

VISUALDSP++™ 3.0

Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs

Revision 4.0, July 2002

Part Number
82-000349-02

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



Copyright Information

© 2002 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, SHARC, the SHARC logo, TigerSHARC, and the TigerSHARC logo are registered trademarks of Analog Devices, Inc.

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

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

CONTENTS

PREFACE

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

CONTENTS

Contacting DSP Publications	xxiii
Notation Conventions	xxiii

LINKER AND UTILITIES

Software Development Flow	1-2
Compiling and Assembling	1-3
Inputs — C/C++ and Assembly Sources	1-4
Input Section Directives in Assembly Code	1-4
Input Section Directives in C/C++ Source Files	1-6
Linking	1-8
Loading/Splitting	1-9
Linking Overview	1-13
Linker Operation	1-14
Directing Linker Operation	1-15
Linking Process Rules	1-16
Linker Description File (.LDF)	1-17
Linking Environment	1-19
Project Builds	1-19
Expert Linker	1-22
Linker Warning and Error Messages	1-23
Link Target Description	1-24
Representing Memory Architecture	1-24
Specifying the Memory Map	1-25
Placing Code on the Target	1-27
Linker Command-Line Reference	1-28

Command Syntax	1-28
Command Line Object Files	1-29
Command-Line File Names	1-30
Objects	1-32
Linker Command-Line Switches	1-33
Switch Format	1-33
Switch Summary	1-34
@ file	1-36
-Darchitecture	1-36
-L path	1-36
-M	1-36
-MM	1-37
-Map file	1-37
-MDmacro[=def]	1-37
-S	1-37
-T file	1-37
-e	1-38
-es secName	1-38
-ev	1-38
-h -help	1-38
-i path	1-38
-ip	1-38
-jcs2l	1-39
-keep symName	1-39

CONTENTS

-o filename	1-39
-od filename	1-40
-pp	1-40
-proc ProcessorID	1-40
-s	1-40
-sp	1-40
-t	1-40
-v	1-41
-version	1-41
-warnonce	1-41
-xref filename	1-41
Memory Management Using Overlays	1-42
Memory Overlays	1-43
Overlay Managers	1-45
Memory Overlay Support	1-46
Example - Managing Two Overlays	1-50
Reducing Overlay Manager Overhead	1-59
Using PLIT{} and Overlay Manager	1-64
 LINKER DESCRIPTION FILE	
LDF Guide	2-2
.LDF File Overview	2-3
Example - Basic .LDF File	2-4
Notes on Basic LDF Example	2-5
LDF Structure	2-9

Command Scoping	2-10
LDF Expressions	2-11
LDF Keywords, Commands, and Operators	2-12
Miscellaneous LDF Keywords	2-13
LDF Operators	2-13
ABSOLUTE() Operator	2-14
ADDR() Operator	2-15
DEFINED() Operator	2-15
MEMORY_SIZEOF() Operator	2-16
SIZEOF() Operator	2-17
Location Counter (.)	2-17
LDF Macros	2-18
Built-In LDF Macros	2-19
User-Defined Macros	2-20
LDF Macros and Command-Line Interaction	2-21
LDF Commands	2-21
ALIGN()	2-22
ARCHITECTURE()	2-23
ELIMINATE()	2-23
ELIMINATE_SECTIONS()	2-24
INCLUDE()	2-24
INPUT_SECTION_ALIGN()	2-24
KEEP()	2-26
LINK_AGAINST()	2-26

CONTENTS

MAP()	2-27
MEMORY{}	2-27
MPMEMORY{}	2-30
OVERLAY_GROUP{}	2-32
Ungrouped Overlay Execution	2-33
Grouped Overlay Execution	2-36
PACKING()	2-37
Efficient Packing	2-39
Inefficient Packing: Null Bytes	2-39
Overlay Packing Formats	2-40
Trivial Packing: No Reordering	2-40
PAGE_INPUT()	2-40
PAGE_OUTPUT()	2-41
PLIT{}	2-42
PLIT Syntax	2-43
Allocating Space for PLITs	2-44
PLIT Examples	2-45
What PLIT Does – Summary	2-46
PROCESSOR{}	2-48
RESOLVE()	2-50
SEARCH_DIR()	2-50
SECTIONS{}	2-51
SHARED_MEMORY{}	2-56
LDF Programming Examples	2-59

Example — Linking for a Single-Processor ADSP-219x System	2-61
Example — Linking Large Uninitialized Variables	2-63
Example — Linking an Assembly Source File	2-65
Example — Linking a Simple C-Based Source File	2-67
Example — Linking Overlay Memory for an ADSP-2191 System	2-74
Example — Linking an ADSP-219x MP System With Shared Memory	2-77
Example — Overlays Used With ADSP-218x DSPs	2-81

EXPERT LINKER

Expert Linker Overview	3-2
Launching the Create LDF Wizard	3-4
Step 1: Specifying Project Information	3-5
Step 2: Specifying System Information	3-6
Step 3: Completing the LDF Wizard	3-8
Expert Linker Window Overview	3-9
Using the Input Sections Pane	3-10
Using the Input Sections Menu	3-10
Mapping an Input Section to an Output Section	3-12
Viewing Icons and Colors	3-13
Sorting Objects	3-15
Using the Memory Map Pane	3-16
Using the Context Menu	3-17
Tree View Memory Map Representation	3-21
Graphical View Memory Map Representation	3-22

CONTENTS

Specifying Pre- and Post-Link Memory Map View	3-28
Zooming In and Out on the Memory Map	3-28
Inserting a Gap into a Memory Segment	3-32
Working with Overlays	3-33
Viewing Section Contents	3-35
Viewing Symbols	3-37
Managing Object Properties	3-40
Managing Global Properties	3-41
Managing Processor Properties	3-42
Managing PLIT Properties for Overlays	3-44
Managing Elimination Properties	3-45
Managing Symbols Properties	3-47
Managing Memory Segment Properties	3-51
Managing Output Section Properties	3-53
Managing Packing Properties	3-55
Managing Alignment and Fill Properties	3-56
Managing Overlay Properties	3-58
Managing Stack and Heap in DSP Memory	3-60
 ADSP-218X LOADER/SPLITTER	
ADSP-218x Loader Guide	4-2
Boot Types	4-3
Determining Boot Mode	4-4
EPROM Booting (BDMA)	4-6
ADSP-218x Loader BDMA Command-Line Reference	4-8

IDMA - Host Booting	4-13
IDMA Loader Command-Line Reference	4-15
Non-Bootable EPROM Images	4-15
ADSP218x Splitter	4-16
Using the Splitter	4-16
Splitter Command-Line Reference	4-17

ADSP-219X LOADER/SPLITTER

ADSP-219x Booting	5-2
ADSP-219x Boot Kernel	5-2
Boot Modes	5-3
Boot Stream Contents	5-4
Parallel EPROM Booting	5-4
Host Booting	5-7
UART Booting	5-7
Serial EPROM Booting	5-8
No-Boot Mode	5-9
Enriching Boot EPROMs with No-Boot Data	5-13
ADSP-219x Boot Loader Utility	5-16
elfloader.exe Command-Line Reference	5-16

ADSP-2192-12 LOADER

ADSP-2192-12 Boot Loader Utility and Run-Time Boot Loader	6-2
Boot Loader Utility (BLU)	6-3
Building the .DXE Files	6-5

CONTENTS

Running the Run-Time Boot Loader and RTBL	6-6
Run-Time Boot Loader and Overlays Over PCI	6-7
Using OvlPciAdrTbl and OvlMgrTbl Symbols	6-8
ADSP-2192 Boot Loader Utility Command-Line Reference	6-10
Command Line	6-14

ARCHIVER

Archiver Guide	7-2
Creating a Library From VisualDSP++	7-3
File Name Conventions	7-3
Making Archived Functions Usable	7-3
Writing Archive Routines: Creating Entry Points	7-4
Accessing Archived Functions From Your Code	7-5
Archiver File Searches	7-6
Archiver Command-Line Reference	7-7
elfar Command Syntax	7-7
Example elfar Command	7-7
Parameters and Switches	7-8
Constraints	7-9
Archiver Symbol Encryption Reference	7-10

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-6
Assembler Object Files (.DOJ)	A-6
Library Files (.DLB)	A-6
Linker Executable Files (.DXE, .SM, and .OVL)	A-7
Linker Memory Map Files (.MAP)	A-7
Loader Hex Format Files (.LDR, .BNM)	A-7
IDMA Host Boot File (.IDM)	A-10
IDMA Unit Operation	A-10
ASCII Byte-Stream Format	A-11
Debugger Files	A-13
Format References	A-14

UTILITIES

ELF File Dumper	B-1
Using the Archiver and Dumper for Disassembly	B-3
Dumping Overlay Library Files	B-4
AEXE2ELF and ELF2AEXE Converters	B-5
Converting From AEXE to ELF	B-5
Converting From ELF to AEXE	B-6

CONTENTS

LINKER LEGACY SUPPORT

Converting from .SYS to .LDF	C-1
Converting C Code section(“...”) to LDF SECTIONS{}	C-2
Converting Assembler .SEGMENT to LDF SECTIONS{}	C-12
Converting SYS .SEGMENT to LDF Commands	C-20
Legacy Assembly of Sources with Absolute Placement Directives ..	C-22

INDEX

PREFACE

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

Purpose of This Manual

The *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-21xx DSPs* contains information about the linker and utilities for ADSP-21xx processors from Analog Devices 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 (.LDF). The manual leads you through using the linker, archiver, and loader (splitter) to produce DSP programs and provides reference information on the file utility software.

Intended Audience

The primary audience for this manual is DSP programmers who are familiar with Analog Devices DSPs. This manual assumes that the audience has a working knowledge of the appropriate DSP architecture and instruction set. Programmers who are unfamiliar with Analog Devices DSPs can use this manual but should supplement it with other texts, such as *Hardware Reference* and *Instruction Set Reference* manuals, that describe your target architecture.

