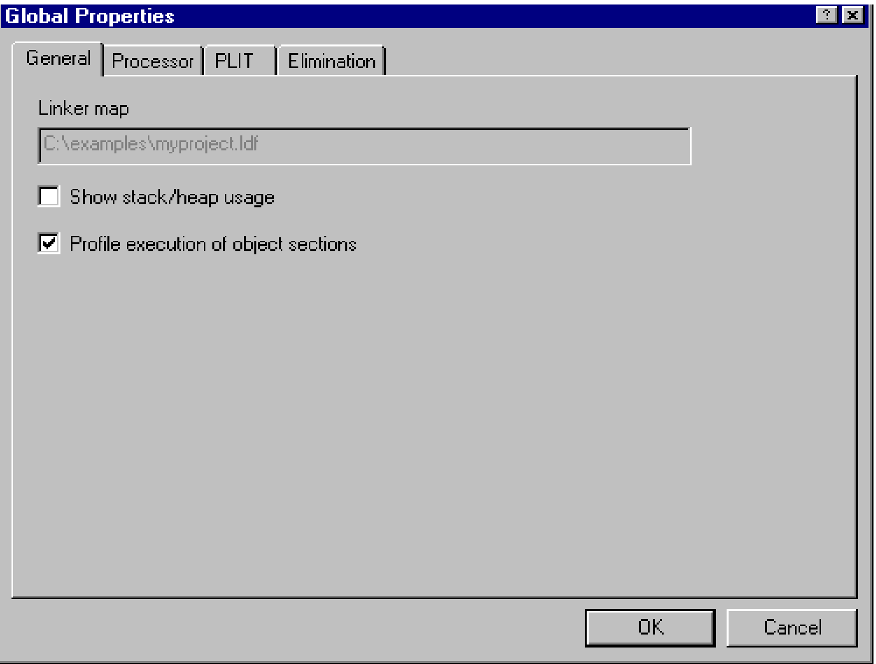


Figure 4-41. General Page of the Global Properties Dialog Box

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 **Processor** tab (Figure 4-42).

The **Processor** tab allows you to reconfigure the processor setup.

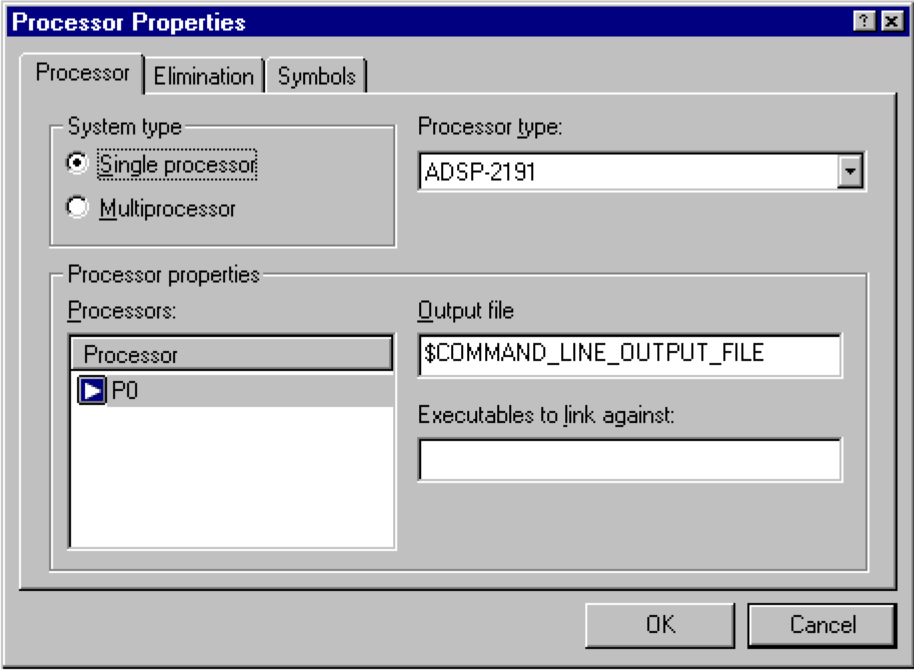


Figure 4-42. Processor Page of the Processor Properties Dialog Box

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, LDF macro, or both.

Additionally, you can rename a processor by selecting the processor, right-clicking, 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 4-43).

Figure 4-43. 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 4-44).

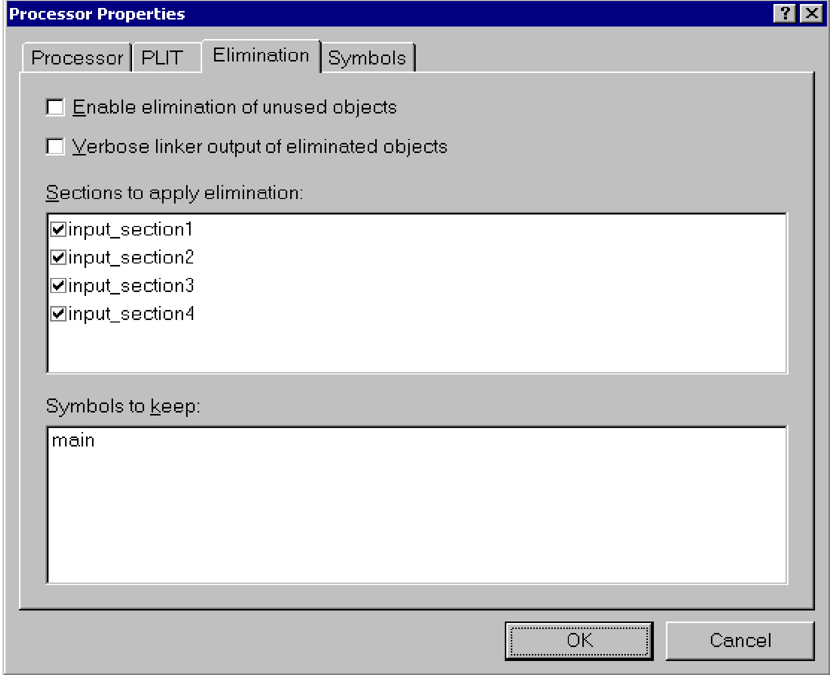


Figure 4-44. Processor Properties Dialog Box – Elimination Tab

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.

When **Verbose linker output of eliminated objects** is selected, the eliminated objects are shown as linker output in the **Output** window's **Build** page during linking. This check box is grayed out when the **Enable elimi-**

nation 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 does 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 (Figure 4-45)
4. Right-click in the **Symbols to keep** window.

   Choose **Add Symbol** to open the dialog box and 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 (Figure 4-46).

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.

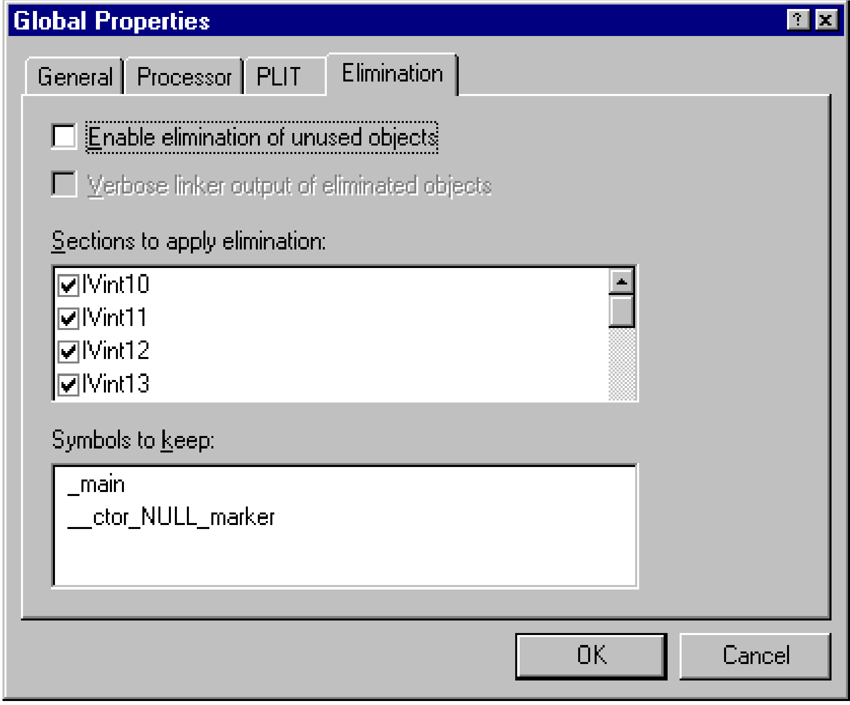


Figure 4-45. Elimination Page of the Global Properties Dialog Box

Choosing **Add Symbol** from the menu invokes the **Add Symbol to Resolve** dialog box (Figure 4-47), which allows 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.

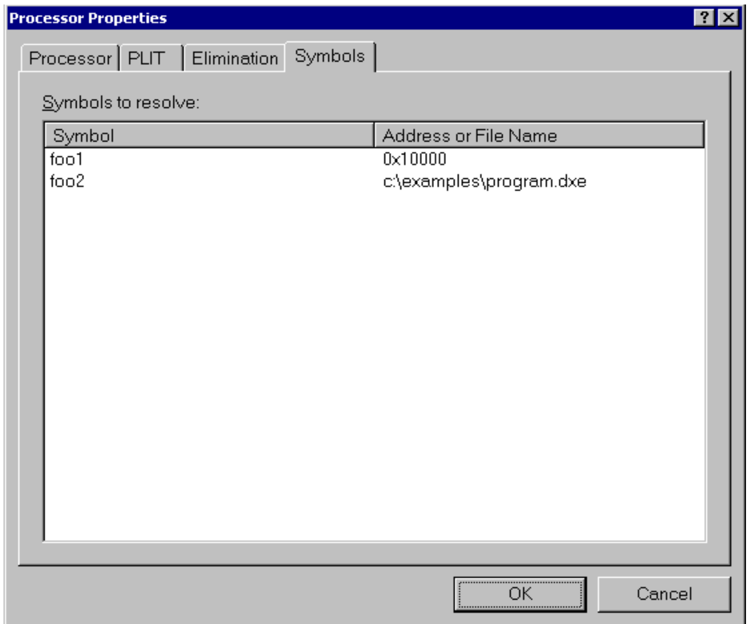


Figure 4-46. Processor Properties Dialog Box – Symbols Tab

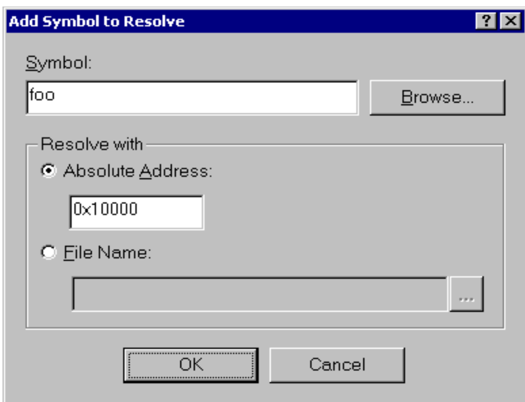


Figure 4-47. Add Symbol to Resolve Dialog Box

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 4-46).
2. Select the symbol you want to delete.
3. Right-click and choose **Remove Symbol**.

Managing Memory Segment Properties

You can specify or 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.

To display the **Memory Segment Properties** dialog box (Figure 4-48):

1. Right-click a memory segment (for example, `PROGRAM` or `MEM_CODE`) in the **Memory Map** pane.

2. Choose **Properties**. The selected segment properties are displayed.

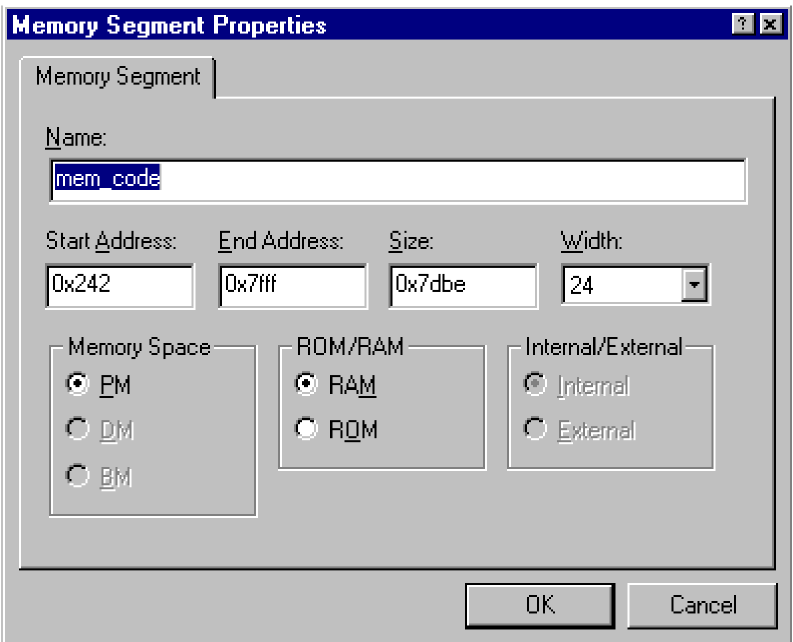


Figure 4-48. Memory Segment Properties Dialog Box

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 (for example, `PROGRAM_DXE` or `CODE_DXE`) in the **Memory Map** pane.
2. Choose **Properties** (Figure 4-49)

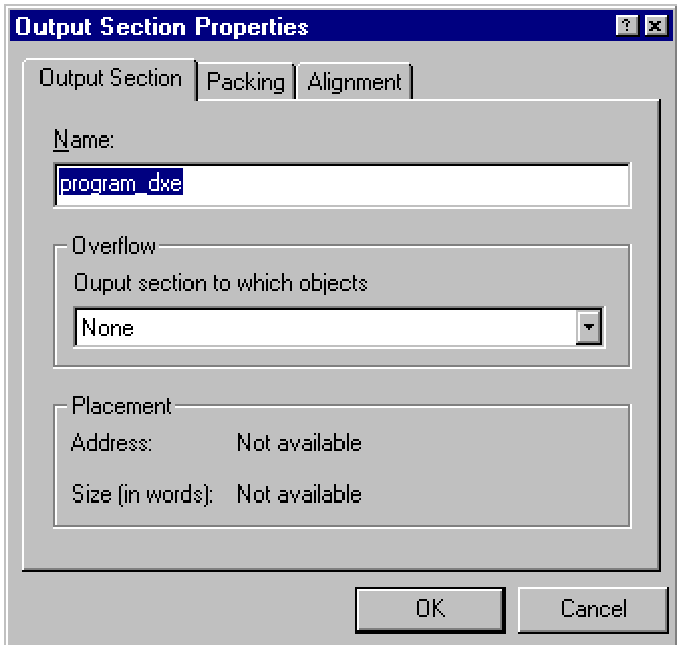


Figure 4-49. Output Section Properties Dialog Box – Output Section Tab

The selections in the output section/segment list includes “None” (for no overflow) and all output sections. You can pin objects to an output section by right-clicking the object and 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 you link the project, the **Placement** box indicates the output section’s address and size as “Not available”. After linking is done, 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.