Manual Contents

The manual contains:

- Chapter 1, “[Linker and Utilities](#)”
Describes each utility. The `linker` command is described.
- Chapter 2, “[Linker Description File](#)”
Describes how to write an `.LDF` file to define the target.
- Chapter 3, “[Expert Linker](#)”
Describes the Expert Linker, an interactive graphical tool that displays and maps DSP memory.
- Chapters 4, 5, and 6, “[Loader](#)”
Describe loading/splitting for ADSP-218x, ADSP-219x, and ADSP-2192-12 DSPs.
- Chapter 7, “[Archiver](#)”
Describes how to combine object files into reusable library files to link routines referenced by other object files.
- Appendix A, “[File Formats](#)”
Describes file formats of the input files for the tools and describes features of output files produced by the tools.
- Appendix B, “[Utilities](#)”
Describes file-conversion and dump utilities.
- Appendix C, “[Linker Legacy Support](#)”
Describes file utilities that support conversion of legacy files.

What's New in This Manual

This edition of the manual documents support for new processors: ADSP-2195, ADSP-2196, and ADSP-2199x. In addition to documenting all existing linker, loader/splitter, and archiver features, this manual describes new switches.

Chapter 3, “[Expert Linker](#)” provides a description of the Expert Linker — a new interactive tool to create a Linker Description File (.LDF) or modify an existing file.

This manual has undergone an extensive reorganization. VisualDSP++ tools manuals contain a Preface. Chapter 1 provides information to help you understand how linking and loading/splitting fits into the DSP development process.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

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

Supported Processors

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

- **ADSP-218x family 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 family 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 Web site, from the product CD-ROM, or from the printed publications (manuals).

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

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices 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 email 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, just log on. Your user name is your email address.

DSP Product Information

For information on digital signal processors, visit our website at www.analog.com/dsp, which provides access to technical publications, datasheets, 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)
+49 (0) 089 76 903 557 (Europe)
- Access the Digital Signal Processor Division's FTP website at
[ftp ftp.analog.com](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++ 3.0 Getting Started Guide for ADSP-21xx DSPs

VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs

Product Information

VisualDSP++ 3.0 Assembler and Preprocessor Manual for ADSP-21xx DSPs

VisualDSP++ 3.0 C Compiler and Library Manual for ADSP-218x DSPs

VisualDSP++ 3.0 C/C++ Compiler and Library Manual for ADSP-219x DSPs

VisualDSP++ 3.0 Product Bulletin for ADSP-21xx DSPs

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 the 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.
.HTM or .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++ manuals in Portable Documentation Format, one .PDF file for each manual. Viewing and printing a .PDF file require a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

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

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

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's **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 shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

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

Using Windows Explorer

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

Using the Windows Start Button

Access VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **VisualDSP**, and **VisualDSP++ Documentation**.

Product Information

From the Web

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

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Manuals

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

Datasheets

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

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

Contacting DSP Publications

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

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

Notation Conventions

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



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



Code has been formatted to fit this manual's page width.

Notation Conventions

Example	Description
Close command (File menu)	Text in bold style indicates the location of an item within the VisualDSP++ environment's menu system. For example, the Close command appears on the File menu.
{this that}	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> .
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .
[this,...]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	A note, providing information of special interest or identifying a related topic. In the online version of this book, the word Note appears instead of this symbol.
	A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word Caution appears instead of this symbol.

1 LINKER AND UTILITIES

This manual describes several ADSP-218x and ADSP-219x program development tools, such as the linker, loader (and splitter), archiver, file-conversion utilities, and ELF file dumper.

This chapter includes:

- [“Software Development Flow” on page 1-2](#) — Shows how linking and loading/splitting fit into the DSP project development process. Describes the linking environment and introduces the Linker Description File (.LDF).
- [“Linking Overview” on page 1-13](#) — Describes the link target, linking environment, and other files used while linking.
- [“Linker Command-Line Reference” on page 1-28](#) — Describes linker command-line switches and their syntax.
- [“Memory Management Using Overlays” on page 1-42](#) — Describes memory overlays as well as advanced LDF commands.

Chapter 2 focuses on the Linker Description File, and chapter 3 describes the Expert Linker.

Software Development Flow

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

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

After running the linker, you test the output with the simulator or emulator. Refer to the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs* and online Help for information about debugging the code.

Finally, you process the debugged executable file(s) through the loader (or splitter) to create output for use on the actual DSP. The output file may reside on another processor (host) or may be burned into a PROM. Chapters 4, 5, and 6 describe options in generating these files. Depending on the DSP, output can be boot-loadable or non-bootable.

[Figure 1-1 on page 1-2](#) compares devices with regard to multiprocessing, boot-loading, and non-bootable files.

Figure 1-1. DSP Features Comparison

DSP	MP	Boot-Loading (output file)	Non-Bootloadable (output file)	Combined (boot-load and Non-Bootloadable)
ADSP-218x	No	elfspl21.exe (.LDR)	elfspl21.exe (.BNU, .BNM, .BNL)	N/A
ADSP-219x ADSP-2199x	No	elfloader.exe (.LDR)	elfloader.exe (.LDR)	elfloader.exe (.LDR)
ADSP-2192-12	Yes	elfloader.exe (.H)	N/A	N/A

The DSP 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 **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 and/or Splitting** — The executable (.DXE), as well as shared memory (.SM) and overlay (.OVL) files, are processed to yield an output file. For ADSP-218x DSPs, this is a .BNM or .IDM file. For ADSP-2192-12 DSPs, this is an .H file. For ADSP-219x DSPs, this is an .LDR file.

Compiling and Assembling

The process starts with source files written in C, C++, or assembly. The compiler (or the 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.

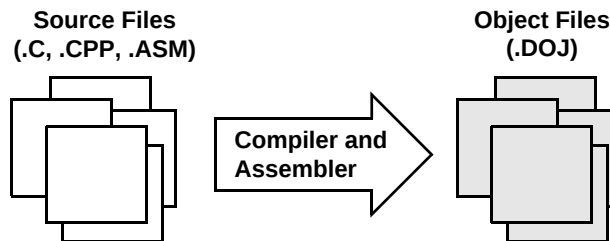


Figure 1-2. Compiling and Assembling

Inputs — C/C++ and Assembly Sources

The first step towards producing an executable is to compile or assemble C, C++, or assembly source files into *object files*. The VisualDSP++ development software assigns a `.D0J` extension to object files ([Figure 1-2](#)).

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++ IDDE 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 DSP system.

Each input section in the LDF requires a unique name, as specified in the source code. Depending on whether the source is C, C++, or assembly, different conventions are used to name an input section (see [“Linker Description File \(.LDF\)” on page 1-17](#)).

Input Section Directives in Assembly Code

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

Example

```
#include      "lib_glob.h"
.SECTION/PM   Lib_Code_Space; /* Section Directive */
.global _abs;
_abs:  rdarg (AX1,1); /* Read arg from stack */
      RTS (DB);
      AR = ABS AX1; /* Take absolute value of input */
      AX1 = AR;
```

In the above example, the VisualDSP++ assembler places the `_abs` global symbol/label and the subsequent code into the `Lib_Code_Space` input section, as it processes this file into object code.

Example

In the following example, the input (`asmFile1.ASM`) to the assembler is:

```
.section/dm data1;
.var array[10];
.section/pm code1;

start:
    ax0 = 0x1234;
    ay0 = 0x5678;
    ar = ax0 + ay0;
```

The output (`asmFile1.D0J`) includes the following two input sections: `data1` and `code1`.

Input Section - data1

```
array [0];
array [1];
...
array [9];
```

Input Section - code1

```
start:
    ax0 = 0x1234;
    ay0 = 0x5678;
    ar = ax0 + ay0;
```

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 compiler uses the section names `program` (for the code) and `data1` (for the data). Additional section names are defined in `.LDF` files.

In C/C++ source files, the optional `section("name")` C language extension may be used to define sections.

Example

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

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

Example

`section ("extern")` in the following example is optional. Note that the new function (`func2`) does not require `section ("extern")`. For more information on LDF sections, refer to [“Specifying the Memory Map” on page 1-25](#).

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

For information on compiler default section names, see your DSP's *VisualDSP++ 3.0 C/C++ Compiler and Library Manual* and column 1 of [Table 1-1 on page 1-26](#).



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

Linking

After you have (compiled and) assembled source files into object files, use the linker to combine the object files into an executable file. By default, the DSP development software gives executable files a `.DXE` extension ([Figure 1-3](#)).

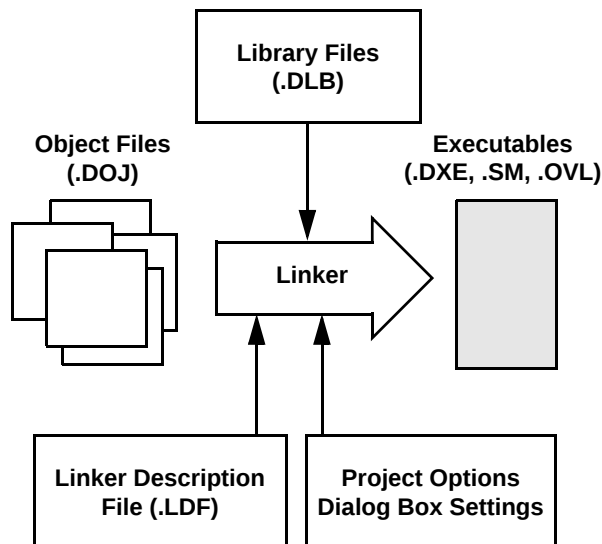


Figure 1-3. Linking

Linking is instrumental in enabling that your code runs efficiently in your target environment. Linking is described in detail in [“Linker Operation” on page 1-14](#).

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

Loading/Splitting

After debugging the .DXE file, you process it through a loader (or splitter) to create an output file for use by the actual DSP. This file may reside on another processor (host) or may be burned into a PROM. Chapters 4, 5, and 6 describe your options. Depending on your DSP system, the output file can be boot-loadable or non-bootable.

[Figure 1-4 on page 1-9](#) shows a simple application of the loader. The loader's input is a single executable (.DXE).