For Blackfin processors, **No packing** is the only packing method available. For ADSP-21xx DSPs, both **Custom** and **No packing** are 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 4-50).

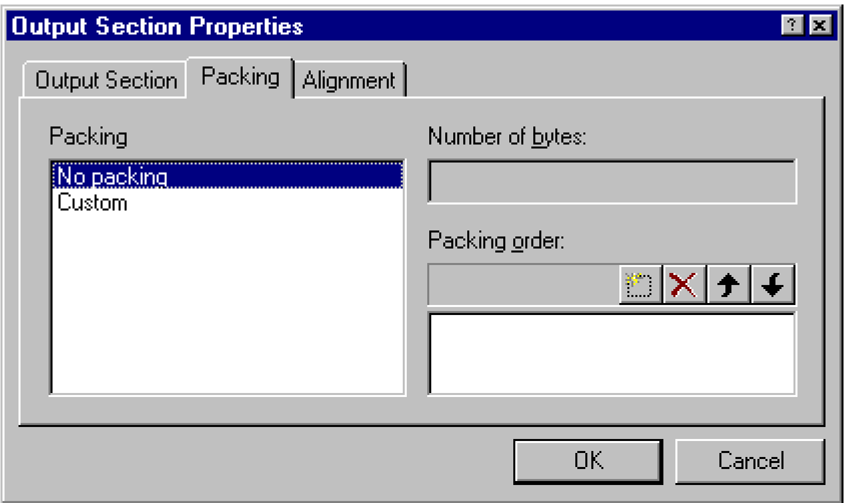


Figure 4-50. Memory Segment Properties Dialog Box – Packing Tab

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 (Figure 4-51).

If you select **No Alignment**, the output section will not be aligned on an address.

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, a gap 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).

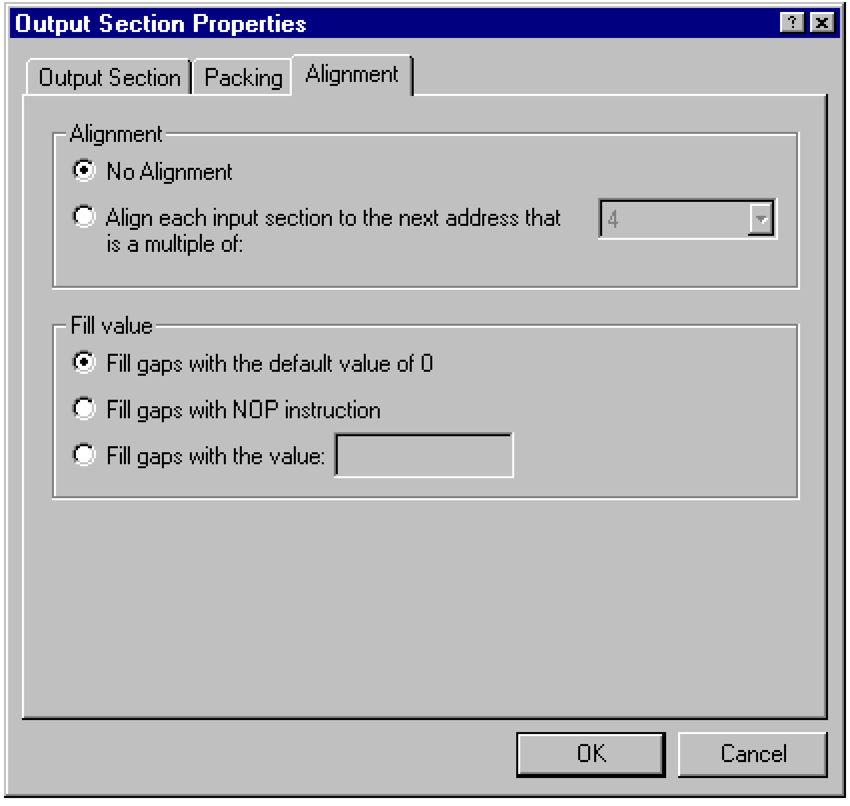


Figure 4-51. Output Section Properties – Alignment Tab

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** and click on the **Overlay** tab (Figure 4-52)

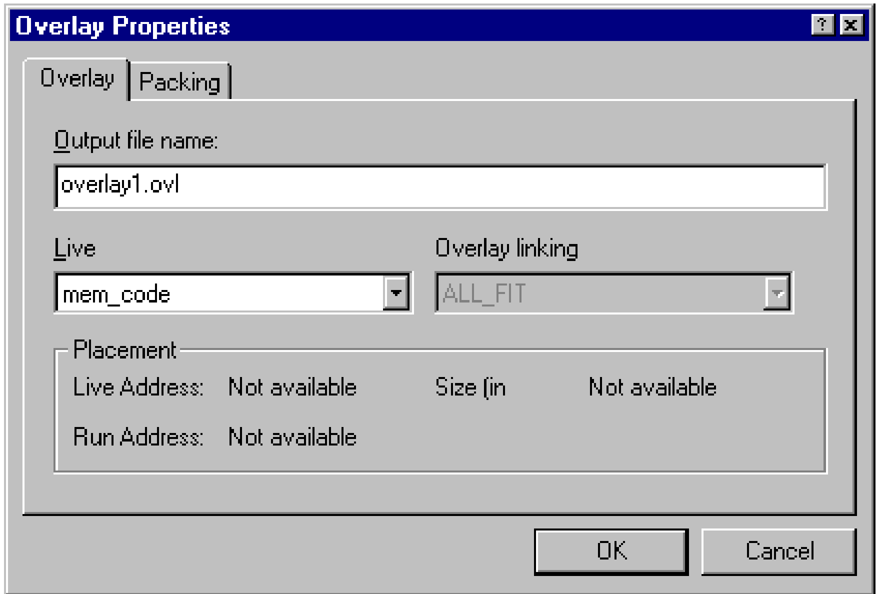


Figure 4-52. Overlay Properties Dialog Box – Overlay Tab

The **Live** memory drop-down list contains all output sections or memory segments within 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 you use 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 Processor Memory

The Expert Linker shows how much space is allocated for your program's heap and stack.

Figure 4-53 shows stack and heap output sections in the **Memory Map** pane. Right-click on either of them to display its properties.

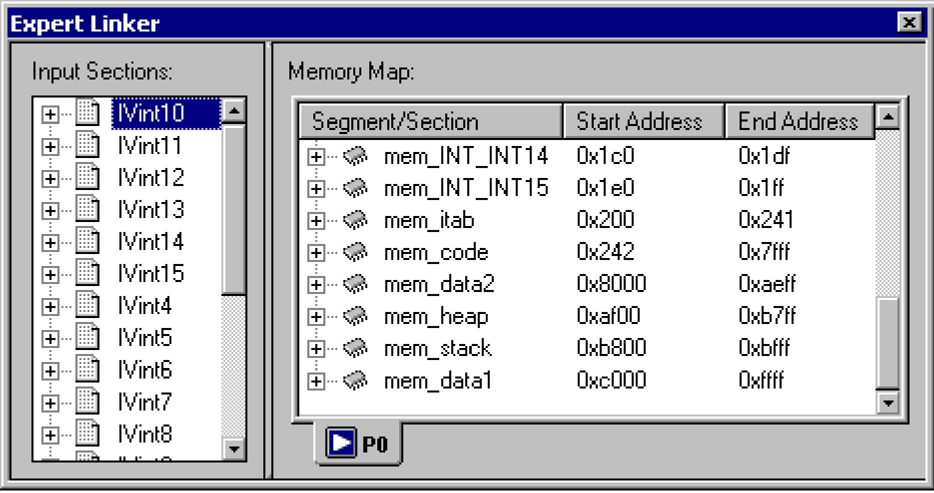


Figure 4-53. Memory Map Window With Stack and Heap Sections

Use the **Global Properties** dialog box to select **Show stack/heap usage** (Figure 4-54). This option graphically displays the stack/heap usage in memory (Figure 4-55).

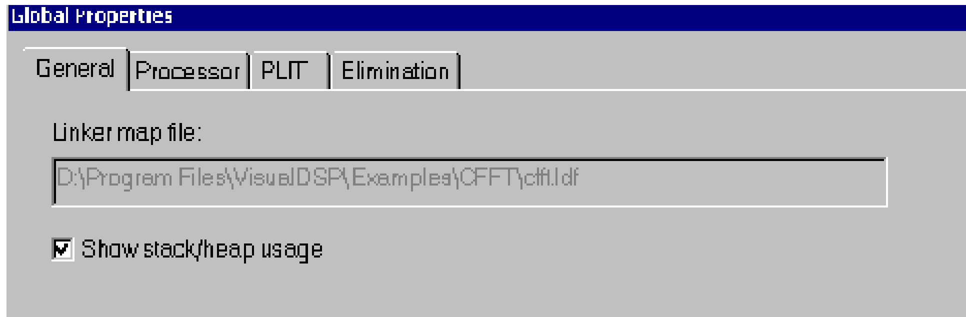


Figure 4-54. Global Properties – Selecting Stack and Heap Usage

The Expert Linker can:

- Locate stacks and 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 action occurs when the target halts after running the program. (assume the unused portion of the stack or heap starts here). 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 and heap memory map blocks. Figure 4-55 shows a possible memory map after running a Blackfin processor project program.

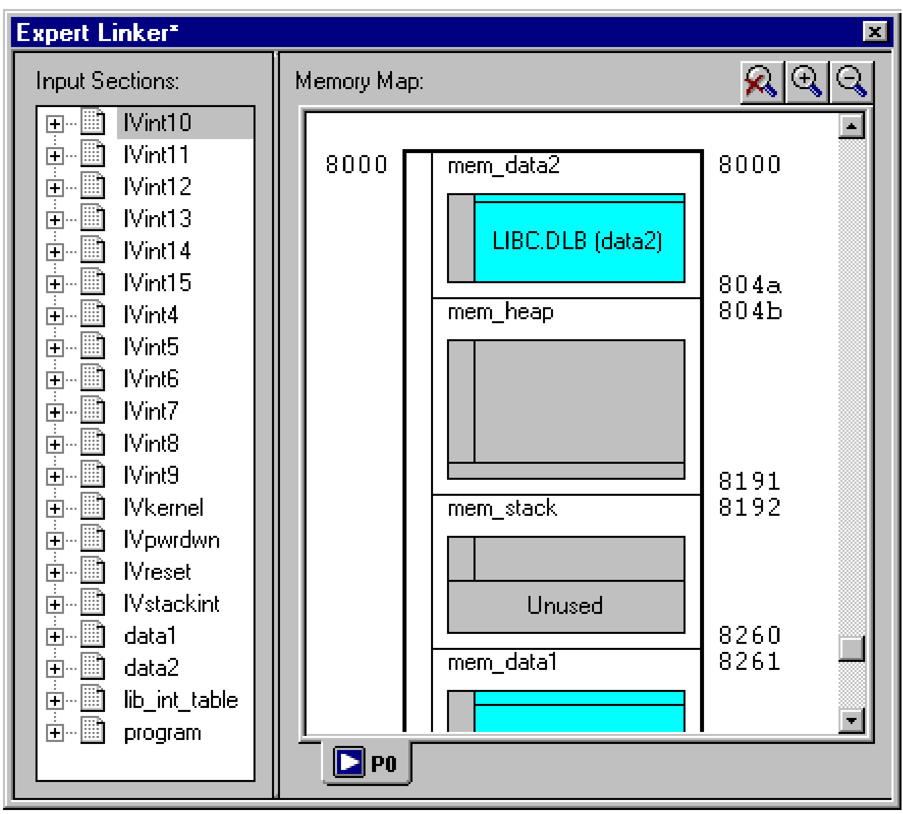


Figure 4-55. Graphical Memory Map Showing Stack and Heap Usage

5 MEMORY OVERLAYS AND ADVANCED LDF COMMANDS

This chapter describes memory management with the overlay functions as well as several advanced LDF commands used for memory management.

This chapter includes:

- "Overview" on page 5-2
  Provides an overview of Analog Devices processor's memory architecture
- "Memory Management Using Overlays" on page 5-4
  Describes memory management using the overlay functions
- "Advanced LDF Commands" on page 5-27
  Describes LDF commands that support memory management with overlay functions, the implementation of physical shared memory, and multiprocessor support

Overview

Current Analog Devices processors use fast Harvard memory architecture that has separate program and data memories. This memory architecture improves processing speed because the machine can fetch program instructions and data in parallel. A data fetch from memory can thus be accomplished in a single memory cycle. Analog Devices DSPs possess a separate program and data memory for speed. For extra speed, data can be kept in program memory as well as in data memory, and there are instructions to fetch data from both memories simultaneously.

To use the full memory bandwidth, Blackfin and ADSP-219x processors possess an instruction cache. Instructions which come from cache memory free up the program memory bus for data access fetch from program memory. This feature permits multiply accumulate instructions to fetch a multiplier and a multiplicand, compute a product, and add the new product to the accumulator, all in a single instruction. In this case, it is important to locate the multiplier data in program memory and the multiplicand data in data memory. It is the programmer's responsibility to assign data buffers to memory. This is done with instructions to the linker. In addition, each DSP project must fit into a different memory arrangement. Different DSP boards have different amounts of memory, located at different addresses.

For example, the ADSP 2181 DSP has the ability to access up to two 8 Kbyte pages of external overlay memory for both program memory and data memory. The DSP still contains 16 Kbyte of program memory and data memory but can access up to another 16 Kbyte of program memory and another 16 Kbyte of data memory.

Blackfin 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 an application efficiently, but they require a user-managed set of DMAs. Since Blackfin

processors do not have a separate built-in overlay manager, 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 the instruction cache.

ADSP-219x DSPs support an external memory interface that allows rather large amounts of memory, but at a penalty in speed. Internal memory is ten times faster than external memory, so it may be desirable to keep large amounts of program code in external memory and to swap parts of it to internal memory for speed in execution. Such a program is said to run in "*overlays*."

For code reuse and portability, a program should not require modification to run in different machines or in different locations in memory. Therefore, the C or assembler source code does not specify the addresses of either code or data. Instead, the source code assigns names to sections of code, data, or both at compile or assembly time to allow the linker to assign physical memory addresses to each section of code or data. The goal is to make the source program's position independent and to let the linker assign all the addresses.

The linker uses Linker Description Files to control "what goes where." At link time, the linker follows directions in the `.LDF` file to place code and data at the proper addresses.

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. 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; for example,

- AN 572 "Overlay Linking on the ADSP-219x"
- EE-152 "Using Software Overlays with the ADSP-219x and VisualDSP 2.0++"
- EE-100 "ADSP-218x External Overlay Memory"

This section describes the use of memory overlays with 16-bit DSPs. The topics are:

- "Introduction to Memory Overlays" on page 5-5
- "Overlay Managers" on page 5-7
- "Memory Overlay Support" on page 5-8
- "Example – Managing Two Overlays" on page 5-12
- "Linker-Generated Constants" on page 5-15
- "Overlay Word Sizes" on page 5-15
- "Storing Overlay ID" on page 5-16
- "Overlay Manager Function Summary" on page 5-17
- "Reducing Overlay Manager Overhead" on page 5-17
- "Using PLIT{} and Overlay Manager" on page 5-22

The following LDF commands facilitate overlay features.

- "OVERLAY_GROUP{}" on page 5-29
- "PLIT{}" on page 5-34

Introduction to 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 function must be done using the memory DMA controller.

Figure 5-1 demonstrates the concept of memory overlays. The two memory spaces are: 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

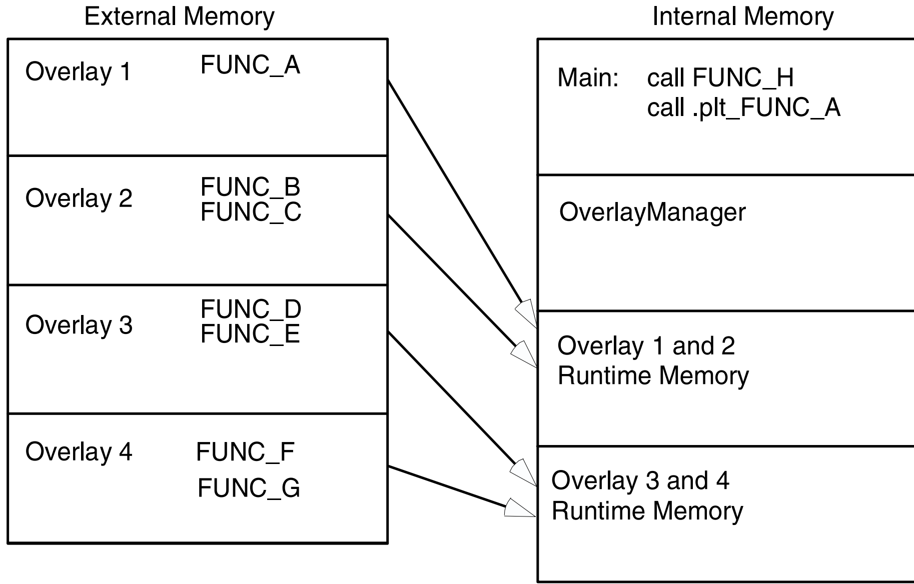


Figure 5-1. Memory Overlays

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.

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 task is accomplished with linker-generated constants and `PLIT{}` commands.

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.

Breakpoints on Overlays

The debugger relies on the presence of the `__ov_start` and`__ov_end` symbols to support breakpoints on overlays.The symbol manager will set a silent breakpoint at each symbol.

The more important of the two symbols is the breakpoint at `_ov_end`. Code execution in the overlay manager should pass through this location once an overlay has been fully swapped in. At this point, the debugger may probe the target to determine which overlays are in context. The symbol manager will now set any breakpoints requested on the overlays and resume execution.

The second breakpoint is at `_ov_start`. The label `_ov_start` should be defined in the overlay manager, in code always executed immediately before the transfer of a new overlay begins. The breakpoint disables all of the overlays in the debugger—the idea being that while the target is running in the overlay manager, the target is “unstable” in the sense that the debugger should not rely on the overlay information it may gather since the target is “in flux”. The debugger will still function without this breakpoint, but there may be some inconsistencies while overlays are being moved in and out.

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 5-1 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 3-33.

In this example, 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 5-1. Overlay Declaration in an .LDF File

```
.dxe_code
  { 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 seg_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_#
_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 that uses 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 by means of 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.

The linker reserves one word (or two bytes in Blackfin processors) at the top of an overlay to house the overlay ID.

Listing 5-2 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 5-2. PLIT Definition in LDF

```
PLIT
{
   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 5-2 on page 5-10. The redirection function calls the PLIT table for overlays 1 and 2 (Table 5-2). For each entry, the linker replaces the generic assembly instructions with specific instructions (where applicable).

Internal Memory

```
Main:                    Plit_table
call.plt_FUNC_A         .plt_FUNC_A:    R0.l = 0x0001;
      .                                 R1.h = 0x0000;
                                        R1.l = 0x2200;
      .                                 jumpOverlayManager;

      .                 .plt_FUNC_B:    R0.l = 0x0001;
call.plt_FUNC_C                         R1.h = 0x0000;
call.plt_FUNC_B                         R1.l = 0x2200;
                                        jumpOverlayManager;

                        .plt_FUNC_C:    R0.l = 0x0002;
                                        R1.h = 0x0000;
                                        R1.l = 0x2300;
                                        jumpOverlayManager;
```

Figure 5-2. Expanded PLIT Table

For example, the first PLIT entry in Listing 5-2 on page 5-10 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 containing 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.

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. These examples are for Blackfin processors.

Listing 5-3. 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.

To verify whether an overlay is in internal memory, place the overlay ID of this overlay into a register (for example, P0 in Blackfin processors) 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.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 moves 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

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 moving the PC to the referenced function. Otherwise, when code is replaced or modified, incorrect code execution may occur. If the program sequencer searches the cache for an instruction and an instruction from the previous overlay is in the cache, that instruction may be executed because the expected cache miss is not received.

Overlay Manager Function 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.

In several circumstances, 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 to complete the transfer 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 and 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 until

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 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 (set for Blackfin processors) 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 whether 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;
     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 the 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. 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 be completed.

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 is completed (if the transfer was necessary). The PC then jumps the run-time location of the function.

The overlay managers in these examples are used universally. Specific applications may require some modifications, which may let you eliminate 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 3-38 for basic syntax information. Refer to "Introduction to Memory Overlays" on page 5-5 for detailed information on overlays.

Figure 5-3 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 these elements are placed into an array or buffer used by an overlay manager.

The two functions in Figure 5-3 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.

```
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 5-3. PLITs and Overlay Memory; main() Calls to Overlays

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 `.plit_X` and `.plit_Y`. Unresolved references to `X` and `Y` are resolved to `.plit_X` and `.plit_Y`.

When the reference and the definition reside in the same executable file, 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 `.plit` command sets up data for the overlay manager, which first loads the overlay that defines the desired symbol, and then branches to that symbol.

Inter-Overlay Calls

PLITs allow you to resolve inter-overlay calls, as shown in Figure 5-4 on page 5-25. Structure the `.LDF` file in a way that ensures the PLIT code generated for inter-overlay function references is part of the `.plit` section for `main()`, which is stored in non-overlay memory.

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

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.

Placing global variables in non-overlay memory optimizes overlays. This action ensures that the proper overlay is loaded before a global variable is called.

Inter-Processor Calls

PLITs resolve inter-processor overlay calls, as shown in Figure 5-5, 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 file that manages the overlay.

Code in Non-Overlay Memory

```
F1:     // function F1 defined
call F2
call F3

/* PLIT & overlay manager handle
calls, using the PLIT for resolving
calls and loading overlays as needed */

.plit_F2:    // set up OM information
        jump OM

.plit_F3:    // set up OM information
        jump OM

/* OM: load overlay defined in setup (from .plt),
   branch to address defined in setup */
```

```
Overlay 1        F2:     // function F2 defined
                 call F1
                 call .plit_F3

Overlay 2        F3:     // function F3 defined
                 call F1
                 call .PLIT_F2
```

Runtime Overlay Memory   `// currently loaded overlay`

Figure 5-4. PLITs and Overlay Memory – Inter-Overlay Calls

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

Not putting global variables in overlays optimizes overlays. This action ensures that the proper overlay is loaded before a global is referenced.

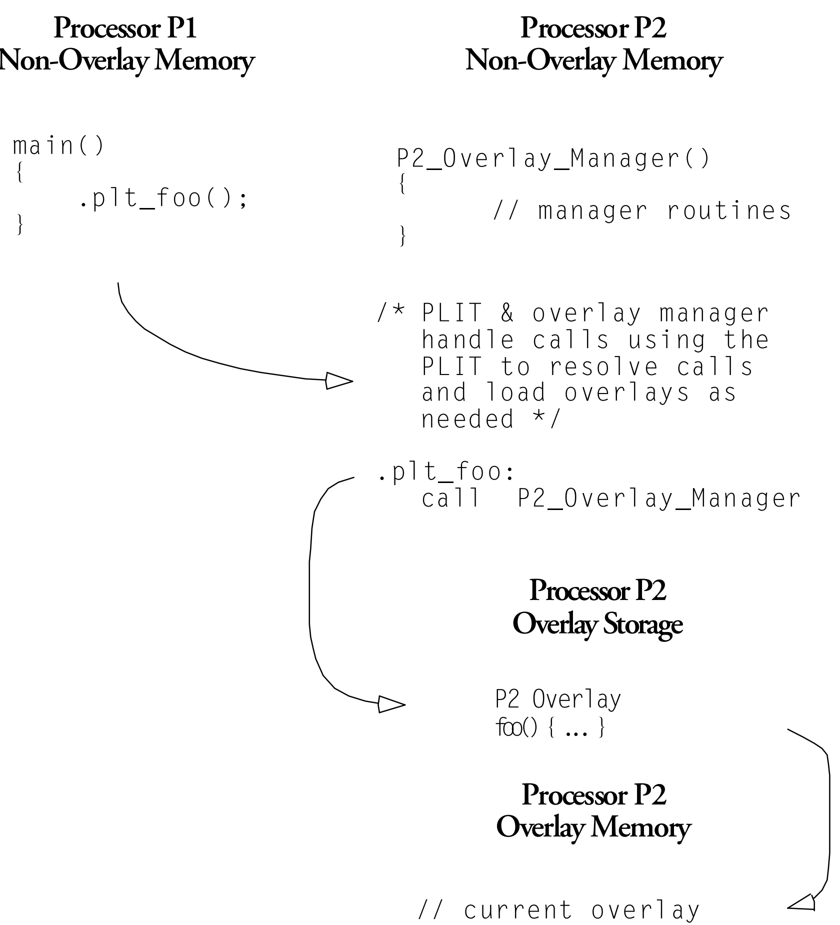


Figure 5-5. PLITs and Overlay Memory – Inter-Processor Calls

Advanced LDF Commands

Commands in the `.LDF` file define the target system and specify the order in which the linker processes output for that system. The LDF commands operate within a scope, which influences the operation of other commands that appear within the range of that scope.

The following LDF commands support advanced memory management functions, overlays, multiprocessor and shared memory features.

- "MPMEMORY{}" on page 5-28
- "OVERLAY_GROUP{}" on page 5-29
- "PLIT{}" on page 5-34
- "SHARED_MEMORY{}" on page 5-38

For detailed information on other LDF commands, refer to "LDF Commands" on page 3-23.

MPMEMORY{}

The MPMEMORY{} command is used with DSPs that implement physical shared memory, such as Blackfin processors and ADSP-2192-12 DSPs.

The MPMEMORY{} command specifies the offset of each processor's physical memory in a multiprocessor target system. After you declare 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 5-8 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 you can declare is architecture-specific. Follow the MPMEMORY{} command with PROCESSOR *processor_name*{} commands, which contain each processor's MEMORY{} and SECTIONS{} commands.

Figure 5-6 shows MPMEMORY{} command syntax.

```
MPMEMORY{shared_segment_commands}

    processor_name {
                      START(address_number)
                    }
```

Figure 5-6. 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 and linker labels follow the same rules. For more information, refer to "LDF Expressions" on page 3-13.

- *`processor_name{placement_commands}`* – Applies the *`processor_name`* offset for multiprocessor linking. Refer to "PROCESSOR{}" on page 3-39 for more information.

The `MEMORY{}` command specifies the memory map for the target system. 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. You can declare unlimited number of memory segments within each `MEMORY{}` command. For more information, see "MEMORY{}" on page 3-29.

OVERLAY_GROUP{}

The `OVERLAY_GROUP{}` command provides legacy support. This command is deprecated and is not recommended for use. When you run the linker, the following warning may occur.

```
[Warning li2534] 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 5-8 for a detailed description of overlay functionality.

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

Overlay declarations syntactically resemble the 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 5-7 demonstrates grouped overlays.

In the simplified examples in Listing 5-4 and Listing 5-5, 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 being executed in run-time memory, which is located in the program memory segment.

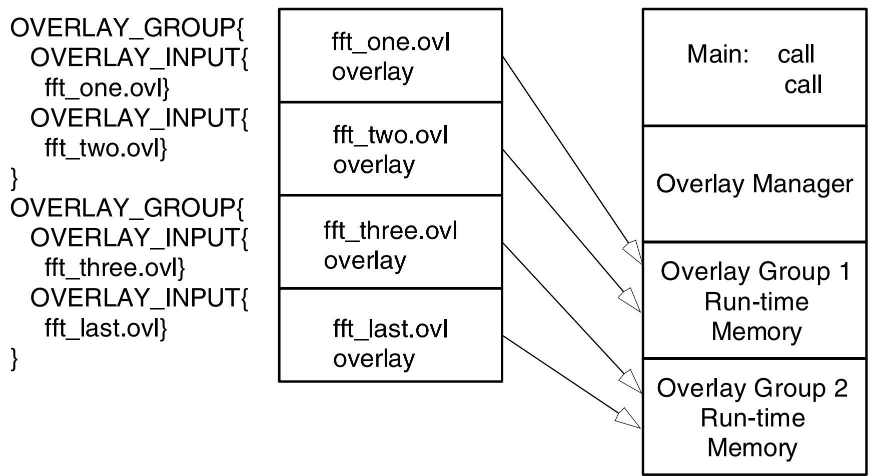


Figure 5-7. Example of Overlays – Grouped

Ungrouped Overlay Execution

In Listing 5-4, as the FFT progresses and overlay functions are called in turn, they are brought into run-time memory in sequence as four function transfers. Figure 5-8 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 5-4. 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.
```

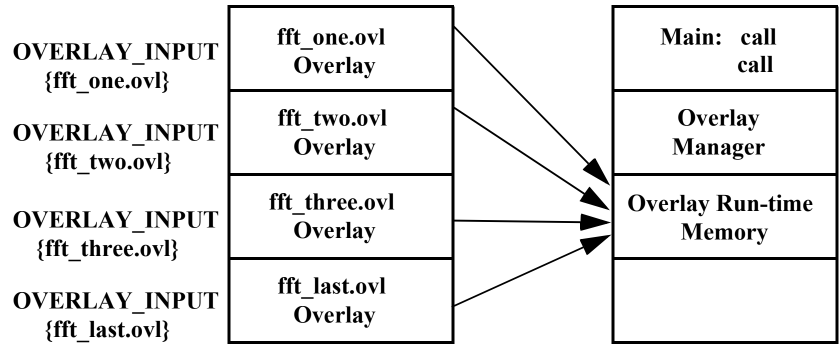


Figure 5-8. Example of Overlays – Not Grouped

```
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 5-5 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.