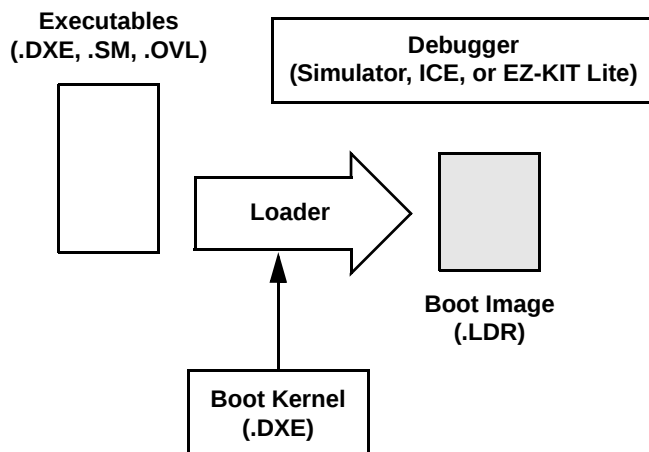


Figure 1-4. Loading/Splitting

For ADSP-218x DSPs, you can use the loader (`elfspl21.exe`) to yield a boot-loadable image (.BNM or .IDM), which is loaded onto memory external to the DSP. When the DSP is reset, the boot-kernel portion of the image is transferred to the DSP's core, and then the instruction and data portion of the image are loaded into the DSP's internal RAM (see [Figure 1-5 on page 1-10](#)) by the boot-kernel.

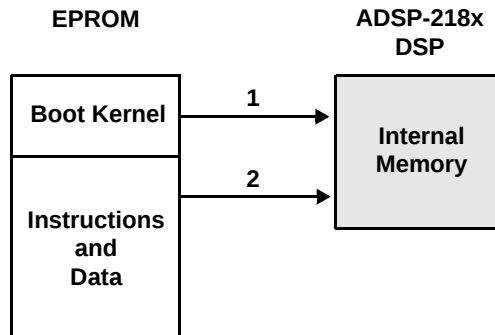


Figure 1-5. ADSP-218x Boot-Loadable Loader (.LDR) File

For ADSP-218x DSPs, you can instead use the splitter (`elfspl21.exe`) to yield a non-bootable image, which is loaded onto memory external to the DSP. As shown in [Figure 1-6 on page 1-11](#), this file has no boot-kernel portion, but is composed of instructions and data. The DSP runs one instruction at a time from the external memory by transferring each individual instruction to the DSP's core as needed. This type of architecture runs slower than the architecture of [Figure 1-5 on page 1-10](#).

For ADSP-219x DSPs, you can choose from three available output image file formats (see [Table 1-7 on page 1-11](#)). The choice is made via a command-line switch to the `elfloader.exe` utility.

[Figure 1-8 on page 1-12](#) shows how multiple ADSP-2192-12 input files — two executables (`.DXE`), a shared memory (`.SM`) file, and overlay files (`.OVL`) — are consumed by the loader to create a single image file (`.LDR`). The example demonstrates the generation of a loader file for a multiprocessor architecture.

 `.SM` and `.OVL` files must reside in the same directory as the input `.DXE` files (or in the current working directory). If your DSP system does not use shared memory or overlays, the files are not required.

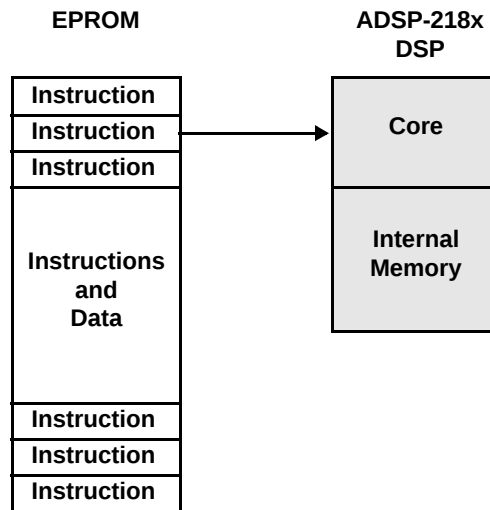


Figure 1-6. ADSP-218x Non-Bootable Loader (.LDR) File

Figure 1-7. ADSP-219x Output Image File Formats

Format	Description
Boot stream file	This .LDR file is booted into DSP internal memory
Non-boot image	This image is run from the EPROM on which it is resident
Combination	Part of the image file is booted into the DSP's internal memory. The other part resides on the EPROM and is run from the EPROM.

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

Refer to chapters 4, 5, and 6 for complete details about loading/splitting for your DSP.

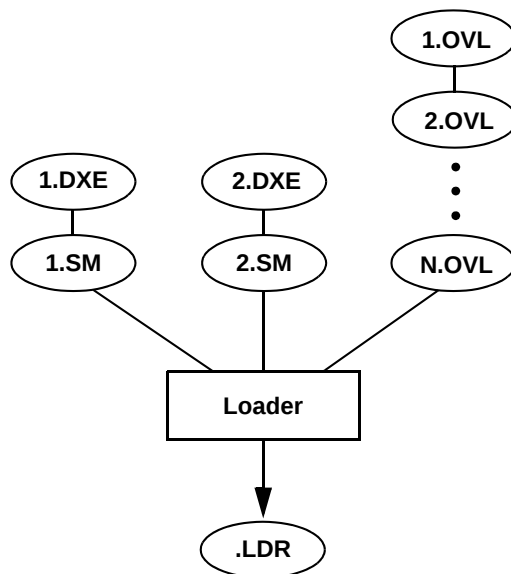


Figure 1-8. Input Files for a Multiprocessor System

Linking Overview

Linking assigns code and data to DSP memory. For a simple single-processor architecture, a single .DXE file may be output. The same linking process may be repeated to create multiple executable files (.DXE) for multiprocessor (MP) architectures. Linking can also produce a shared memory (.SM) file for an MP system. A large executable can be split into a smaller executable and overlays (.OVL files), which contain code that is called in (swapped into internal DSP memory) as needed. [“Memory Management Using Overlays” on page 1-42](#) provides information about advanced topics.

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

Linker Operation

Figure 1-9 on page 1-14 illustrates a basic linking operation. In this figure, several object files (.DOJ) are linked to produce a single executable file (.DXE). The Linker Description File (.LDF) directs the linking process.

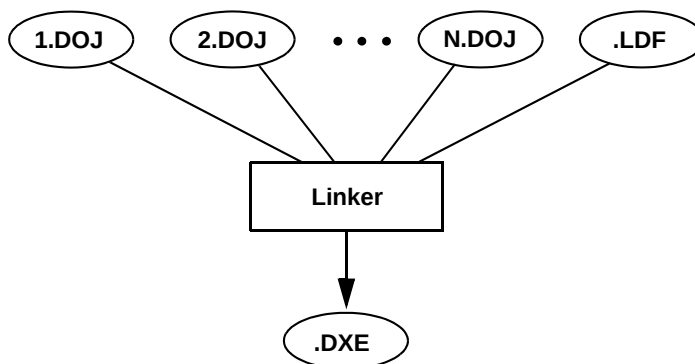


Figure 1-9. Linking Object Files to Produce an Executable File

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

When a DSP architecture is for a multiprocessor system, you generate a .DXE file for each processor. For example, an ADSP-2192-12 contains two cores, and requires two .DXE files. The DSPs in a multiprocessor architecture share memory. When directed by statements in the .LDF file, the linker produce a shared memory executable (.SM) file, the code in which is used by multiple processors.

Overlay files (.OVL), another linker output, support applications that require more program instructions and data than can fit in the processor's internal memory. Refer to [“Memory Management Using Overlays” on page 1-42](#) for more information.

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

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

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

Directing Linker Operation

Linker operations are directed by these options and commands:

- Linker (linker.exe) options (switches)
- .LDF file commands
- **Link** page settings

Linker options control how the linker processes object files and library files. The options specify criteria such as search directories, map file output, and dead-code elimination. You select linker options via linker command-line switches (see [“Linker Command-Line Reference” on page 1-28](#)) or by settings on the **Link** page of the **Project Options** dialog box within the VisualDSP++ environment.

Linking Overview

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

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

Using directives in the .LDF file, the linker:

- Reads input sections in the object files and maps them to output sections in the executable. More than one input section may be placed in an output section.
- Maps each output section in the executable to a *memory segment*, a contiguous range of memory addresses on the target DSP. 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/assemble object(s).
- The compiler and assembler produce object code with labels that direct various portion(s) to particular output sections.

- As directed by the `.LDF` file, the linker maps each input section in the object code to an output section in the `.DxE` file.
- As directed by the `.LDF` file, the linker maps each output section to a memory segment.
- Each input section may contain multiple code items, but a code item may appear in one input section only.
- More than one input section may be placed in an output section.
- Each memory segment must have a specified width.
- Contiguous addresses on different-width hardware must reside in different memory segments.
- More than one output section may map to a memory segment if the output sections fit completely within the memory segment.

Linker Description File (.LDF)

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



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

Linking Overview

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

Refer to [“Linker Description File” on page 2-1](#) for details about this important file.

Linking Environment

The linking environment includes command windows and the VisualDSP++ IDDE. At a minimum, you can run DSP development tools (such as the linker) via a command line and view its output in standard output.

VisualDSP++ provides an environment that simplifies the DSP 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 allows you to choose whether to build a library (.DLB) file, an executable (.EXE) file, or an image file. Error and warning messages display in the **Output** window.

Project Builds

The linker runs from an operating system command line, which you can issue from either the VisualDSP++ IDDE or a command window.

[Figure 1-10](#) shows the VisualDSP++ environment with the **Project** window and an .LDF file open in an **editor** window.

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



Refer to the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs* or online Help for information about the VisualDSP++ environment. Online Help provides powerful search capabilities. To obtain information on a code item, parameter, or error, just select text in an IDDE **editor** window and press the keyboard's **F1** key.

Linking Overview

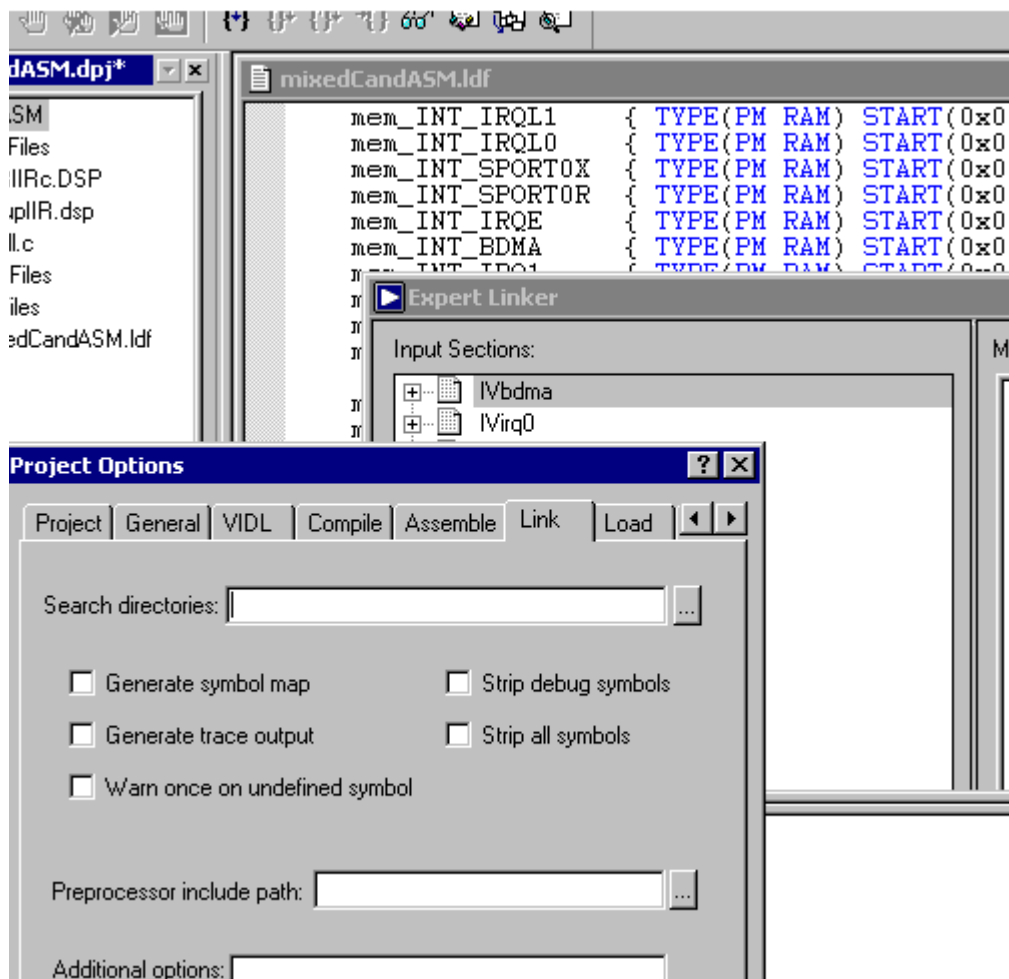


Figure 1-10. VisualDSP++ Environment

Within VisualDSP++, specify tool settings for project builds. You modify linker options via the **Link** page of the **Project Options** dialog box (Figure 1-10).

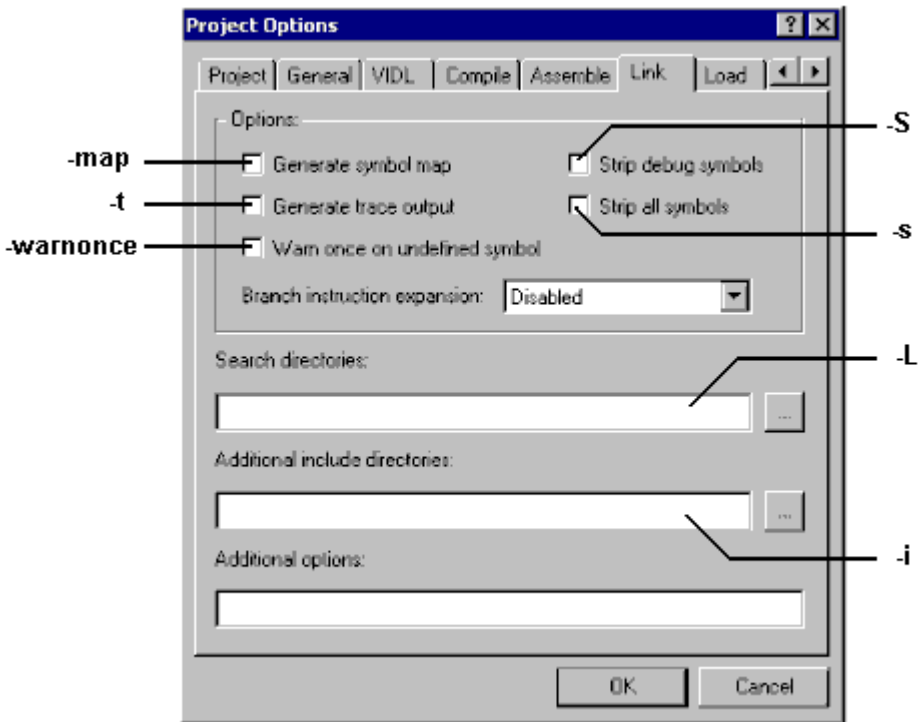


Figure 1-11. Link Page

The callouts in [Figure 1-11](#) refer to the corresponding linker command-line switches. In addition to selecting options on the **Link** page, use the page's **Additional options** field to specify linker switches and parameters that do not have corresponding **Link** page controls. See [“Linker Command-Line Reference” on page 1-28](#) for more information.

Linking Overview

Expert Linker

The VisualDSP++ IDDE features an interactive tool (the *Expert Linker*) to map code or data to specific memory segments. If you are developing a new DSP 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, you can use drag-and-drop techniques to arrange the object files in a graphical memory mapping representation. When you are satisfied with the memory layout, generate the executable file (.DXX).

Figure 1-12 shows the Expert Linker window, which comprises two panes: Input Sections and Memory Map (output sections). Refer to “[Expert Linker](#)” on page 3-1 for information about the Expert Linker.

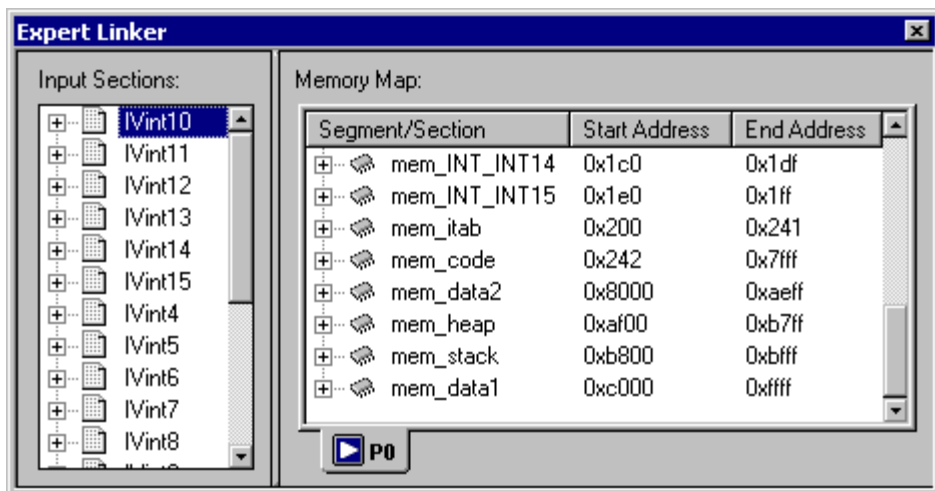


Figure 1-12. Expert Linker Window

Linker Warning and Error Messages

Linker messages are written to the VisualDSP++ **Output** window (or to standard output when you run the linker from a command line). Messages describe problems the linker encountered while processing the .LDF file.

Warnings indicate processing errors that do not prevent the linker from producing a valid output file, such as an unused symbol in your code.

Errors are issued when the linker encounters situations that prevent the production of a valid output file.

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

Example

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

Interpreting Linker Messages

Within VisualDSP++, the **Output** window's **Build** page displays project build status and error messages. In most cases, double-click on a message to view the line in the source file causing the problem. Descriptions of linker messages are accessible from VisualDSP++ online Help.

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

For example, if an input section from the object file is not placed by the LDF, a cross-reference error occurs at every object that refers to labels in the missing section. Fix this problem by reviewing the LDF and specifying all sections that need placement. For more information, see the *VisualDSP++ 3.0 User's Manual for ADSP-21xx DSPs* or online Help.

Link Target Description

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

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

 If the project does not include an `.LDF` file, the linker uses a default LDF that matches the `-Darchitecture` switch on the linker's command line (or the **Processor** option specified on the **Project** page of the **Project Options** dialog box in the VisualDSP++ IDE. The examples in this manual are for ADSP-219x DSPs.

It is assumed that you understand your DSP's memory architecture, which is completely described in the DSP'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 the information in this command 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, PM code, DM data, heap space, and stack space. Refer to [“Specifying the Memory Map”](#).
- **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 by two characteristics: PM or DM, and 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. Use [“Overlay Declaration in LDF” on page 1-46](#) as an example.

For complete information, refer to [“MEMORY{ }” on page 2-27](#).

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 (LDF) for a hypothetical project. This file specifies several memory segments that support the `SECTIONS{ }` command (see [“SECTIONS{ }” on page 2-51](#)).

Linking Overview

The three steps involved in allocating memory are demonstrated below:

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

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

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

[Table 1-1](#) shows how input sections, output sections, and memory segments correspond in the default ADSP-2191 LDF. Refer to your DSP’s default .LDF file and to its *Hardware Reference* manual for details.

Table 1-1. 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

2. **Memory characteristics** — ADSP-218x and ADSP-219x DSPs have a 24-bit addressing range to support memory addresses from 0x0 to 0xFFFFF.

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

3. **Linker MEMORY{} Command** — referring to steps 1 and 2, specify the target's memory with the MEMORY{} command.

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, identifying its memory type (PM or CODE, or DM or DATA) and noting when its 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 named program and data1 for code and data, respectively. These sections correspond to your source when you do not specify a section by means of a section() operator.

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.

Linker Command-Line Reference

This section provides reference information, including:

- [“Command Syntax”](#)
- [“Linker Command-Line Switches” on page 1-33](#)

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

 When you use the linker via the VisualDSP++ IDDE, the settings on the **Link** page (**Project Options** dialog box) correspond to linker command-line switches. VisualDSP++ calls the linker with those switches when you link your code. For more information, see the *VisualDSP++ 3.0 User's Manual for ADSP-21xx DSPs* and VisualDSP++ online Help.

Command Syntax

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

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

The linker command requires `-proc processorID` or a `-T<ldf name>` for the link to proceed. If the command line omits the `-proc processorID`, the LDF file following the `-T` switch must contain a `-proc processorID` command.

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

The following is an example linker command.

```
linker -proc ADSP-2191 p0.doj p1.doj -T target.ldf -t -o
program.dxe
```

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

- [“Command Line Object Files”](#)
- [“Switch Format” on page 1-33](#)
- [“Command-Line File Names” on page 1-30](#)

 Use `-proc processorID` instead of `-Darchitecture` on the command line to select the target processor. See [Table 1-3 on page 1-34](#) for more information.