Listing 5-5. 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 task 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 file, which has the section name `.PLIT`.

The `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 are executed each time a call to a function in that overlay is detected.

The `PLIT{}` command provides 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 5-8 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 5-22.

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 5-9 shows the general syntax of the `PLIT{}` command and indicates how the linker handles a symbol (`symbol`) local to an overlay function.

PLIT{*plit_commands*}

instruction
symbol = PLIT_SYMBOL_OVERLAYID [*symbol*]
symbol = PLIT_SYMBOL_ADDRESS

Figure 5-9. Syntax Tree of the PLIT{} Command

Parts of the `PLIT{}` command are:

- *`instruction`* – 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 *symbol* 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 the sequence of assembly code in each *`plit_command`*. 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 process is controlled by 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.

- No input section is 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 according to the 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 Example

This is an 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 rule, minimize overlay transfer traffic. Improve performance by designing code to ensure 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, to ensure 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.

SHARED_MEMORY{}

The ADSP-2192 DSP is the only ADSP-21xx DSP that supports `SHARED_MEMORY{}`. All Blackfin processors support this function.

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. The `SHARED_MEMORY{}` command is used to produce `.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 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.

Figure 5-10 shows the syntax for the SHARED_MEMORY{} command, followed by definitions of its components.

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

Figure 5-10. SHARED_MEMORY{} Command Syntax

The command components are:

- OUTPUT() – Specifies the output file name (*file_name*.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 5-11 shows the scope of SHARED_MEMORY{} commands in the LDF.

```
The MEMORY{} command appears in a scope that is available to
any SHARED_MEMORY{} command or PROCESSOR{} command that uses
the shared memory. To achieve this type of scoping across
multiple links, place the shared MEMORY{} command in a
separate LDF and use the INCLUDE() command to include that
memory in both links.

MEMORY
  {
    my_shared_ram
      {
        TYPE(RAM) START(0x04000000) LENGTH(8k) WIDTH(32)
      }
  }
SHARED_MEMORY
  {

    OUTPUT(shared.sm)

    SECTIONS
      {
        my_shared_sections{section_commands}
          > my_shared_ram
      }
  }
PROCESSOR p0{
  processor_commands with link against shared memory}
PROCESSOR p1{
  processor_commands with link against shared memory}
```

Figure 5-11. LDF Scopes for SHARED_MEMORY{}

6 ARCHIVER

The VisualDSP++ archiver, `elfar.exe`, combines object files (`.DOJ`) 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:

- "Archiver Guide" on page 6-2
  Introduces the archiver's functions
- "Archiver Command-Line Reference" on page 6-11
  Describes archiver operations by means of command-line switches

You can run the archiver 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) to produce 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
- Allows embedded version information into a library built with `elfar`

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 a general-purpose utility. 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 (as shown in "Archiver Command-Line Reference" on page 6-11).

To maintain code consistency, use the conventions in Table 6-1.

When you create a library, VisualDSP++ writes *`<projectname>.DLB`*.

Table 6-1. File Name Extensions used with Archiver

Extension	File Description
`.DLB`	Library file
`.DOJ`	Object file. Input to archiver.
`.TXT`	Text file used as input with the `-i` 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 (or function) in code 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++" on page 6-3). When you build 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 you do not have to enter all the file names, just the library name.

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 `.LDF` 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.

Tagging an Archive with Version Information

The archiver supports embedding version information into a library built with `elfar`.

Basic Version Information

You can "tag" an archive with a version. The easiest way to tag an archive is using the `-t` switch (see Table 6-2 on page 6-12) which takes an argument (the version number). For example,

```
elfar -t 1.2.3 lib.dlb
```

The `-t` switch can be used in addition to any other `elfar` switch. For example, a version can be assigned at the same time that a library is created:

```
elfar -c -t "Steve's sandbox Rev 1" lib.dlb *.doj
```

To hold version information, the archiver creates an object file, `__version.doj`, that have version information in the `.strtab` section. This file is not made visible to the user.

An archive without version information will not have the __version.doj entry. The only operations on the archive using elfar that will add version information are those that use the -t switch. That is, an archive without version information will not pick up version information unless you specifically request it.

If an archive contains version information (__version.doj is present), all operations on the archive preserve that version information, except operations that explicitly request version information to be stripped from the archive (see "Removing Version Information from an Archive" on page 6-9).

If an archive contains version information, that information can be printed with the -p command.

```
elfar -p lib.dlb
::User Archive Version Info: Steve's sandbox Rev 1
a.doj
b.doj
```

To highlight the version information, precede it with "::".

User-Defined Version Information

You can provide any number of user-defined version values by supplying a text with those values. The text file can have any number of entries. Each line in the file begins with a name (a single token, for example, not-embedded white space), followed by a space and then the value associated with that name. As an example, consider the file foo.txt:

```
my_name neo
my_location zion
CVS_TAG matrix_v_8_0
other version value can be many words; name is only one
```

This file defines four version names: my_name, my_location, CVS_TAG, and other. The value of my_name is neo; the value of other is "version value can be many words; name is only one".

To tag an archive with version information from a file, use the `-tx` switch (see Table 6-2 on page 6-12) which accepts the name of that file as an argument:

```
elfar -c -tx foo.txt lib.dlb object.doj
elfar -p lib.dlb
::CVS_TAG matrix_v_8_0
::my_location zion
::my_name neo
::other version value can be many words; name is only one
```

You can add version information to an archive that already has version information. The effect is additive. Version information already in the archive is carried forward. Version information that is given new values is assigned the new values. New version information is added to the archive without destroying existing information.

Printing Version Information

As mentioned above, when printing the contents of an archive, the `-p` command (see Table 6-2 on page 6-12) prints any version information. Two more forms of the `-p` switch can be used to examine version information.

The `-pv` switch prints only version information, and does not print the contents of the archive. This switch provides a quick way to check the version of an archive.

The `-pva` switch prints all version information. Version names without values are not be printed with `-p` or `-pv` but are shown with `-pva`. In addition, the archiver keeps two additional kinds of information:

```
elfar -a lib.dlb t*.doj
elfar -pva lib.dlb
::User Archive Version Info: 1.2.3
::elfar Version: 4.5.0.2
::__log: -a lib.dlb t*.doj
```

The archiver version that created the archive is stored in `__version.doj` and is available using the `-pva` switch. Also, if any operations that caused the archive to be written have been executed since the last version information was written, these commands appear as part of special version information called "`__log`". The log prints a line for every command that has been done on the archive since the last update of the version information.

Removing Version Information from an Archive

Every operation has a special form of switch that can cause an archive to be written and request that the version information is not written to the archive. Version information already in the archive would be lost. Adding "`nv`" (no version) to a command strips version information. For example,

```
elfar -anv lib.dlb new.doj
elfar -dnv lib.dlb *
```

In addition, a special form of the `-t` switch (see Table 6-2 on page 6-12), which take no argument, can be used for stripping version information from an archive:

```
elfar -tnv lib.dlb   // only effect is to remove version info
```

Checking Version Number

You can have version numbers conform to a strict format. The archiver will confirm that version numbers given on the command line conform to a `nn.nn.nn` format (three numbers separated by "."). The `-twc` switch (see Table 6-2 on page 6-12) causes the archiver to raise a warning if the version number is not in this form. The check ensures that the version number starts with a number in that format.

Adding Text to Version Information

You can add additional text to the end of the version information.

For example,

```
elfar -twc "1.2 new library" lib.dlb
[Warning ar0081] Version number does not match num.num.num format
      Version 0.0.0 will be used.
elfar -pv lib.dlb
::User Archive Version Info: 0.0.0 1.2 new library
```

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"
- "Archiver Parameters and Switches"
- "Command-Line Constraints"
- "Archiver Symbol Name Encryption"

elfar Command Syntax

Use the following syntax to run elfar from the command line.

```
elfar -[a|c|d|e|p|r] <options> library_file object_file ...
```

Table 6-2 on page 6-12 describes each switch.

Example:

```
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 6-1 on page 6-3 lists typical file types, file names, and extensions.

Symbol Encryption:

When using symbol encryption, use the following syntax.

```
elfar -s [-v] library_file in_library_file exclude_file  type-letter
```

Refer to "Archiver Symbol Name Encryption" on page 6-15 for more information.

Archiver Parameters and Switches

Table 6-2 describes each archiver part of the command. Switches must appear before the name of the archive file.

Table 6-2. Command-Line Options and Entries

Item	Description
lib_file	Specifies the library that the archiver modifies. This parameter appears after the switch.
obj_file	Identifies one or more object files that the archiver uses when modifying the library. This parameter must appear after *lib_file*. 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
-anv	Appends one or more object files and clears version information
-c	Creates a new *lib_file* containing the listed object files
-d	Removes the listed *object files* from the specified *lib_file*
-dnv	Removes the listed *obj_file(s)* from the specified *lib_file.* and clears version information
-e	Extracts the specified file(s) from the library
-i *filename*	Uses *filename*, a list of object files, as input. This file lists *obj_file(s)* to add or modify in the specified *lib_file* (.DLB).
-M	Prints dependencies. Available only with the -c switch.
-MM	Prints dependencies and creates the library. Available only with the -c switch.

Table 6-2. Command-Line Options and Entries (Cont'd)

Item	Description
-p	Prints a list of the *obj_file*(s) (.DOJ) in the selected *lib_file* (.DLB) to standard output.
-pv	Prints only version information in library to standard output
-pva	Prints all version information in library 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" on page 6-15.
-t *verno*	Tags the library with version information in string
-tx *filename*	Tags the library with version information in the file
-twc *ver*	Tags the library with version information in the num.num.num form
-tnv	Clears version information from a library
-v	(Verbose) Outputs status information as the archiver processes files.
-version	Prints the archiver (elfar) version to standard output
-w	Removes all archiver-generated warnings
-Wnnnn	Selectively disables warnings specified by one or more message numbers. For example, -W0023 disables warning message ar0023.

The elfar utility enables you to specify files in an archive by using the wildcard character '*'. For example, the following commands are valid:

```
elfar -c lib.dlb *.doj   // create using every .doj file
elfar -a lib.dlb s*.doj  // add objects starting with 's'
elfar -p lib.dlb *1*     // print the files with '1' in their name
elfar -e lib.dlb *       // extract all files from the archive
elfar -d lib.dlb t*.doj  // delete .doj files starting with 't'
elfar -r lib.dlb *.doj   // replace all .doj files
```

The -c, -a, and -r switches use the wildcard to look up the filenames in the file system. The -p, -e, and -d switches use the wildcard to match file names in the archive.

Command-Line 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. The -i and -v switches are not operational switches, and can appear later.
- When using the archiver's -p switch, you do not need to identify members on the command line.
- 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.
- Wildcard options are supported with the use of the wildcard character "*").

- The *obj_file* name (.DOJ object file) can be added, removed, or replaced in the *lib_file*.
- The archiver's command line is *not* case sensitive.

Archiver Symbol Name Encryption

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.

The -s switch (see in Table 6-2) is used to encrypt symbols in <in_library_file> to produce <library_file>. Symbols in <exclude_file> are not encrypted, and <type-letter> provides the first letter of scrambled names.

Command Syntax

The following command line encrypts symbols in an existing archive file.

```
elfar -s [-v] library_file in_library_file exclude_file  type-letter
```

where:

-s – Selects the encryption operation.

-v – Selects verbose mode, which provides statistics on the symbols that were encrypted.

library_file – Specifies the name of the library file (.DLB) to be produced by the encryption process

in_library_file – 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.

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.

Encryption 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" would not be shown by the debugger, and "._foo.end" would 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 is 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 are 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, unrelated external symbols of the same length may be encrypted identically.

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 subtypes[1] 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++ 3.5 C/C++ Compiler and Library Manual* for appropriate target architectures.

[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++ 3.5 Assembler and Preprocessor Manual* for appropriate target architectures.

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

in a 16-bit wide buffer, the assembler loads the equivalent 8-bit value into the most significant 8 bits in the 8-bit memory location and zero-fills the lower eight bits.

Table A-1. Numeric Formats

Convention	Description
0x*number* H#*number* h#*number*	Hexadecimal number.
number D#*number* d#*number*	Decimal number.
B#*number* b#*number*	Binary number.
O#*number* o#*number*	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 For information on the assembler and preprocessor, see the *VisualDSP++ 3.5 Assembler and Preprocessor Manual* for appropriate target architectures.

Linker Description Files (.LDF)

Linker Description Files (.LDF) are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language, see "LDF Commands" on page 3-23.

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 2-30.

Build Files

Build files are produced by the VisualDSP++ development tools when you build 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 (.XML)" 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 in 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 in 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 (.XML)

The linker can output memory map 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, its output can be an ASCII format file. ASCII-format files are text representations of ROM memory images that you can use in post-processing. For more information, refer to no-boot mode information in the *VisualDSP++ Loader Manual for 16-Bit Processors*.

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 (`.DXE`, `.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:

- The hexadecimal number which is converted to binary
- The number of binary bits read which 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 or 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

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. This appendix describes the ELF file dumper utility.

elfdump – ELF File Dumper

The ELF file dumper (`elfdump.exe`) utility extracts data from ELF-format executable files (`.DXE`) and yields a text showing the ELF file's contents.

The `elfdump` utility 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
`-arall`	Prints every library member
`-help`	Prints the list of elfdump switches to stdout
`-ph`	Prints the program header table

Table B-1. ELF File Dumper Command-Line Switches (Cont'd)

Switch	Description
`-sh`	Prints the section header table. This switch is the default when no options are specified.
`-notes`	Prints note segment(s)
`-n` *name*	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. This is same as `-fh -ph -sh -notes -n '*'`.
`-ost`	Omits string table sections
`-v`	Prints version information
objectfile	Specifies 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`. The `-n` and `-i` options can have a modifier letter after the main option character to force section contents to be formatted as: `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 to 32, 32 is used `hN` Dumps contents in hex, N bytes per group `HN` Dumps contents in hex, (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 6-1.

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 are 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.

C LDF PROGRAMMING EXAMPLES FOR BLACKFIN PROCESSORS

This appendix provides several typical LDFs. used with Blackfin processors. As you modify these examples, refer to the syntax descriptions in "LDF Commands" on page 3-23.

This appendix provides the following examples.

- "Linking for a Single-Processor System" on page C-2
- "Linking Large Uninitialized or Zero-initialized Variables" on page C-4
- "Linking for Assembly Source File" on page C-6
- "Linking for C Source File – Example 1" on page C-8
- "Linking for Complex C Source File – Example 2" on page C-11
- "Linking for Overlay Memory" on page C-17

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 you link an executable file for a single-processor system, the .LDF file describes the processor's memory and places code for that processor. The .LDF file in Listing C-1 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
- $OBJS and $LIBS macros retrieve 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 by using the memory definitions

Listing C-1. 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,
libcpprt_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 or Zero-initialized Variables

When linking an executable file that contains large uninitialized variables, use the NO_INIT (equivalent to SHT_NOBITS legacy qualifier) or ZERO_INIT section qualifier to reduce the file size.

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 action can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur long download times when used with an emulator. This situation also may occur when you boot from a loader file (because of the increased file size). Listing C-2 shows an example of assembly source code. Listing C-3 shows the use of the NO_INIT and ZERO_INIT sections to avoid initialization of a segment.

The LDF can omit an output section from the output file. The NO_INIT qualifier directs the linker to omit data for that section from the output file.

Refer to "SECTIONS{}" on page 3-42 for more information on the NO_INIT and ZERO_INIT section qualifiers.

The NO_INIT qualifier corresponds to the /UNINIT segment qualifier in previous (.ACH) development tools. Even if you do not use NO_INIT, 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 action reduces the loader's output file size, but still requires execution time for the processor to initialize the memory with zeros.

Listing C-2. Large Uninitialized Variables: Assembly Source