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.

- The standard object file (`.DOJ`) is produced by the assembler.
- The command line may list 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. Refer to Chapter 6 [“Archiver” on page 7-1](#) for information about archives.
- An executable (`.DXE`) file to be linked against. Refer to `$COMMAND_LINE_LINK_AGAINST` in [“Built-In LDF Macros” on page 2-19](#).

 The linker command line (except for file names) is case sensitive. For example, `linker -t` differs from `linker -T`.

Object File Names

Linker Command-Line Reference

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

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

If the file exists before the link begins, the linker opens the file to verify its type before processing the file. [Table 1-2 on page 1-31](#) lists valid file extensions used by the linker.

Command-Line File Names

Many linker switches take a file name as an optional parameter. [Table 1-2 on page 1-31](#) lists the types of files, names, and extensions that the linker expects on file name arguments.

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

1. Specified path — If the command line includes relative or absolute path information on the command line, the linker searches that location for the file.
2. Specified directories — If you do not include path information on the command line and the file is not in the default directory, the linker searches for the file in the search directories specified with the `-L` (path) command-line switch, and then searches directories

specified by `SEARCH_DIR` commands in the `.LDF` file. Directories are searched in order of appearance on the command line or in the LDF.

3. Default directory — If you do not include path information in the LDF named by the `-T` switch, the linker searches for the LDF in the current working directory. If you use a default LDF (by omitting LDF information in the command line and instead specify `-Darchitecture`), the linker searches in the processor-specific LDF directory; for example, `...\$ADI_DSP\ADSP-219x\ldf`.

For more information on file searches, see [“LDF Macros” on page 2-18](#).

When you provide an input or output file name as a command-line parameter, follow these guidelines.

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

The linker follows the conventions for file name extensions that appear in [Table 1-2](#).

Table 1-2. File Name Extension Conventions

Extension	File Description
.DLB	Library (archive) file
.DOJ	Object file
.DXE	Executable file
.LDF	Linker Description File

Linker Command-Line Reference

Table 1-2. File Name Extension Conventions (Cont'd)

Extension	File Description
.OV	Overlay file
.SM	Shared memory file

Objects

The linker handles an object (file) by its file type, which is determined as follows:

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

The linker treats object (.obj) and library (.lib) files that appear on the command line as object files to be linked. The linker treats executable (.exe) 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` linker macro. Both are described in [“Built-In LDF Macros” on page 2-19](#).

If you do not specify link objects on the command line or in the .LDF file, the linker generates an appropriate informational/error message.

Linker Command-Line Switches

This section describes the linker's command-line switches. [Table 1-3](#) briefly describes each switch with regard to case sensitivity, equivalent switches, switches overridden/contradicted by the one described, and naming and spacing constraints for parameters.

Switch Format

The linker provides switches to select operations and modes. The standard linker 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.

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
```

Linker Command-Line Reference

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

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

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

Switch Summary

[Table 1-3](#) briefly describes each switch. Each individual switch is described in detail following this table.

Table 1-3. Linker Command-Line Switches - Summary

Switch	More Info	Description
<code>@ file</code>	on page 1-36	Uses the specified file as input on the command line.
<code>-Darchitecture</code>	on page 1-36	Specifies the target architecture (processor).
<code>-L path</code>	on page 1-36	Adds the path name to search libraries for objects.
<code>-M</code>	on page 1-36	Produces dependencies.
<code>-MM</code>	on page 1-37	Builds and produces dependencies.
<code>-Map file</code>	on page 1-37	Outputs a map of link symbol information to a file.
<code>-MDmacro[=def]</code>	on page 1-37	Defines and assigns value <code>def</code> to a preprocessor macro.
<code>-S</code>	on page 1-37	Omits debugging symbols from the output file.
<code>-T filename</code>	on page 1-37	Names the LDF.

Table 1-3. Linker Command-Line Switches - Summary (Cont'd)

Switch	More Info	Description
-e	on page 1-38	Eliminates unused symbols from the executable.
-es <i>secName</i>	on page 1-38	Names input sections (<i>secName</i> list) to which elimination algorithm is being applied.
-ev	on page 1-38	Eliminates unused symbols verbosely.
-h -help	on page 1-38	Outputs the list of command-line switches and exits.
-i <i>path</i>	on page 1-38	Includes search directory for preprocessor include files.
-ip	on page 1-38	Fills in fragmented memory with individual data objects that fit. Also requires objects to have been assembled using the assembler's -ip switch.
-jcs21	on page 1-39	(ADSP-219x only) Converts out-of-range short calls and jumps to the longer form.
-keep <i>symName</i>	on page 1-39	Retains unused symbols.
-o <i>filename</i>	on page 1-39	Outputs the named executable file.
-od <i>filename</i>	on page 1-40	Specifies the output directory.
-pp	on page 1-40	Stops after preprocessing.
-proc <i>ProcessorID</i>	on page 1-40	Selects a target processor.
-s	on page 1-40	Strips symbol information from the output file.
-sp	on page 1-40	Skips preprocessing.
-t	on page 1-40	Outputs the names of link objects.
-v	on page 1-41	Verbose. Outputs status information.
-version	on page 1-41	Outputs version information and exits.
-warnonce	on page 1-41	Warns only once for each undefined symbol.
-xref <i>filename</i>	on page 1-41	Outputs a list of all cross-referenced symbols.

Linker Command-Line Reference

@ *file*

Uses *file* as input to the linker command line. This switch circumvents environmental command-line length restrictions. The *file* may not start with “linker” (it cannot be a linker command line). Any white space (including “newline”) in *file* serves to separate tokens.

-*Darchitecture*

Specifies target architecture/processor; for example, -DADSP-2191. White space is not permitted between -D and *architecture*. The architecture entry is case-sensitive and must be available in your VisualDSP++ installation. This switch must be used if no LDF is specified on the command line (see -T). It also must be used if the specified LDF does not specify ARCHITECTURE(). Architectural inconsistency between this switch and the LDF causes an error.



Using -proc *processorID* instead of -*Darchitecture* on the command line is preferential in making the target processor selection.

-L *path*

Adds path name to search libraries and objects. Not case-sensitive; spacing is unimportant. The *path* parameter enables searching for any file, including the LDF itself. May be repeated to add multiple search paths. The paths named with this switch are searched before arguments in the LDF's SEARCH_DIR{} command.

-M

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

-MM

Directs the linker to check a dependency and to output the result to `std-out`, and also to perform the build.

-Map file

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

-MDmacro[=def]

Defines and assigns value *def* to the preprocessor macro named *macro*. For example, `-MDTEST=BAR` executes code following `#ifdef TEST==BAR` in the LDF (but not code following `#ifdef TEST==XXX`). If `=def` is not included, *macro* is defined and set to “1”, so code following `#ifdef TEST` is executed. May be repeated.

-S

Omits debugging symbol information (*not* all symbol information) from the output file. Compare with `-s` switch, [on page 1-40](#).

-T file

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

Linker Command-Line Reference

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

-e

Eliminates unused symbols from the executable.

-es *secName*

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

-ev

Eliminates unused symbols and verbose — report on each symbol eliminated.

-h | -help

Displays a summary of command-line options and exits.

-i *path*

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

-ip

Individual placement. Fills 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, providing more efficient memory packing.

 The `-ip` switch works only with objects assembled with 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 `-noip` option (in assembler) turns off the individual placement option. See the *VDSP++ 3.0 Assembler and Preprocessor Manual for ADSP-218x/219x DSPs*.

-jcs2l

(ADSP-219x only) Converts out-of-range short calls and jumps to the longer form. Refer to the **Branch expansion instruction** option on the **Link** page of the **Project Options** dialog box.

-keep symName

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

-o filename

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

Linker Command-Line Reference

-od filename

Specifies the value of the `$COMMAND_LINE_OUTPUT_DIRECTORY` LDF macro. This allows you to make a command-line change that propagates to many places without changing the `.LDF` file. Refer to [“Built-In LDF Macros” on page 2-19](#)

-pp

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

-proc ProcessorID

Specifies the target processor; for example `-proc ADSP-2191`.

-s

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

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

-sp

Skips preprocessing. Links without preprocessing the LDF.

-t

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

-v

Verbose. Outputs status information while linking.

-version

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

-warnonce

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

-xref *filename*

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

Memory Management Using Overlays

To reduce DSP system costs, many applications employ DSPs with small amounts of on-chip memory and place much of the program code and data off chip. Memory overlays are used to run applications efficiently.

The linker supports the linking of executables for systems with overlay memory. Applications notes (such as EE-67 and EE-152) on the Analog Devices Web site provide detailed descriptions of this technique.

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

- [“Memory Overlays” on page 1-43](#)
- [“Overlay Managers” on page 1-45](#)
- [“Memory Overlay Support” on page 1-46](#)
- [“Example - Managing Two Overlays” on page 1-50](#)
- [“Reducing Overlay Manager Overhead” on page 1-59](#)
- [“Using PLIT{} and Overlay Manager” on page 1-64](#)

The following LDF commands facilitate advanced linker features.

- [“OVERLAY_GROUP{}” on page 2-32](#)
- [“PLIT{}” on page 2-42](#)
- [“MPMEMORY{}” on page 2-30](#)
- [“PACKING\(\)” on page 2-37](#)
- [“PAGE_INPUT\(\)” on page 2-40](#)
- [“PAGE_OUTPUT\(\)” on page 2-41](#)

Memory Overlays

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*.

There are two main types of overlays: *physical overlays* (also called hard overlays) and *BDMA/IDMA* (also called soft overlays). Each of these overlay types has advantages and disadvantages.

Hard overlays were added to ADSP-218x DSPs to increase the amount of RAM that the DSP can address. This was accomplished by creating PMOVLAY and DMOVLAY registers, which allow the DSP to access different physical memory regions mapped to the same address range. Different blocks of PM/DM may be accessed, but only 16K of each block is valid at any given time.



For information about this overlay structure, refer to the *ADSP-2100 Family User's Manual* (available on the Analog Devices DSP Web site).

Soft overlays are not physically present in SRAM at all times, but rather, are transferred dynamically into internal program and/or data memory via the BDMA/IDMA ports at runtime.

Overlays are a “many to one” memory mapping system. Several overlays may “live” (are 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.

Memory Management Using Overlays

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

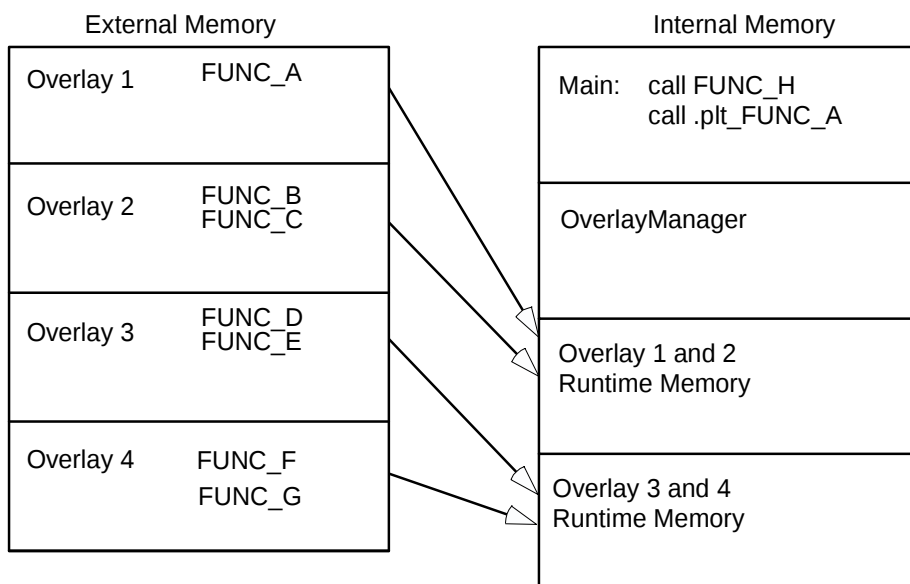


Figure 1-13. Memory Overlays

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

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

Overlay Managers

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

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.

Memory Overlay Support

The overlay support provided by the DSP tools includes:

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

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

[Listing 1-1](#) shows the portion of an LDF that defines two overlays. This overlay declaration configures the two overlays to share a common run-time memory space. `OVERLAY_INPUT` command syntax is described in “`OVERLAY_GROUP{}`” on [page 2-32](#).

- `OVLY_one` contains `FUNC_A` and lives in memory segment `ovl_code`.
- `OVLY_two` contains functions `FUNC_B` and `FUNC_C`. It also lives in memory segment `ovl_code`.

Listing 1-1. Overlay Declaration in LDF

```
.pm_code
{ OVERLAY_INPUT {
    OVERLAY_OUTPUT (OVLY_one.ovl)
    INPUT_SECTIONS (FUNC_A.doj(program))
} >ovl_code

OVERLAY_INPUT
OVERLAY_OUTPUT (OVLY_two.ovl)
INPUT_SECTIONS (FUNC_B.doj(program) FUNC_C.doj(program))
} >ovl_code
} >pm_code
```

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

The LDF configures the overlays and provides the information necessary for the overlay manager to load the overlays. The information provided by the linker 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 are different when internal and external memory widths differ. A system that contains 16-bit external memory may require data packing (depending upon the processor's external bus hardware support) to store an overlay containing instructions in an architecture using, for example, 24-bit instructions. Overlay live word size (number of words in the overlay) is based on the number of 16-bit words required to pack all the 24-bit instructions.



For the ADSP-219x, the software overlay linker-generated constants are 24-bit values (which corresponds to the complete 24-bit address range for the ADSP-219x), so the generated constants must be stored in PM buffers, not DM.

Memory Management Using Overlays

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

You must define PLIT code within the Linker Description File (.LDF). 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 following is an example call instruction to an overlay function:

```
AX0 = DM(I0,M3);
AX1 = AX0 * MR2;
CALL FUNC_A;          /* Call to function in overlay */
DM(I3,M3) = AX1;
```

If FUNC_A is in an overlay, the linker replaces the function call with the following instruction:

```
AX0 = DM(I0,M3);
AX1 = AX0 * MR2;
CALL .plt_FUNC_A;     / * Call to PLIT entry */
DM(I3,M3) = AX1;
```

.plt_FUNC_A is the entry in the PLIT that contains defined instructions. These instructions prepare the overlay manager to load the overlay containing FUNC_A. The instructions executed in the PLIT are specified within the LDF.

When an overlay manager is called via the jump instruction of the PLIT table, AX0 contains the referenced function's overlay ID, and AX1 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.

[Listing 1-2](#) is an example PLIT definition from an LDF, where register AX0 is set to the value of the overlay ID that contains the referenced symbol, and register AX1 is set to the run-time address of the referenced symbol. The last instruction branches to the overlay manager that uses the initialized registers to determine which overlay to load (and where to jump to execute the called overlay function).

Listing 1-2. PLIT Definition in LDF

```
PLIT
{
    AX0 = PLIT_SYMBOL_OVERLAYID;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP OverlayManager;
}
```

The linker expands the PLIT definition into individual entries in a table. An entry is created for each overlay symbol as shown in [Listing 1-3](#). The redirection function calls the PLIT table for overlays 1 and 2. For each entry, the linker replaces the generic assembly instructions with specific instructions (where applicable).

For example, the first PLIT entry in [Listing 1-3](#) 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, AX0 contains the referenced function's overlay ID, and AX1 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.

Memory Management Using Overlays

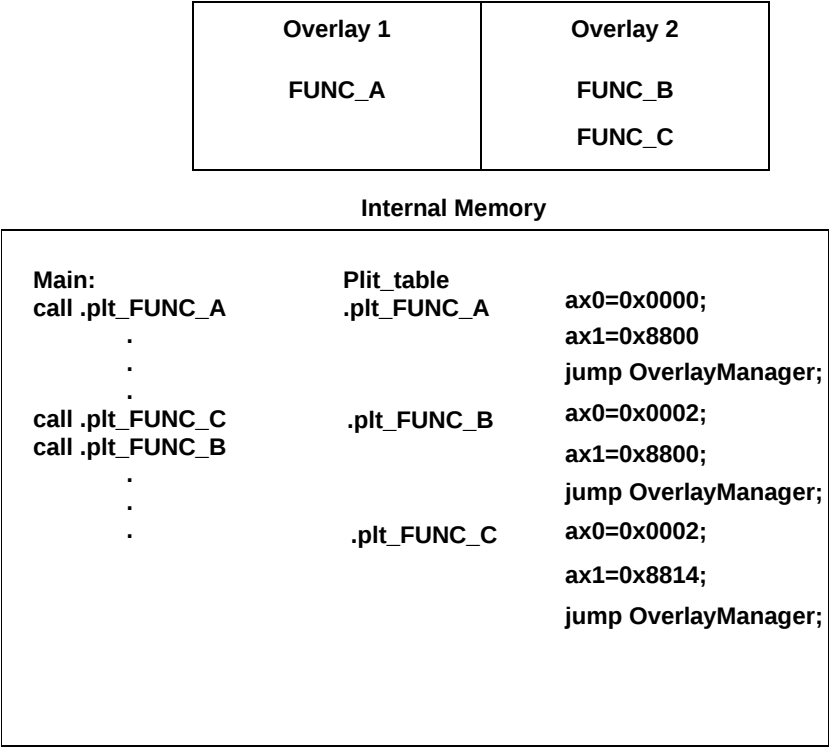


Figure 1-14. Expanded PLIT Table

Example - Managing Two Overlays

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

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 in the text.

Listing 1-3. 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 pm_code. All instruction and data defined in the pm_code memory segment within the Fft_1st_last.doj file are part of the fft_one.ovl overlay. All instructions and data defined in pm_code within the file Fft_mid.doj are part of overlay fft_two.ovl. The result is two functions within each overlay.

Memory Management Using Overlays

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. Since the third function is in overlay `fft_two`, when it 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 already in internal memory, place the overlay ID of each overlay already loaded and the overlay to be loaded in registers, for example `AX0` and `AX1`, and compare:

```
        /* Is overlay already in internal memory? */
AY0 = AY0 -AY1;
        /* If so, do not transfer it in. */
if EQ 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:   idle;
        jump wait;
```

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

```
PLIT
{
```

```

    AX0 = PLIT_SYMBOL_ID;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP OverlayManager;
}

```

Register **AX0** contains the overlay ID that contains the referenced symbol, and register **AX1** contains the run-time address of the referenced symbol. The final instruction jumps the program counter (PC) to the starting address of the overlay manager. The overlay manager uses the overlay ID in conjunction with the overlay constants generated by the linker to transfer the proper overlay into internal memory. Once the transfer is complete, the overlay manager sends the PC to the address of the referenced symbol stored in **AX1**.

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;

```

Memory Management Using Overlays

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.

The overlay live and run word sizes are different when the internal and external memory widths differ. For example, the instruction word width of the ADSP-21xx DSP is 24 bits. A system containing 16-bit wide external memory requires data packing to store an overlay containing instructions. The overlay live word size (number of words in the overlay) is based on the number of 16-bit words required to pack all the 24-bit instructions.

[Figure 1-15 on page 1-56](#) shows the difference between overlay live size and overlay run size. Notice the following differences.

- Overlays 1 and 2 are instruction overlays with a 24-bit run word width. Because external memory is 16 bits, their live word width is 16 bits.
- Overlay 1 contains one function with 16 instructions. Overlay 2 contains two functions with a total of 40 instructions.
- The live word sizes for overlays 1 and 2, respectively, are 24 and 60 words.
- The run word sizes for overlays 1 and 2, respectively, are 16 and 40 words.

These are the linker-generated constants.

```

_ov_startaddress_1      = 0x20000
_ov_startaddress_2      = 0x20018
_ov_endaddress_1        = 0x20017
_ov_endaddress_2        = 0x20023
_ov_word_size_run_1     = 0x10
_ov_word_size_run_2     = 0x28
_ov_word_size_live_1    = 0x18
_ov_word_size_live_2    = 0x3C
_ov_runtimestartaddress_1 = 0x8800
_ov_runtimestartaddress_2 = 0x8800

```

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 (i#) register.

```

.VAR liveAddresses[2] = _ov_startaddress_1,
                        _ov_startaddress_2;
.VAR runAddresses[2]  = _ov_runtimestartaddress_1,
                        _ov_startaddress_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;

```

Before preparing the Direct Memory Access (DMA), the overlay manager stores the values contained in each register it uses onto a run-time stack. The stack stores the values of all data registers, address generator registers, and other registers consumed by the overlay manager.