```
.SECTION  extram_area;               /* 1Mx8 EXTRAM */
.BYTE     huge_buffer[0x006000];
.SECTION  zero_extram_area;
.BYTE     huge_zero_buffer[0x006000];
```

Listing C-3. 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 )
           // NO_INIT section isn't written to the output file
   SECTION {
      extram_ouput NO_INIT {
        INPUT_SECTIONS( $OBJECTS ( extram_area ) )}
      >mem_extram;
   SECTION {
      zero_extram_output ZERO_INIT {
         INPUT_SECTIONS ( $OBJECTS ( zero_extram_area ) )}
      >mem_extram;
      }   // end section
   }      // end processor P0
```

Linking for Assembly Source File

Listing C-5 shows an example .LDF file for an ADSP-BF535 DSP that describes a simple memory placement of an assembly source file. The file in Listing C-4 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 3-1 on page 3-4 for information on items in this basic example.

Listing C-4. 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 C-5. 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

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))
        >L2SRAM
        }   // end section
    }       // end processor P0
```

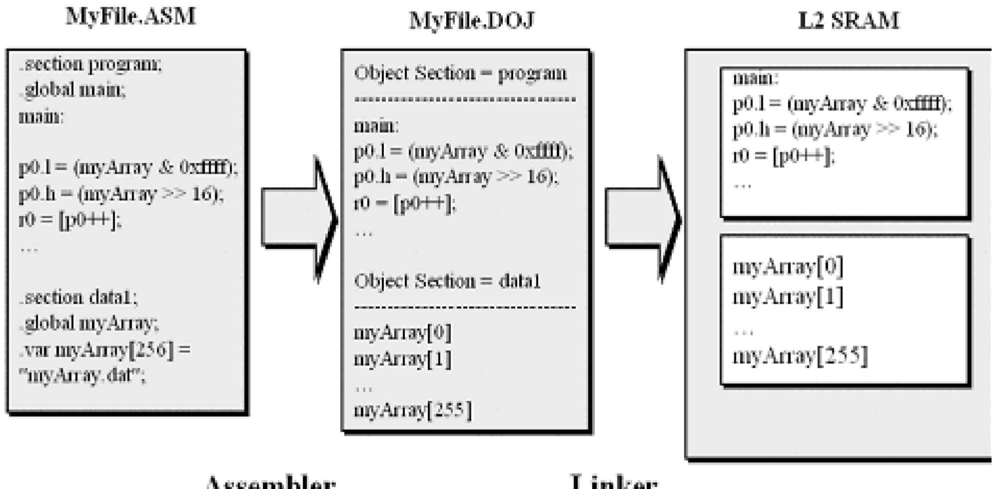


Figure C-1. Assembly-to-Memory Code Placement

Linking for C Source File – Example 1

Listing C-7 shows an example LDF that describes the memory placement of a simple C source file (Listing C-6) 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 3-1 on page 3-4 for information on items in this basic example.

Listing C-6. 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 C-7. 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(cplb) $LIBRARIES(cplb))
  INPUT_SECTIONS( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
  INPUT_SECTIONS( $OBJECTS(cplb_data) $LIBRARIES(cplb_data))
  // Align L2 constructor data segments on a 1-byte boundary
  // (C++ only)
  INPUT_SECTION_ALIGN(1)
  INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
 }>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. This file contains code and data that is to reside in, and execute from, L1, L2, Scratchpad SRAM, and 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 3-1 on page 3-4 for information on items in this complex example.

Listing C-8. 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;

...
}
```

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. Also see Figure C-2 on page C-16.

Listing C-9. 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 libcpprt535.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
      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{}
```

Linking for Complex C Source File – Example 2

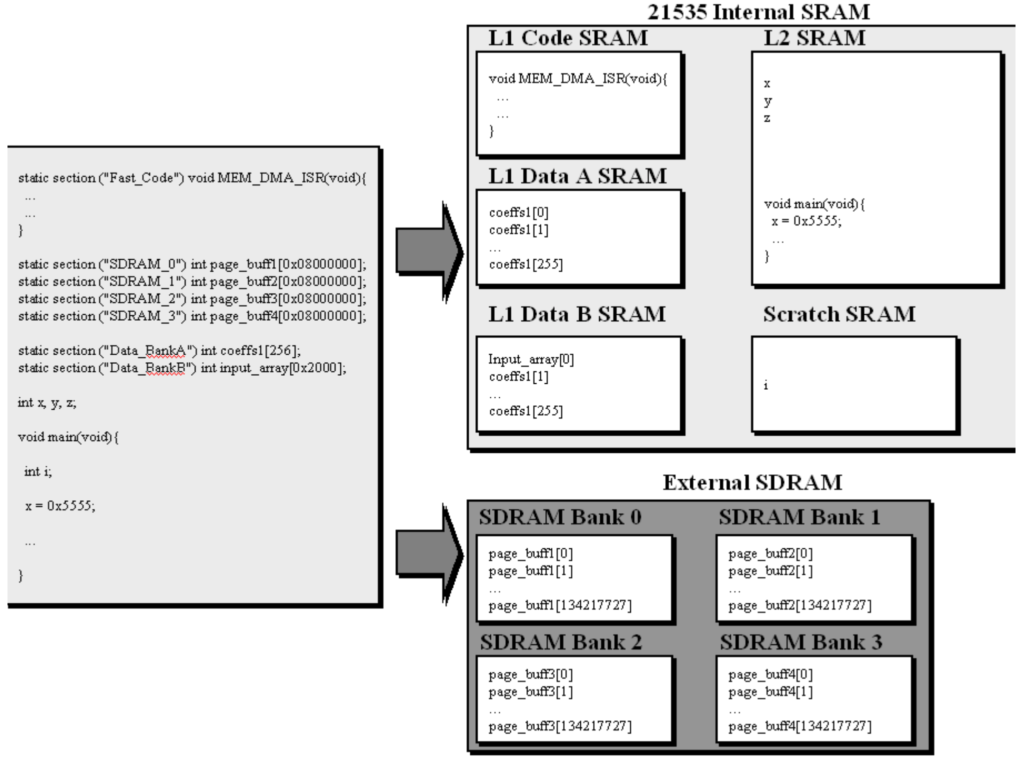


Figure C-2. C-to-Memory Code Placement

Linking for Overlay Memory

When you link 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 C-10 shows an example .LDF file for an overlay-memory system. For more information on this .LDF file, see the comments in the listing.

Listing C-10. 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
    {
        dxe_reset { INPUT_SECTIONS($OBJECTS(IVreset))
        } >MEM_PROGRAM
        dxe_itab  { INPUT_SECTIONS($OBJECTS(IVpwrdwn))

   // 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

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
}
```

D LDF PROGRAMMING EXAMPLES FOR ADSP-21XX DSPS

This appendix provides several typical LDFs used with ADSP-218x and ADSP-219x DSPs. As you modify these examples, refer to the syntax descriptions in "LDF Commands" on page 3-23.

This appendix provides the following examples:

- "Linking for a Single-Processor ADSP-219x System" on page D-3
- "Linking Large Uninitialized or Zero-initialized Variables" on page D-5
- "Linking an Assembly Source File" on page D-7
- "Linking a Simple C-Based Source File" on page D-9
- "Linking Overlay Memory for an ADSP-2191 System" on page D-16
- "Linking an ADSP-219x MP System With Shared Memory" on page D-19
- "Overlays Used With ADSP-218x DSPs" on page D-23

The source code for several programs is bundled with your development software. Each program includes an `.LDF` file. For working examples of the linking process, examine the `.LDF` files that come with the examples loacted in the following directories.

```
<VisualDSP++ InstallPath>\218x\Examples
```

```
<VisualDSP++ InstallPath>\219x\Examples
```

A variety of per-processor default `.LDF` files that come with the development software provide an example LDF for each processor's internal memory architecture. Default LDFs are in the following directories.

```
<VisualDSP++ InstallPath>\218x\ldf
```

```
<VisualDSP++ InstallPath>\219x\ldf
```

Linking for a Single-Processor ADSP-219x System

When you link an executable for a single-processor system, the LDF describes the processor's memory and places code for that processor. The LDF in Listing D-1 shows a single-processor LDF. Note the following commands in this LDF.

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

Listing D-1. Single-Processor System LDF Example

```
ARCHITECTURE(ADSP-219x)

SEARCH_DIR ( $ADI_DSP\219x\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 219x\lib relative to the install directory
```

```
LIBS libc.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 libraries (.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("219x_memory.ldf") }

   SECTIONS                // Specify the output sections
   {
        INCLUDE( "219x_sections.ldf" )
   }                       // end P0 sections
}                          // end P0 processor
```

Linking Large Uninitialized or Zero-initialized Variables

When linking an executable file that contains large uninitialized variables, use the NO_INIT (equivalent to SHT_NOBITS legacy qualifier) or ZERO_INIT section qualifier to reduce the file size.

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 action can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur long download times when used with an emulator. This situation also may occur when you boot from a loader file (because of the increased file size). Listing D-2 shows an example of assembly source code. Listing D-3 shows the use of the NO_INIT and ZERO_INIT sections to avoid initialization of a segment.

The LDF can omit an output section from the output file. The NO_INIT qualifier directs the linker to omit data for that section from the output file.

Refer to "SECTIONS{}" on page 3-42 for more information on the NO_INIT and ZERO_INIT section qualifiers.

The NO_INIT qualifier corresponds to the /UNINIT segment qualifier in previous (.ACH) development tools. Even if you do not use NO_INIT, 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 action reduces the loader's output file size, but still requires execution time for the processor to initialize the memory with zeros.

Listing D-2. Large Uninitialized Variables: Assembly Source

```
.SECTION/DATA extram_area;          /* 1Mx16 EXTRAM */
.VAR          huge_buffer[0x006000];
.SECTION      zero_extram_area;
.VAR          huge_zero_buffer[0x006000];
```

Listing D-3. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-219x)
$OBJECTS = $COMMAND_LINE_OBJECTS;   // Libraries & objects from
                                    // the command line
MEMORY {
   mem_extram {
      TYPE(DM RAM) START(0x10000) END(0x15fff) WIDTH(16)
      }   // end segment
   }      // end memory

PROCESSOR P0 {
   LINK_AGAINST ( $COMMAND_LINE_LINK_AGAINST )
   OUTPUT ( $COMMAND_LINE_OUTPUT_FILE )
          // NO_INIT section is not written to output file
   SECTION {
      extram_output NO_INIT {
        INPUT_SECTIONS ( $OBJECTS ( extram_area ) )}
      >mem_extram;
   SECTION {
      zero_extram_output ZERO_INIT {
        INPUT_SECTIONS ( $OBJECTS ( zero_extram_area ) )}
      >mem_extram;
      }   // end section
   }      // end processor P0
```

Linking an Assembly Source File

Listing D-5 shows an example .LDF file (for an ADSP-2191 DSP) that describes a simple memory placement of an assembly source file (Listing D-4). The LDF file includes two commands, MEMORY and SECTIONS, which describe specific memory and system information. Arrays x_input and y_input are stored in two different memory blocks to take advantage of the ADSP-2191's Harvard architecture.

Listing D-4. MyFile.ASM

```
.SECTION/CODE program;
.GLOBAL _main;
_main:
I2 = x_input;
L2 = 0;                 /* linear buffer  */
M0 = 1;
I6 = y_input;
AX0 = I6;
reg(B6) = AX0;          /* circular buffer */
L6 = length(y_input);
M6 = 1;
AX0 = DM(I2+=M0), AY1 = PM(I6+=M6);
...
.SECTION/DATA data1;
.VAR x_input[256];
.SECTION/DATA data2;
.VAR/CIRC y_input[256] = "myinput.dat";
```

Notice the data2 section and the use of uppercase keywords.

Listing D-5. Simple LDF Based on Assembly Source File Only

```
ARCHITECTURE(ADSP-2191)
// Libraries from the command line are included
// in COMMAND_LINE_OBJECTS.
$OBJECTS =   $COMMAND_LINE_OBJECTS;

MEMORY
{
mem_code { TYPE(PM RAM) START(0x000000) END(0x007fff) WIDTH(24) }
mem_data1{ TYPE(DM RAM) START(0x008000) END(0x00bfff) WIDTH(16) }
mem_data2{ TYPE(DM RAM) START(0x00c000) END(0x00ffff) WIDTH(16) }
}
PROCESSOR p0
{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
   SECTIONS
   {
      sec_code {
         INPUT_SECTIONS( $OBJECTS(program)  )
      } > mem_code

      sec_data1 {
         INPUT_SECTIONS( $OBJECTS(data1) )
      } > mem_data1

      sec_data2 {
         INPUT_SECTIONS( $OBJECTS(data2) )
      } > mem_data2

   }   // SECTIONS
       // PROCESSOR p0
```

Linking a Simple C-Based Source File

Listing D-7 shows an example .LDF file that describes the memory placement of a simple C source file (Listing D-6).

Listing D-6. Simple C Source File

```
int x_input[256];

main()
{
int i;

for (i=0, i<256; i++)
     x_input[i] = 1;

}   // end main
```

Listing D-7. Simple C-based LDF Example for an ADSP-2191DSP

```
ARCHITECTURE(ADSP-2191)

SEARCH_DIR( $ADI_DSP\219x\lib )

// Example interrupt vector table
$INTTAB   = 219x_int_tab.doj;

// libsim provides fast, mostly host emulated I/O only supported
// by the simulator. The libio library provides I/O processing
// mostly done by the 219X target that is supported by the
// emulator and simulator. Libio is the default used,
// but if __USING_LIBSIM is defined libsim will be used.
//   from the driver command line, use:
//      "-flags-link -MD__USING_LIBSIM=1"
//   in the IDDE, add -MD__USING_LIBSIM=1 to
//   to the Additional options field of the Link page

#ifdef __USING_LIBSIM
```

```
$IOLIB     = libsim.dlb;
#else   // !__USING_LIBSIM
$IOLIB     = libio.dlb;
#endif   //  __USING_LIBSIM

// When an object that was compiled as C++ is included on the
// link line, the __cplusplus macro is defined to link with the
// C++ libraries and run-time mechanisms. Use the compiler driver
// (cc219x) to link C++ compiled objects to ensure that
// any static initialisations and template C++
// matters are resolved.