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

Memory Management Using Overlays

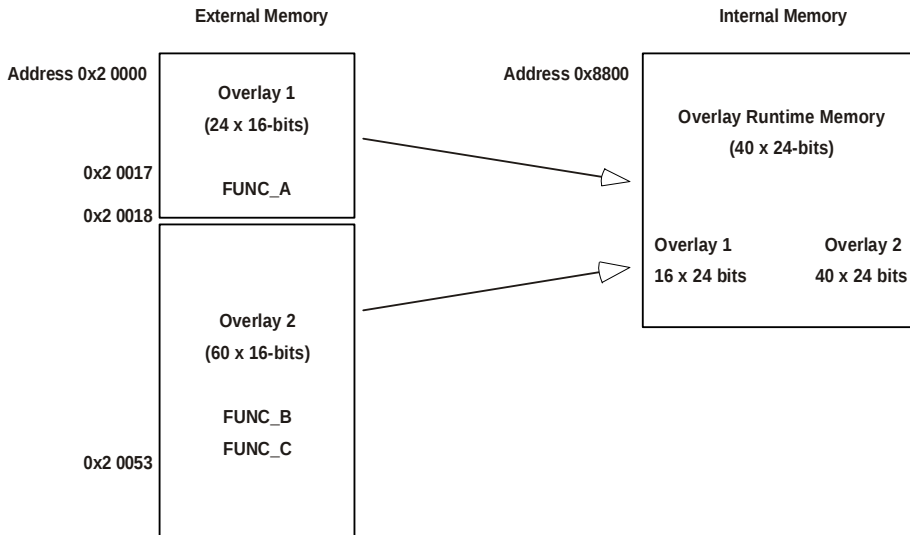


Figure 1-15. Example of Overlay Live and Run Memory Sizes

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.

The following segment of the overlay manager function creates the run-time stack, stores the overlay ID in a modify register, and checks the overlay ID stored in `ov_id_loaded`.

```
/* _overlayID is defined as AX0, set in the PLIT of the LDF.  
   Set up DMA transfer to internal memory. Store values  
   of registers used by the overlay manager on the stack. */  
  
dm(ov_stack) = i1;  
dm(ov_stack+1) = m3;
```

```

dm(ov_stack+2) = AF;
dm(ov_stack+3) = MR2;

/* Use the overlay ID as an index (must subtract one) */
AX0 = AX0-1;          /* Overlay ID -1          */
m3 = AX0;             /* Offset into the arrays containing
                      linker-defined overlay constants. */

MR2 = dm(ov_id_loaded);
AX0 = AX0-MR2;
if EQ jump continue;
dm(ov_id_loaded) = m3;

AX0 = i0;             dm(ov_stack+4) = AX0;
AX0 = m0;             dm(ov_stack+5) = AX0
AX0 = 10;             dm(ov_stack+6) = AX0

```

The overlay manager uses the value of the linker-generated constants to set up the DMA transfer as shown in the following code segment of the overlay manager function.

Index register `i1` points to the first location of the array. The overlay ID is stored in modify register `m3`. Index and modify registers in DAG instructions read the appropriate elements from the array.

```

/* Get overlay run and live addresses from memory */
/* for use in setting up the master mode DMA.      */
i1 = runAddresses;
i0 = liveAddresses;

AX0 = 0;          /* Disable DMA */
dm(DM_ADDR) = AX0;

AX0 = dm(m0,i0);  /* Set DMA external pointer */
dm(DMOVLY) = AX0; /* to overlay live address */

```

Memory Management Using Overlays

```
AX0 = pm(m3,i1);    /* Set DMA internal pointer */
dm(PMOVLY) = AX0;   /* to overlay run address */

i0 = runWordSize;   /* # of workds set up in internal memory */
/* 24-bit word size for instructions */

AX0 = 1;            /* Set DMA external modifier */
dm(IDMAD) = AX0;

i1 = liveWordSize;  /* # of words stored in external memory */
/* Word size is most likely 16 bits */

dm(IDMAA) = AX0;    /* Set DMA internal modify to 1 */

AX0 = dm(m0,i0);    /* Set DMA external count */
dm(PUCR) = AX0;     /* to overlay run size */

AX0 = pm(m3,i1);    /* Set DMA external count */
dm(ASTAT) = AX0;    /* to overlay run size */

/* DMA enabled, instruction word, Master, 16-24 packing */
AX0 = 0x2e1;
dm(DM_ADDR) = AX0;
```

On completion of the transfer, the overlay manager restores register values from the run-time stack, flushes the cache, and then jumps the PC to the run-time location of the referenced function.

It is very important to flush the cache before jumping to the referenced function; when code is replaced or modified, incorrect code execution may occur if the cache is not flushed. If the program sequencer searches

the cache for an instruction, and an instruction from the previous overlay is in the cache, the cached instruction may be executed rather than receiving the expected cache miss.



The cache should only be flushed if cache is enabled. Cache flushing is an optional step for the overlay manager and is done only if cache is enabled in your system.

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

Memory Management Using Overlays

to execute an overlay function that may or may not be in internal memory at the time of the execution request. If the function is not in internal memory a transfer must occur before execution.

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

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

Since the linker handles all overlay symbol references in the same way (jump to PLIT table then overlay manager) it is up to the overlay manager to distinguish between a symbol reference requesting the load of an overlay function and a symbol reference requesting the execution of an overlay function. In the example, the overlay manager uses a buffer in memory as a flag to indicate whether the function call (symbol reference) is a load or an execute request.

The overlay manager first determines whether the referenced symbol is in internal memory. If not, it sets up the DMA transfer. If the symbol is not in internal memory and the flag is set for execution, the core waits for the

transfer to complete (if necessary) and then executes the overlay function. If the symbol is set for load, the core returns to the instructions immediately following the location of the function load reference.

Every overlay function call requires initializing the load/execute flag buffer. Here, the function calls are delayed branch calls. The two slots in the delayed branch contain instructions to initialize the flag buffer. Register AX0 is set to the value that is placed in the flag buffer, and the value in AX0 is stored in memory; 1 indicates a load, and 0 indicates an execution call. At each overlay function call, the load buffer **must** be updated.

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

```
call fft_first_2_stages;
    AX0 = 0;
    dm(prefetch) = AX0;
call fft_middle_stages;
    AX0 = 0;
    dm(prefetch) = AX0;
call fft_next_to_last;
    AX0 = 0;
    dm(prefetch) = AX0;
call fft_last_stage;
    AX0=0;
    dm(prefetch) = AX0;
```

The next set of instructions represents a load function call.

```
call fft_middle_stages;
    /* This function call pre-loads the function */
    /* into the overlay run memory. */
AX0 = 1;
```

Memory Management Using Overlays

```
dm(prefetch) = AX0;  
/* Set pre-fetch flag to 1 to indicate a load. */
```

The implementation 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 is loading into one run-time location, a second overlay function is executing 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(pm_code) )  
        PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)  
    } >ovl_code // Overlay to live in section ovl_code  
    OVERLAY_INPUT  
    {  
        ALGORITHM(ALL_FIT)  
        OVERLAY_OUTPUT(fft_three.ovl)  
        INPUT_SECTIONS( Fft_ovl.doj(pm1_code) )  
        PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)  
    } >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(pm3_code) )
        PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)
    } >ovl_code // Overlay to live in section ovl_code
OVERLAY_INPUT
{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_last.ovl)
    INPUT_SECTIONS( Fft_ovl.doj(pm2_code) )
        PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)
    } >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. By placing an input section between overlay declarations, multiple run-time locations are allocated.

Additional instructions are included to determine whether function call is a load or an execution call. If the function call is a load, the overlay manager initiates the DMA transfer and then jumps the PC back to the location where the call was made. If the call is an execution call, the overlay manager determines whether the overlay is currently in internal memory. If so, the PC jumps to the run-time location of the called function. If the overlay is not in the internal memory, a DMA transfer is initiated and the core waits for the transfer to complete.

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

Memory Management Using Overlays

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

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

Using PLIT{} and Overlay Manager

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

Refer to [“PLIT{}” on page 2-42](#) for basic syntax information. Refer to [“Memory Management Using Overlays” on page 1-42](#) for detailed information on overlays.

[Figure 1-16](#) 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.

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

The two functions in [Figure 1-16](#) 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 1-16. PLITs & Overlay Memory; main() Calls to Overlays

Memory Management Using 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, 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 will first load the overlay that defines the desired symbol, and then branch to that symbol.

Inter-Overlay Calls

PLITs allow you to resolve inter-overlay calls, as shown in [Figure 1-17 on page 1-67](#). Structure the LDF in such a way that 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. Optimize overlays by not placing global variables in overlays. This avoids the difficulty of ensuring the proper overlay is loaded before a global is called.

Inter-Processor Calls

PLITs resolve inter-processor overlay calls, as shown in [Figure 1-18 on page 1-68](#), for DSP systems that permit one processor to access the memory of another processor. When one processor calls into another processor's overlay, the call increases the size of the `.plit` section in the executable that manages the overlay.