#ifdef __cplusplus
$CLIBS      = libc.dlb, libdsp.dlb, libcpp.dlb, libcpprt.dlb;
$START      = 219x_cpp_hdr.doj;
#else   // !__cplusplus
$CLIBS      = libc.dlb, libdsp.dlb;
$START      = 219x_hdr.doj;
#endif  //  __cplusplus

// Libraries from the command line are included
// in COMMAND_LINE_OBJECTS.

$OBJECTS   = $START, $INTTAB, $COMMAND_LINE_OBJECTS;
$LIBRARIES = $IOLIB, $CLIBS;

// This memory map is set up to facilite testing of the tool
// chain. Code and data area are as large as possible. Code
// is placed in page 0, starting with space reserved for the
// interrupt table. All data is placed in page 1. Note that
// the run-time header must initialize the data page registers
// to 1 to match this placement of program data. All pages are
// 64K words.

MEMORY
{
      // The memory section where the reset vector resides
mem_INT_RSTI { TYPE(PM RAM) START(0x000000) END(0x00001f)
WIDTH(24) }
```

```
      // The memory sections where the interrupt vector code and
      // an interrupt table used by library functions resides.
      // The library functions concerned include signal(),
      // interrupt(), raise(), and clear_interrupts().
mem_INT_PWRDWN   { TYPE(PM RAM) START(0x000020) END(0x00003f)
                   WIDTH(24) }
mem_INT_KERNEL   { TYPE(PM RAM) START(0x000040) END(0x00005f)
                   WIDTH(24) }
mem_INT_STKI     { TYPE(PM RAM) START(0x000060) END(0x00007f)
                   WIDTH(24) }
mem_INT_INT4     { TYPE(PM RAM) START(0x000080) END(0x00009f)
                   WIDTH(24) }
mem_INT_INT5     { TYPE(PM RAM) START(0x0000a0) END(0x0000bf)
                   WIDTH(24) }
mem_INT_INT6     { TYPE(PM RAM) START(0x0000c0) END(0x0000df)
                   WIDTH(24) }
mem_INT_INT7     { TYPE(PM RAM) START(0x0000e0) END(0x0000ff)
                   WIDTH(24) }
mem_INT_INT8     { TYPE(PM RAM) START(0x000100) END(0x00011f)
                   WIDTH(24) }
mem_INT_INT9     { TYPE(PM RAM) START(0x000120) END(0x00013f)
                   WIDTH(24) }
mem_INT_INT10    { TYPE(PM RAM) START(0x000140) END(0x00015f)
                   WIDTH(24) }
mem_INT_INT11    { TYPE(PM RAM) START(0x000160) END(0x00017f)
                   WIDTH(24) }
mem_INT_INT12    { TYPE(PM RAM) START(0x000180) END(0x00019f)
                   WIDTH(24) }
mem_INT_INT13    { TYPE(PM RAM) START(0x0001a0) END(0x0001bf)
                   WIDTH(24) }
mem_INT_INT14    { TYPE(PM RAM) START(0x0001c0) END(0x0001df)
                   WIDTH(24) }
mem_INT_INT15    { TYPE(PM RAM) START(0x0001e0) END(0x0001ff)
                   WIDTH(24) }
mem_itab         { TYPE(PM RAM) START(0x000200) END(0x000241)
                   WIDTH(24) }
```

```
        // The default program memory used by the compiler.
mem_code         { TYPE(PM RAM) START(0x000242) END(0x006fff)
                   WIDTH(24) }
        // The default PM data memory used by the compiler.
mem_data2        { TYPE(PM RAM) START(0x007000) END(0x007fff)
                   WIDTH(24) }

        // The default DM data memory used by the compiler.
#ifdef __cplusplus
   mem_ctor      { TYPE(DM RAM) START(0x008000) END(0x0080FF)
                   WIDTH(16) }
   mem_data1     { TYPE(DM RAM) START(0x008100) END(0x00f1ff)
                   WIDTH(16) }
#else
   mem_data1     { TYPE(DM RAM) START(0x008000) END(0x00f1ff)
                   WIDTH(16) }
#endif

        // Memory section used for dynamic allocation routines.
   mem_heap      { TYPE(DM RAM) START(0x00f200) END(0x00f9ff)
                   WIDTH(16) }

        // The memory section used for the software stack pointed
        // to by STACKPOINTER(I4) and FRAMEPOINTER(I5).
   mem_stack     { TYPE(DM RAM) START(0x00fa00) END(0x00ffff)
                   WIDTH(16) }
}

PROCESSOR p0
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS
   {

     sec_INT_RSTI {
       INPUT_SECTIONS ( $OBJECTS(IVreset) $LIBRARIES( IVreset ) )
     } > mem_INT_RSTI
```

```
sec_INT_PWRDWN {
  INPUT_SECTIONS ($OBJECTS(IVpwrdwn)$LIBRARIES(IVpwrdwn ) )
} > mem_INT_PWRDWN

sec_INT_STKI {
  INPUT_SECTIONS($OBJECTS(IVstackint)$LIBRARIES(IVstackint) )
} > mem_INT_STKI

sec_INT_KERNEL {
  INPUT_SECTIONS($OBJECTS(IVkernel)$LIBRARIES( IVkernel ) )
} > mem_INT_KERNEL

sec_INT_INT4 { INPUT_SECTIONS( $OBJECTS( IVint4 )
          $LIBRARIES( IVint4 ) )
               } > mem_INT_INT4
sec_INT_INT5 { INPUT_SECTIONS( $OBJECTS( IVint5 )
          $LIBRARIES( IVint5 ) )
               } > mem_INT_INT5
sec_INT_INT6 { INPUT_SECTIONS( $OBJECTS( IVint6 )
          $LIBRARIES( IVint6 ) )
               } > mem_INT_INT6
sec_INT_INT7 { INPUT_SECTIONS( $OBJECTS( IVint7 )
          $LIBRARIES( IVint7 ) )
               } > mem_INT_INT7
sec_INT_INT8 { INPUT_SECTIONS( $OBJECTS( IVint8 )
          $LIBRARIES( IVint8 ) )
               } > mem_INT_INT8
sec_INT_INT9 { INPUT_SECTIONS( $OBJECTS( IVint9 )
          $LIBRARIES( IVint9 ) )
               } > mem_INT_INT9
sec_INT_INT10 { INPUT_SECTIONS( $OBJECTS( IVint10 )
          $LIBRARIES( IVint10 ) )
               } > mem_INT_INT10
sec_INT_INT11 { INPUT_SECTIONS( $OBJECTS( IVint11 )
          $LIBRARIES( IVint11 ) )
               } > mem_INT_INT11
sec_INT_INT12 { INPUT_SECTIONS( $OBJECTS( IVint12 )
          $LIBRARIES( IVint12 ) )
               } > mem_INT_INT12
```

```
    sec_INT_INT13 { INPUT_SECTIONS( $OBJECTS( IVint13 )
              $LIBRARIES( IVint13 ) )
                   } > mem_INT_INT13
    sec_INT_INT14 { INPUT_SECTIONS( $OBJECTS( IVint14 )
              $LIBRARIES( IVint14 ) )
                   } > mem_INT_INT14
    sec_INT_INT15 { INPUT_SECTIONS( $OBJECTS( IVint15 )
              $LIBRARIES( IVint15 ) )
                   } > mem_INT_INT15

    sec_itab { INPUT_SECTIONS( $OBJECTS(lib_int_table)
              $LIBRARIES(lib_int_table))
             } > mem_itab

    sec_code { INPUT_SECTIONS( $OBJECTS(program)
              $LIBRARIES(program) )
             } > mem_code

    sec_data1 { INPUT_SECTIONS( $OBJECTS(data1)
              $LIBRARIES(data1) )
              } > mem_data1

    sec_data2 { INPUT_SECTIONS( $OBJECTS(data2)
              $LIBRARIES(data2) )
              } > mem_data2

  // provide linker variables describing the stack (grows down)
  //     ldf_stack_limit is the lowest address in the stack
  //     ldf_stack_base is the highest address in the stack
    sec_stack {
         ldf_stack_limit = .;
         ldf_stack_base = . + MEMORY_SIZEOF(mem_stack) - 1;
    } > mem_stack

    sec_heap {
         heap = .;
         heap_size = MEMORY_SIZEOF(mem_heap);
         heap_end = . + MEMORY_SIZEOF(mem_heap) - 1;
    } > mem_heap
```

```
#ifdef __cplusplus
     sec_ctor {
         __ctors = .;   /* points to the start of the section */
         INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor))
         INPUT_SECTIONS( $OBJECTS(ctor_end) $LIBRARIES(ctor_end))
     } > mem_ctor
#endif

     } // SECTIONS
}    // PROCESSOR p0

// end of file
```

Linking Overlay Memory for an ADSP-2191 System

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

Listing D-8 shows an example overlay memory .LDF. For more information, see the comments in the listing.

Listing D-8. Overlay-Memory System LDF Example

```
ARCHITECTURE(ADSP-2191)
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY{
   mem_seg_rth    { TYPE(PM RAM) START(0x000000) END(0x0001ff)
      WIDTH(24) }   // interrupt vector table locations
   mem_seg_code   { TYPE(PM RAM) START(0x000200) END(0x0002ff)
      WIDTH(24) }   // static memory segment for non-overlay code
   mem_seg_plit   { TYPE(PM RAM) START(0x000300) END(0x00037f)
      WIDTH(24) }   // static memory segment for PLIT code
   mem_seg_pm_data    { TYPE(PM RAM) START(0x000380) END(0x0003ff)
      WIDTH(24) }   // static memory segment for PM data segment
   mem_seg_ovl    { TYPE(PM RAM) START(0x000400) END(0x007fff)
      WIDTH(24) }   // run address range for overlay functions

   mem_seg_dm_data    { TYPE(DM RAM) START(0x008000) END(0x00ffff)
      WIDTH(16) }   // static memory segment for DM data segment

   mem_ovl1_liv_space    { TYPE(PM RAM) START(0x200000)
      END(0x2000ff) WIDTH(24) }    // live address range for
                                    overlay function #1
```

```
   mem_ovl2_liv_space   { TYPE(PM RAM) START(0x200100)
      END(0x2001ff) WIDTH(24) }   // live address range for
                                 overlay function #2
   mem_ovl3_liv_space   { TYPE(PM RAM) START(0x200200)
      END(0x2002ff) WIDTH(24) }   // live address range for
                                 overlay function #3
}

PROCESSOR p0{
   LINK_AGAINST($COMMAND_LINE_LINK_AGAINST)
   OUTPUT($COMMAND_LINE_OUTPUT_FILE)

   PLIT{
      dm(save_ax0) = ax0;   // save ax0 register before calling
                           overlay manager (which uses ax0)
      dm(save_ay0) = ay0;   // save ay0 register before calling
                           overlay manager (which uses ay0)
         ax0 = PLIT_SYMBOL_OVERLAYID;
                           // assign ax0 with the overlay ID#
         ay0 = PLIT_SYMBOL_ADDRESS;
                           // assign ay0 with the run address for
                           the desired overlay function

      ljump _OverlayManager;
             // jump to the overlay manager function
   }

   SECTIONS{
      dxe_seg_rth{
         INPUT_SECTIONS("2191_ASM_Interrupt_Table.doj"(interrupts))
      } >mem_seg_rth

      dxe_seg_code{
         INPUT_SECTIONS("Main.doj"(seg_code) "DMA Overlay
               Manager.doj"(seg_code))
      } >mem_seg_code

      .plit{}> mem_seg_plit   // define the live address for the
            PLIT table in its own special memory segment
```

```
        dxe_seg_pm_data{
           INPUT_SECTIONS("PM Data.doj"(seg_pmdata))
        } >mem_seg_pm_data

        dxe_seg_ovl{
           OVERLAY_INPUT{
              ALGORITHM(ALL_FIT)
              OVERLAY_OUTPUT("Function_1_Add.ovl")
              INPUT_SECTIONS("Function_1_Add.doj"(seg_code))
           } >mem_ovl1_liv_space
           // Overlay to live in section ovl1_liv_space

           OVERLAY_INPUT{
              ALGORITHM(ALL_FIT)
              OVERLAY_OUTPUT("Function_2_Mult.ovl")
              INPUT_SECTIONS("Function_2_Mult.doj"(seg_code))
           } >mem_ovl2_liv_space
           // Overlay to live in section ovl2_liv_space

           OVERLAY_INPUT{
              ALGORITHM(ALL_FIT)
              OVERLAY_OUTPUT("Function_3_Sub.ovl")
              INPUT_SECTIONS("Function_3_Sub.doj"(seg_code))
           } >mem_ovl3_liv_space
              // Overlay to live in section ovl3_liv_space
        } >mem_seg_ovl   // Overlays run in this memory segment

        dxe_seg_dm_data{
           INPUT_SECTIONS("DM_Data.doj"(seg_data)
                 "DMA Overlay Manager.doj"(seg_data))
        } >mem_seg_dm_data

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

Linking an ADSP-219x MP System With Shared Memory

When you link executable files for multiprocessor (MP) memory or a shared memory system, the LDF describes the shared memory and each processor's separate memory (with offsets), and places code for each processor. Listing D-9 shows a multiprocessor system and shared memory LDF.

Listing D-9. LDF for a Multiprocessor System with Shared Memory

```
ARCHITECTURE(ADSP-219x)
SEARCH_DIR( $ADI_DSP\219x\lib )

// Multiprocessor memory space is allocated with the MPMEMORY{}
// command. The values represent an “addend” that the linker
// uses when it resolves undefined symbols in one DXE to symbols
// defined in another DXE. The addend is added to each defined
// symbol’s value.
// For example, PROCESSOR project PSH0 contains the undefined
// symbol “buffer”, PROCESSOR project PSH1 defines “buffer”
// at address 0x22000. The linker will “fix up” the reference
// to “buffer” in PSH0’s code to address:
//   0x22000 + MPMEMORY(PSH1) = 0x22000 + 0x280000 = 0x2a2000

MPMEMORY
{
   PSH0 { START (0x200000) }
   PSH1 { START (0x280000) }
}
MEMORY {        // Used for all processors. Alternatively, a
   seg_reset    // PROCESSOR could describe its own MEMORY
      { TYPE(PM RAM) START(0x00000) END(0x00004) WIDTH(24) }
   seg_itab
      { TYPE(PM RAM) START(0x000004) END(0x0000ff) WIDTH(24) }
   seg_code
      { TYPE(PM RAM) START(0x000100) END(0x00ffff) WIDTH(24) }
```

```
    seg_dmda
       { TYPE(DM RAM) START(0x010000) END(0x017fff) WIDTH(16) }
    seg_data2
       { TYPE(PM RAM) START(0x018000) END(0x01ebff) WIDTH(24) }
    seg_heap
       { TYPE(DM RAM) START(0x01ec00) END(0x01efff) WIDTH(16) }
    seg_stack
       { TYPE(DM RAM) START(0x01f000) END(0x01ffff) WIDTH(16) }
    }

 $LIBRARIES = lib219x.dlb, libc.dlb;

 // This LDF specifies three link projects. The first is a
 // shared memory project against which the PROCESSOR projects
 // will be linked. The file containing the shared data
 // buffers is defined in shared.c.

 SHARED_MEMORY {
    $SHARED_OBJECTS = shared.doj;
    // The output name of this shared object is subsequently
    // used in the PROCESSOR project's LINK_AGAINST command
    OUTPUT(shared.sm)
    // shared.c has only data declarations. No need to
    // specify an output section other than "seg_dmda".
    SECTIONS {
       seg_dmda {INPUT_SECTIONS( $SHARED_OBJECTS(seg_dmda))
       } > seg_dmda
    }
 }

 // The second link project is a DXE project. It will be linked
 // against the SHARED link project defined above.

 PROCESSOR PSH0 {
    $PSH0_OBJECTS =  psh0.doj, 219x_hdr.doj;
    LINK_AGAINST(shared.sm)
    OUTPUT( psh0.dxe )

    SECTIONS {
       dxe_pmco
```

```
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_pmco)
            $LIBRARIES(seg_pmco)) }                  > seg_pmco
         dxe_pmda
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_pmda)
            $LIBRARIES(seg_pmda)) }                  > seg_pmda
         dxe_dmda
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_dmda)
            $LIBRARIES(seg_dmda)) }                  > seg_dmda
         dxe_init
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_init)
            $LIBRARIES(seg_init)) }                  > seg_init
         dxe_rth
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_rth)
            $LIBRARIES(seg_rth))  }                  > seg_rth
         stackseg {   // allocate a stack for the application
            ldf_stack_space = .;
            ldf_stack_length = 0x2000;  }            > seg_stak
         heap {       // allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + 0x2000;
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
         } > seg_heap
      }
   }

   // The last project defined in this LDF is another DXE
   // project. This PROCESSOR project will be linked against both
   // the SHARED and the PSH0 DXE projects defined above.
   PROCESSOR PSH1 {
      $PSH1_OBJECTS = psh1.doj, 219x_hdr.doj;
      LINK_AGAINST(shared.sm, psh0.dxe)
      OUTPUT(psh1.dxe)
      SECTIONS {
         dxe_pmco
            {INPUT_SECTIONS( $PSH1_OBJECTS(seg_pmco)
            $LIBRARIES(seg_pmco)) }                  > seg_pmco
         dxe_pmda
            {INPUT_SECTIONS( $PSH1_OBJECTS(seg_pmda)
            $LIBRARIES(seg_pmda)) }                  > seg_pmda
         dxe_dmda
```

```
        {INPUT_SECTIONS( $PSH1_OBJECTS(seg_dmda)
        $LIBRARIES(seg_dmda)) }                  > seg_dmda
     dxe_init
        {INPUT_SECTIONS( $PSH1_OBJECTS(seg_init)
        $LIBRARIES(seg_init)) }                  > seg_init
     dxe_rth
        {INPUT_SECTIONS( $PSH1_OBJECTS(seg_rth)
        $LIBRARIES(seg_rth)) }                  > seg_rth
     stringstab
        {INPUT_SECTIONS( $PSH1_OBJECTS(.stringstab)
        $LIBRARIES(.stringstab)) }
     SDB
        {INPUT_SECTIONS( $PSH1_OBJECTS(.SDB)
        $LIBRARIES(.SDB)) }
     lnno.seg_pmco
        {INPUT_SECTIONS($PSH1_OBJECTS(.lnno.seg_pmco)
        $LIBRARIES(.lnno.seg_pmco)) }
     stackseg {   // stack for the application
        ldf_stack_space = .;
        ldf_stack_length = 0x2000; }            > seg_stak

     heap {        // heap for the application
        ldf_heap_space = .;
        ldf_heap_end = ldf_heap_space + 0x2000;
        ldf_heap_length = ldf_heap_end - ldf_heap_space;
        }                                       > seg_heap
     }
}
```

Overlays Used With ADSP-218x DSPs

This example details the handling of overlay pages for ADSP-218x DSPs. The following file (`main.asm`) is part of an example program (ADSP-2189M ASM HardWare Overlay) shipped with VisualDSP++.

```
.section/pm   program;
.global       START;

.extern      ADD, SUB, MULT;   // external PM modules which
               // reside in external PM overlay regions
.extern      ONE, TWO;    // external DM variables which reside
               // in external DM overlay regions
.extern      FOUR, FIVE;  // external DM variables which
               // reside in external DM overlay regions
,extern      THREE, RESULT;    // external DM variables which
               // reside in non-overlay/fixed-memory DM region

// Beginning of main program
START:
   mstat = 0x10;       // configure core for integer mode
   dmovlay = 1;        // jump to external DM overlay region #1
     ax0 = dm(ONE);  // read value of memory mapped variable
                       // that lives in external DM
                       // overlay region #1 into ax0
   dmovlay = 2;        // jump to external DM overlay region #2
   ay0 = dm(TWO);      // read value of memory mapped variable
                       // that lives in external DM
                       // overlay region #2 into ay0
   pmovlay = 4;        // jump to PM overlay region #4
   call ADD;           // Call the ADD function which lives
                       // in external PM overlay  #4
     ax0 = dm(RESULT);// read value of memory mapped variable
                       // that lives in internal non-overlay
                       // DM memory region into ax0
     ay0 = dm(THREE); // read value of memory mapped variable
                       // that lives in internal non-overlay
                       // DM memory region into ay0
   pmovlay = 0;        // jump to internal PM overlay region #0
```

```
    call SUB;           // Call the SUB function that lives
                        // in internal PM overlay #0
       mr=0;            // Clear out the MR register,
                        // we'll need it later
    dmovlay = 4;        // jump to internal dm overlay region #4
       mx0 = DM(FOUR);  // read value of memory mapped variable
                        // that lives in internal DM
                        // overlay region #4 into mx0
    dmovlay = 5;        // jump to internal DM overlay region #5
       my0 = dm(FIVE);  // read value of memory mapped
                        // variable that lives in internal DM
                        // overlay region #5 into my0
    pmovlay = 5;        // jump to int. PM overlay memory region #5
    call MULT;          // Call the MULT function that resides
                        // in internal PM overlay #5
 DONE:
    idle;               // wait here until an interrupt occurs
    jump DONE;          // jump back to "idle" instruction after
                        // returning from interrupt subroutine
```

I INDEX

Symbols

$ADI_DSP LDF macro 3-22
$COMMAND_LINE_LINK_AGAINST LDF macro 3-21
$COMMAND_LINE_OBJECTS LDF macro 3-21
$COMMAND_LINE_OUTPUT_DIRECTORY LDF macro 3-21
$COMMAND_LINE_OUTPUT_FILE LDF macro 3-9, 3-21
$macroname LDF macro 3-22
$OBJECTS LDF macro 3-7
+?ags-pp linker command-line switch 2-42
.ASM files
  assembler A-3
.DAT files
  initialization data A-3
.DLB files 2-32, A-6
  description of A-6
  symbol name encryption 6-15
.DOJ files 2-32
  about A-5
.DXE files 1-6, 2-32, A-6
  data extraction B-1
  linker A-6
.LDF files 2-32, A-4
  commands in 2-3, 3-23, 5-27
  comments in 3-11
  creating in Expert Linker 4-4
  memory segments 2-4
  output sections 2-4
.LDR files
  ASCII-format A-8
  hex-format A-6
  splitter output A-8
.MEMINIT section name 3-43
.OVL files 1-6, 2-32, 3-47, A-6, B-4
  dumping B-4
  extracting content from B-4
  linker A-6
.SECTION directive 1-4
.SM file 5-38
.SM files 1-6, 2-32, A-6
  linker A-6
.TXT files A-5
  linker A-5
.XML file 2-39, 3-29, A-6
@filename linker command-line switch 2-38
__SILICON_REVISION__ macro 2-47
_ov_end breakpoint 5-7, 5-8
_ov_endaddress_# 5-9, 5-22

_ov_runtimestartaddress_# 5-9, 5-22
_ov_size_# 5-9, 5-22
_ov_start breakpoint 5-8
_ov_startaddress_ 5-22
_ov_startaddress_# 5-9
_ov_word_size_live_# 5-9, 5-22
_ov_word_size_run_# 5-9, 5-22

A

-a archiver command-line switch 6-12
absolute data placements 2-43
ABSOLUTE() LDF operator 3-16
adding
  input sections 4-12
  LDF macros 4-12
  library files 4-12
  object files 4-12
ADDR() LDF operator 3-17
address
  setting for command-line arguments 2-29
ADSP-218x DSPs
  overlays D-23
ALGORITHM() LDF command 3-47
ALIGN() LDF command 3-24
alignment
  specifying properties 4-65
ALL_FIT LDF identifier 3-47, 4-68
-anv archiver command-line switch 6-12
ARCHITECTURE() LDF command 3-24
archive
  files See library files A-6
  library file 6-1
  members A-6
  writing library files in 6-3
archive routines
  creating entry points 6-4
archiver
  about 6-1
  adding text to version information 6-10
  adding version information 6-6
  checking version number 6-9
  command constraints 6-14
  command-line switches 6-12
  command-line syntax 6-11
  deleting version information 6-9
  file searches 6-6
  handling arbitrary files 6-2
  printing version information 6-8
  removing version information 6-9
  running 6-11, 6-12
  symbol name encryption 6-15
  tagging with version 6-6
  use in code disassembly B-3
  using wildcard character 6-13
ARGV section 2-20, 2-29
assembler
  initialization data files (.DAT) A-3
  object files (.DOJ) A-5
  source files (.ASM) 1-3, A-3

B

B0 bytes 3-34
base address
  setting for command-line arguments 2-29
BEST_FIT LDF identifier 3-48
Blackfin processors
  memory ranges 2-20
bootup
  sections 2-20
branch expansion instruction 2-42, 2-43, 2-44, 2-46
branch instructions 5-36
breakpoints
  on overlays 5-7
build files
  description of A-5
built-in LDF macros 3-21
byte_order_list byte 3-34

C

-c archiver command-line switch 6-12
C/C++
  source files A-2
cache
  memory 2-12
calls
  inter-overlay 5-24
  inter-processor 5-24
color selection
  in Expert Linker 4-15
command-line arguments base address
  setting 2-29
commands
  LDF 3-23, 5-27
  linker 2-30
comma-separated option 2-41, 2-42
comments
  .LDF file 3-11
compiler
  source files (.C .CC) 1-3
constdata input section 2-20
converting
  library members to source code B-3
  out-of-range short calls and jumps 2-43, 2-44
Create LDF wizard 4-4
ctor input section 2-20
custom processors 2-45

D

-d archiver command-line switch 6-12
-Darchitecture (target architecture) linker command-line switch 2-38
data placement 2-42
data1 input section 2-20
debugger
  files A-9
declaring
  macros 3-22
DEFINED() LDF operator 3-18
directories
  supported by linker 2-33

disassembly
library member B-3
using archiver B-3
using dumper B-3
-dnv archiver command-line switch 6-12
DSPs
development software 1-2
dumper
use in code disassembly B-3
DWARF
references A-10

E

-e (eliminate unused symbols) linker command-line switch 2-41
-e archiver command-line switch 6-12
ELF file contents B-1
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 6-1
command-line reference 6-11
elfdump.exe
about B-1
command-line switches B-1
used by Expert Linker 4-35
ELIMINATE() LDF command 3-25
ELIMINATE_SECTIONS() LDF command 3-26
elimination
enabling 3-25, 3-27
specifying properties 4-55
empty bytes 3-34
emulation
passing arguments in Blackfin processors 2-29
encryption
symbol names in libraries 6-15
end address
memory segment 3-32
END() LDF identifier 3-32
errors
linker 2-10
-es (eliminate listed sections) linker command-line switch 2-41
-ev (eliminate unused symbols, verbose) linker command-line switch 2-41
executable files 1-6, A-6
expanding
items in memory map 4-21
Expert Linker
about 4-1
adding input sections, object files and LDF macros 4-13
adding output section to memory segment 4-20
adding shared memory 4-20
adding shared memory segments 4-45
color selection 4-15

- deleting objects 4-13
- displaying global properties 4-13
- expanding items 4-21
- expanding LDF macro 4-13
- Input Sections pane 4-12
- invalid memory segments 4-19
- launching 4-3
- Legend dialog box 4-14
- mapping sections in 4-14
- memory map graphical view 4-23
- Memory Map pane 4-18
- multiprocessing tasks 4-45
- object properties 4-50
- overlays 4-33
- overview 2-9, 4-1
- profiling object sections 4-40
- removing LDF macro 4-13
- resize cursor 4-26
- specifying memory segments 4-20

extracting
- data from ELF executable files B-1

F

files
- .ASM A-3
- .DAT A-3
- .DLB A-6
- .DOJ A-5
- .DXE A-6
- .LDR (ASCII-format) A-8
- .LDR (hex format) A-6
- .OVL A-6
- .SM A-6
- .TXT A-5
- .XML A-6
- 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
- input A-2
- library A-6
- linker command-line 2-32
- linker command-line (.TXT) A-5
- object 2-34
- output 1-6

FILL() LDF command 3-26, 3-46

FIRST_FIT LDF identifier 3-47

-flags-meminit linker command-line switch 2-41

fragmented memory
- filling in 2-42

Full Memory mode 2-16

G

gaps
- inserting into memory segment 4-31

H

-h (help) assembler switch 2-42

heap
- graphic representation 4-69
- managing in memory 4-69
- program section 2-20

hex-format files
.LDR A-6
Host Memory mode 2-16

I
-i (include search directory) linker command-line switch 2-42
-i filename archiver command-line switch 6-12
icons
Expert Linker 4-14
unmapped icon 4-14
IDMA
port 2-16
INCLUDE() LDF command 3-26
individual data placement option 2-43
input sections
adding 4-12
directives 1-4
names 2-18
source code 1-3
Input Sections pane 4-12
menu selections 4-12
INPUT_SECTION_ALIGN() LDF command 3-26
INPUT_SECTIONS() LDF identifier 3-9, 3-45
inserting
gaps into memory segment 4-31
inter-overlay calls 5-24
inter-processor calls 5-24
-ip (individual placement) linker command-line switch 2-42

J
-jcs2l (convert out-of-range short calls) linker command-line switch 2-43
-jcs2l+ (convert out-of-range short calls) linker command-line switch 2-44
jumps
converting 2-43

K
-keep (keep unused symbols) linker command-line switch 2-44
KEEP() LDF command 3-27

L
-L path (libraries and objects) linker command-line switch 2-39
L1 memory 2-12
L2 memory 2-12
LDF commands
about 2-3, 3-23, 5-27
ALLIGN() 3-24
ARCHITECTURE() 3-24
ELIMINATE() 3-25
ELIMINATE_SECTIONS() 3-26
FILL() 3-46
INCLUDE() 3-26
INPUT_SECTION_ALIGN() 3-26
KEEP() 3-27
LINK_AGAINST() 3-28
MAP() 3-29

MEMORY{} 3-29, 5-29
MPMEMORY{} 3-32, 5-28
OVERLAY_GROUP{} 3-33, 5-29
OVERLAY_INPUT{} 3-46
PACKING() 3-33
PAGE_INPUT() 3-37
PAGE_OUTPUT() 3-38
PLIT{} 3-46, 5-34
PROCESSOR{} 3-39
RESOLVE() 3-40
SEARCH_DIR() 3-41
SECTIONS{} 3-42
SHARED_MEMORY{} 3-48, 5-38

LDF file
- commands 3-23
- default 2-11
- keywords 3-14
- miscellaneous keywords 3-15
- operators 3-16
- purpose 2-5
- structure 3-11

LDF macros
- about 3-20
- adding 4-12
- built-in 3-21
- command-line input 3-22
- expanding 4-13
- removing 4-13

LDF operators
- about 3-19
- ABSOLUTE() 3-16
- ADDR() 3-17
- DEFINED() 3-18
- MEMORY_SIZEOF() 3-18
- SIZEOF() 3-19

legends
- Expert Linker 4-13

LENGTH() LDF identifier 3-32

librarian
- VisualDSP++ 6-1

library
- symbol name encryption 6-15

library files
- about A-6
- adding 4-12
- creating 6-3
- searching 6-2

library members 6-1
- converting to source code B-3

library routines
- using 6-5

Link tab
- setting linker options 2-7

link target 2-11

LINK_AGAINST() LDF command 3-28

linker 1-2
- about 2-1
- command-line files (.TXT) A-5
- command-line switches 2-34
- command-line syntax 2-30
- commands 2-30
- describing the target 2-11
- error messages 2-10
- executable files A-6
- file name conventions 2-33

linking object files 2-34
memory map files (.XML) A-6
options 2-3
outputs 1-6
overlay constants generated by 5-9
warning messages 2-10
linker command-line switches
-Darchitecture 2-38
-Dprocessor 2-38
-e 2-41, 2-44
-es secName 2-41
-ev 2-41
-flags-meminit 2-41
-flags-pp 2-42
-h (help) 2-42
-i (include search directory) 2-42
-ip (individual placement) 2-42
-jcs2l 2-43
-jcs2l+ 2-44
-keep symbolName 2-44
-L path 2-39
-M 2-39
-Map filename 2-39
-MDmacro 2-39
-meminit 2-44
-MM 2-39
-o filename 2-44
-od directory 2-45
-Ovcse (VCSE optimization) 2-40
-pp 2-45
-S 2-40
-s (strips all symbols) 2-46
-save-temps 2-46
-si-revision version (silicon revision) 2-46
-sp 2-48
-sp (skip preprocessing) 2-48
-t (trace) 2-48
-T filename 2-40, 2-49
-v (verbose) 2-48
-version (display version) 2-48
-warnonce 2-48
-Wwarn num (override error message) 2-40
-xref filename 2-49
Linker Description File
overview 2-5, 3-1
linker.exe 1-2
linking
about 2-2
controlling 2-3
file with large uninitialized variables C-4, D-5
file with large zero-initialized variables C-4, D-5
multiprocessor system D-19
overlay memory system C-17, D-16
process rules 2-4
single-processor system C-2, D-3
loader
creating bootloadable image 1-8
hex-format files A-6
location counter 3-19
definition of 3-19

M
-M (dependency check and output) linker command-line switch 2-39
-M archiver command-line switch 6-12
macros
  LDF 3-20
  preprocessor 3-20
  user-declared 3-22
-Map (filename) linker command-line switch 2-39
map file 2-34, 3-29
MAP() LDF command 3-29
mapping
  input sections to output sections 4-14
-MDmacro (macro value) linker command-line switch 2-40
MEM_ARGV memory section 2-20
MEM_ARGV section 2-29
MEM_BOOTUP memory section 2-20
MEM_HEAP memory section 2-20
MEM_PROGRAM section 2-20
MEM_STACK memory section 2-20
MEM_SYSSTACK memory section 2-20
-meminit linker command-line switch 2-44
memory
  allocation 2-12, 2-18
  architecture 2-12
  architecture representation 2-11
  Blackfin processor range 2-20
  initializer 2-41, 2-44, 3-43
  managing heap/stack 4-69
  map files A-6
  overlays 5-4, 5-5
  partitions 4-18
  segment declaration 2-12
  segment length 3-32
  segments 4-18
  types 2-12, 3-31
memory map
  generating 2-39
  graphical view 4-23
  highlighted objects in 4-26
  post-link view 4-27
  pre-link view 4-27
  specifying 2-18
  tree view 4-22
  viewing 4-19
Memory Map pane 4-19, 4-20
  overlays 4-33
  zooming in/out 4-28
memory segments
  about 1-3
  changing size of 4-25
  gap 4-31
  invalid 4-19
  MEMORY{} command 4-18
  rules 2-4
  size 4-22
  specifying properties 4-61
  start address 4-22

MEMORY_SIZEOF() LDF operator 3-18
MEMORY{} LDF command 2-11, 3-8, 3-29, 5-29
  segment_declaration 3-30
-MM (dependency check, output and build) linker command-line switch 2-39
-MM archiver command-line switch 6-12
modes
  Full Memory 2-16
  Host Memory 2-16
MPMEMORY{} LDF command 3-32, 5-28
multiple overlays 4-33
multiprocessor systems
  linking D-19

N

NO_INIT qualifier 3-44, C-4, D-5
null bytes 3-34, 3-36

O

-o filename linker command-line switch 2-44
object files 1-3
  adding 4-12
  linking into executable 2-2
object properties
  managing with Expert Linker 4-50
objects
  deleting 4-13
  sorting 4-17
-od (output directory) linker command-line switch 2-45
operators
  LDF 3-16
output directory
  specifying 2-45
output sections
  about 2-3
  dumping 2-19
  rules 2-4
  specifying properties 4-62
OUTPUT() LDF command 3-9, 4-18
ov_id_loaded buffer 5-16
-Ovcse (VCSE optimization) linker command-line switch 2-40
overlay
  ALL_FIT algorithm 4-68
  identifier 5-10
overlay file
  producing 3-47
overlay ID 5-16
overlay library files B-4
overlay manager
  about 5-4, 5-6, 5-7
  assembly code 5-36
  constants 5-15
  major functions 5-7
  performance summary 5-17
  placing constants 5-16
  PLIT table 5-12
  storing overlay ID 5-16
overlay memory

linking for C-17, D-16
OVERLAY_GROUP{} LDF command 3-33, 5-29
OVERLAY_ID LDF identifier 3-47
OVERLAY_INPUT{} LDF command 3-46
OVERLAY_OUTPUT() LDF command 3-47
overlays
address 5-9, 5-15
ADSP-218x DSPs D-23
constants 5-9, 5-14
debugging 5-7
dumping library files B-4
grouped 5-30
grouping 5-30
in Memory Map pane 4-33
live space 4-33
loading and executing 5-18
loading instructions with PLIT 5-37
managing properties 4-67
memory 5-4, 5-5
multiple 4-33
numbering 5-22
reducing overhead 5-17
run space 4-33
special symbols 5-22
ungrouped 5-30
word size 5-9, 5-15

P

-p archiver command-line switch 6-13
packing
efficient 3-35
specifying properties 4-64
PACKING() LDF command 3-33
PAGE_INPUT() LDF command 3-37
PAGE_OUTPUT() LDF command 3-38
paged memory 3-37
pinning
to output section 4-21
PLIT
about 5-10
allocating space for 5-36
constants 5-35
executing user-defined code 5-10
overlay management 5-7
resolving inter-overlay calls 5-24
specifying properties 4-54
summary 5-37
syntax 5-34
PLIT_SYMBOL constants 5-38
PLIT_SYMBOL_ADDRESS 5-35
PLIT_SYMBOL_OVERLAYID 5-35
PLIT{} LDF command 3-46, 5-34
about 5-34
in SECTIONS{} 3-46
instruction qualifier 5-35
PLIT_SYMBOL_ADDRESS 5-35
PLIT_SYMBOL_OVERLAYID 5-35

-pp (end after preprocessing) linker command-line switch 2-45
pp. exe preprocessor 1-7
preprocessor 1-7
  compiler 1-7
  running from linker 2-45
-proc (target processor) assembler switch 2-45
procedure linkage table (PLIT) 3-38, 5-34
  about 5-10
  using 5-22
processor
  selection 2-38
  specifying properties 4-53
PROCESSOR{} LDF command 3-39
program
  counter 5-14
  sections 2-20
project builds
  linker 2-6
Project Options dialog box 2-7
PROM 3-31
-pv archiver command-line switch 6-13
-pva archiver command-line switch 6-13

R

-r archiver command-line switch 6-13
RAM 3-31
references
  file formats A-10
removing
  LDF macro 4-13
reserving space for 5-10
resize cursor 4-26
RESOLVE() LDF command 3-29, 3-40
RESOLVE_LOCALLY() LDF command 3-48
ROM 3-31
run-time initialization qualifiers 3-43
run-time initialization. 3-43
RUNTIME_INIT qualifier 3-44

S

-s (strip all symbols) linker command-line switch 2-46
-S (strip debug symbols) linker command-line switch 2-40
-s archiver command-line switch 6-13
-save-temps linker command-line switch 2-46
SEARCH_DIR() LDF command 3-41
section_name qualifier 3-43
SECTIONS{} LDF command 2-26, 3-9, 3-42
segment declaration 3-30
setting
  command-line arguments base address 2-29

SHARED_MEMORY{} LDF command 3-48, 5-38
short calls
converting 2-43
SHT_NOBITS
section qualifier C-4, D-5
SHT_NOBITS keyword 3-44, C-4, D-5
silicon revision setting 2-46
simulation
passing arguments in Blackfin processors 2-29
-si-revision (silicon revision) linker command-line switch 2-46
SIZE() LDF command 3-48
SIZEOF() LDF operator 3-19
software
development 1-2
software development phases 1-2
sorting
objects 4-17
source code
in input sections 1-3
source files 1-3
assembly instructions A-3
C/C++ A-2
fixed-point data A-3
-sp (skip preprocessing) linker command-line switch 2-48
special section name
.MEMINIT 3-43
.PLIT 3-43
splitter 1-8
ASCII-format files (.LDR) A-8
SPORT data files A-9
stack
graphic representation 4-69
managing in memory 4-69
sections 2-20
stacks
managing in memory 4-69
start address
memory segment 3-31
symbol
declaration 3-7
manager 5-7
symbols
adding 4-58
deleting 4-60
encryption of names 6-15
managing properties 4-57
removing 2-46
viewing 4-39
sysstack
sections 2-20

T

-t (trace) linker command-line switch 2-48
-t archiver command-line switch 6-13
-T file (executable program placement) linker command-line switch 2-40
target processor
specifying 2-38
-tnv archiver command-line switch 6-13
tree view
memory map 4-22

-twc ver archiver command-line switch 6-13
-tx filename archiver command-line switch 6-13
TYPE() command 3-31

U

uninitialized variables C-4, D-5
unmapped object icon 4-14
user-declared macros 3-22
utilities
  archiver (elfar.exe) 6-1
  file dumper (elfdump.exe) B-1

V

-v (verbose) archiver command-line switch 6-13
-v (verbose) linker command-line switch 2-48
VCSE method calls 2-40
-version (display version) linker command-line switch 2-48
-version archiver command-line switch 6-13
version information
  built in with archiver 6-6
viewing
  .OVL file content B-4
  archive files B-4
  icons and colors 4-14
  input sections 4-12
  memory map 4-19
VisualDSP++
  archiver 6-1
  creating library files 6-3
  Expert Linker 4-2
  librarian 6-1

W

-w (remove warning) archiver command-line switch 6-13
warnings
  linker 2-10
-warnonce (single symbol warning) linker command-line switch 2-48
wildcard character
  specifying archive files 6-13
wizards
  Create LDF 4-4
-Wnnnn archiver command-line switch 6-13
word width (number of bits) 3-32
-Wwarn num (override error message) linker command-line switch 2-40

X

-xref (external reference file) linker command-line switch 2-49

Z

ZERO_INIT qualifier 3-44