Code in Non-Overlay Memory

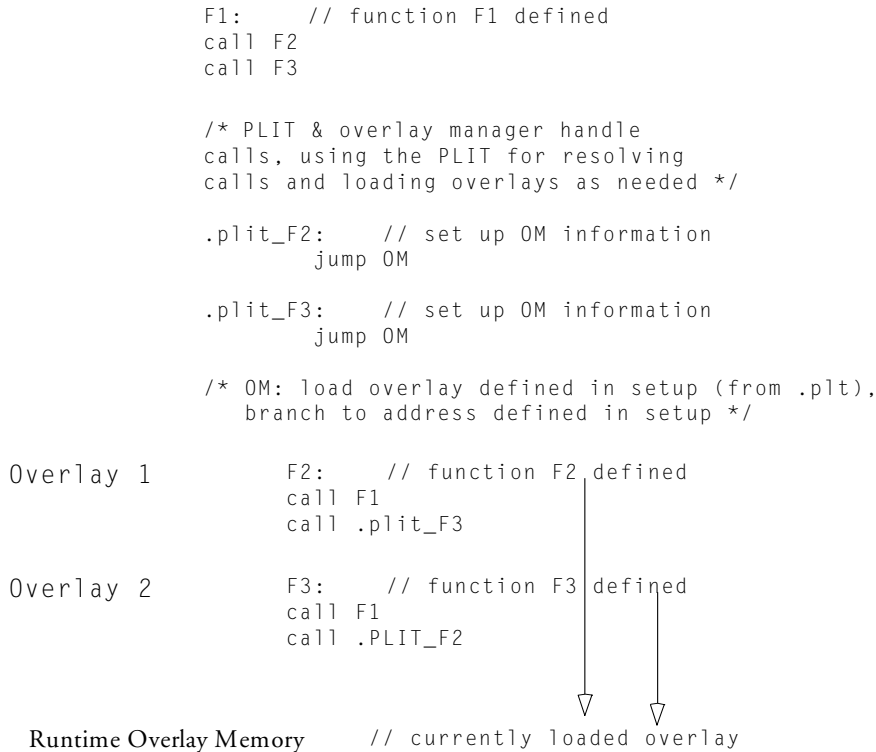


Figure 1-17. PLITs and Overlay Memory; Inter-Overlay Calls

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

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

Memory Management Using Overlays

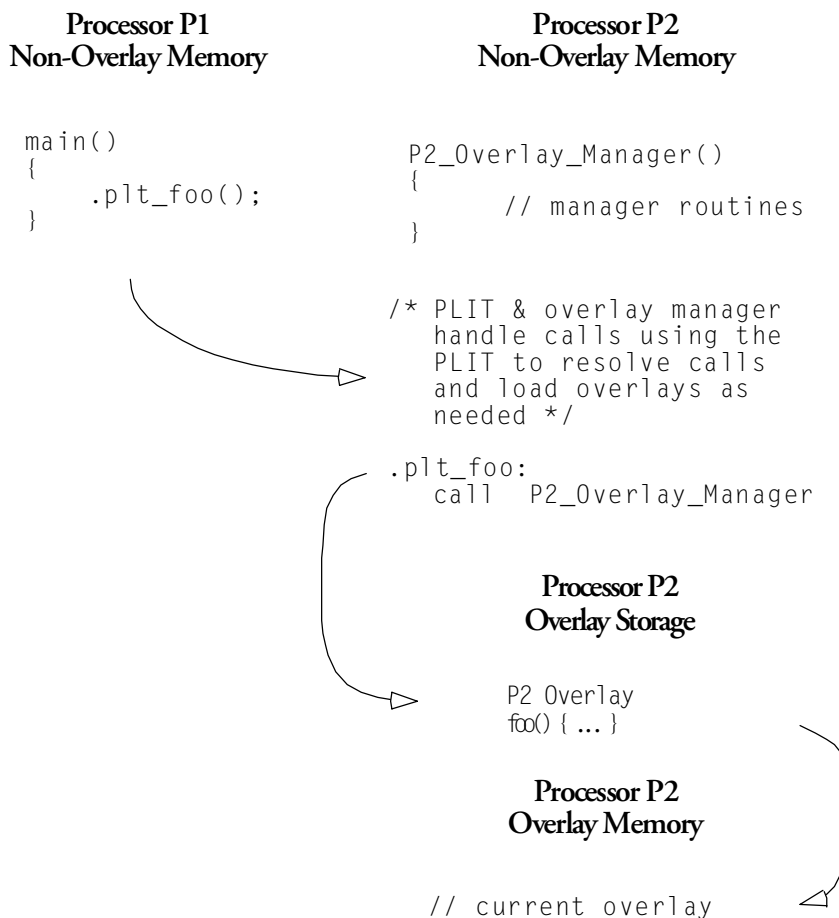


Figure 1-18. PLITs and Overlay Memory - Inter-Processor Calls

2 LINKER DESCRIPTION FILE

Each DSP project requires one Linker Description File (.LDF), which provides the means to precisely specify how to link your project.

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



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

This chapter is an .LDF file reference and includes:

- [“LDF Guide” on page 2-2](#) — Describes Linker Description File syntax, commands, macros, operators, and programming techniques.
- [“LDF Programming Examples” on page 2-59](#) — Provides example LDF files for various system architectures.

LDF Guide

The *Linker Description File* (LDF) allows you to develop code for any DSP system. The LDF defines your system to the linker and specifies how the linker creates executable code for your system. This section describes LDF syntax and provides LDF examples for typical systems.

This section contains:

- [“.LDF File Overview” on page 2-3](#)
- [“LDF Structure” on page 2-9](#)
- [“LDF Expressions” on page 2-11](#)
- [“LDF Keywords, Commands, and Operators” on page 2-12](#)
- [“LDF Operators” on page 2-13](#)
- [“LDF Macros” on page 2-18](#)
- [“LDF Commands” on page 2-21](#)



The linker runs the preprocessor on the LDF, so you can use preprocessor commands (such as `#defines`) within the LDF. For information about preprocessor commands, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for ADSP-21xx DSPs*.

Assembler section declarations in this document correspond to the ADSP-21xx DSP family assembler’s `.SECTION` directive (or the legacy `.MODULE` and `.VAR/SEG=segName` directives).

Refer to [“LDF Programming Examples” on page 2-59](#) and to example DSP programs shipped with VisualDSP++ for example `.LDF` files supporting typical system models.

.LDF File Overview

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

```
symbol = label
```

```
symbol = function_name
```

```
symbol = variable_name
```

If you neither write an LDF nor import an LDF into your project, VisualDSP++ links the code using a default LDF. The default LDF is based on the processor specified in the VisualDSP++ IDE's **Project Options** dialog box. Default LDF files are packaged with your DSP tool distribution kit in a subdirectory specific to your target processor's family (ADSP-218x DSPs or ADSP-219x DSPs). One default LDF is provided for each DSP supported by your VisualDSP++ installation.

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

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



The linker may output warning messages and error messages. You must resolve these messages to enable the linker to produce valid output. See [“Linker Warning and Error Messages” on page 1-23](#) for more information.

Example - Basic .LDF File

[Listing 2-1](#) is an example of a basic .LDF file (formatted for readability). Note the MEMORY{} and SECTIONS{} commands and refer to [“Notes on Basic LDF Example” on page 2-5](#). Other LDF examples for assembly and C source files are in [“LDF Programming Examples” on page 2-59](#).

Listing 2-1. Example LDF

```
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 Example

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

These notes describe features of the `.LDF` file presented in [Listing 2-1 on page 2-4](#).

- `ARCHITECTURE(ADSP-219x)` specifies the target architecture. It dictates possible memory widths and address ranges, the register set, and other structural information for use by the debugger, linker, loader, splitter, and utility software. The target architecture must be installed in VisualDSP++.
- `SEARCH_DIR()` specifies directory paths to be searched for libraries and object files. This example’s argument (`$ADI_DSP\219x\lib`) specifies one search directory. For more information, see [“SEARCH_DIR\(\)” on page 2-50](#).

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

- `$OBJECTS` is an example of a string macro, which expands to a comma-delimited list. A string macro improves readability by replacing long text strings. Conceptually similar to preprocessor macro support (`#defines`) also available in the LDF, string macros

are independent. In this example, `$OBJECTS` is synonymous with `$COMMAND_LINE_OBJECTS`, and expands to a comma-delimited list of the input files to be linked.

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

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

- `$COMMAND_LINE_OBJECTS` (more [on page 2-19](#)) is an LDF *command-line macro*, which expands at the linker command line into the list of input files. Each linker invocation from the VisualDSP++ IDE has a command-line equivalent. When using the VisualDSP++ IDE, `$COMMAND_LINE_OBJECTS` represents the `.DOJ` file of every source file in the VisualDSP++ **Project** window.

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

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

The following examples are functionally identical.

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

```
$DOJS = main.doj, fft.doj;

sec_program {
    INPUT_SECTIONS($DOJS(program))

} >mem_program;
```

- Each `PROCESSOR{}` command ([on page 2-48](#)) generates a single executable.
- The `MEMORY{}` command ([on page 2-27](#)) 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 label, a start and end address (or segment length), a memory width, and a memory type (such as program, data, stack, and so on). Memory segments must have distinct names. These names occupy different namespaces from input section names and output section names. Thus, a memory segment and an output section may have the same name.
- The `OUTPUT()` command ([on page 2-49](#)) produces an executable (.DXE) file and specifies the file name.

In this example, the argument to the `OUTPUT` command is the `$COMMAND_LINE_OUTPUT_FILE` macro ([on page 2-20](#)). The linker names the executable according to the text following the `-o` switch (which corresponds to the name specified in the **Project Options** dialog box when the linker is invoked via the VisualDSP++ IDDE).

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

- `SECTIONS{}` ([on page 2-51](#)) 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 that the linker uses as an input to resolve the mapping to the appropriate memory segment declared in the LDF.

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

For example, in [Listing 2-1 on page 2-4](#), two output sections (`sec_code` and `sec_code2`) are mapped to one memory segment (`seg_code`), as shown below.

```
sec_code {INPUT_SECTIONS($OBJECTS (code))} >seg_code  
sec_code2 {INPUT_SECTIONS($OBJECTS (y_input))} >seg_code
```

The first line places the object code assembled from the `code` input section into the `sec_code` output section and maps the output section to the `seg_code` memory segment. The second line places the object code assembled from the `y_input` input section into the `sec_code2` output section and maps the output section to the `seg_code` memory segment.

The two pieces of code (`sec_code` and `sec_code2`) are adjacent in the `seg_code` memory segment. `INPUT_SECTIONS()` commands are processed in order, so the `seg_code` output section appears first, followed by `y_input`. If there are more than one file in the `y_input` section, the order likely follows the order of the `$OBJECTS` macro, which may in turn depend on the files listed on the command line.

Also note the repetition of the name “`seg_code`” in the first line of code. This demonstrates the independence of the input section, output section, and memory segment namespaces.

LDF Structure

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

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

Declare common (shared) memory definitions in the global LDF scope before the `PROCESSOR{ }` commands. See [“Command Scoping” on page 2-10](#) for more information.

Comments in the .LDF File

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

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

For more information on LDF structure, see [“Link Target Description” on page 1-24](#), [“Placing Code on the Target” on page 1-27](#), and [“LDF Programming Examples” on page 2-59](#).

Command Scoping

There are two LDF scopes: global and command.

A *global scope* occurs outside commands. Commands and expressions that appear in the global scope are always available and are visible in all subsequent scopes. LDF macros are available globally, regardless of the scope in which the macro is defined (see “[LDF Macros](#)” on page 2-18).

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

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

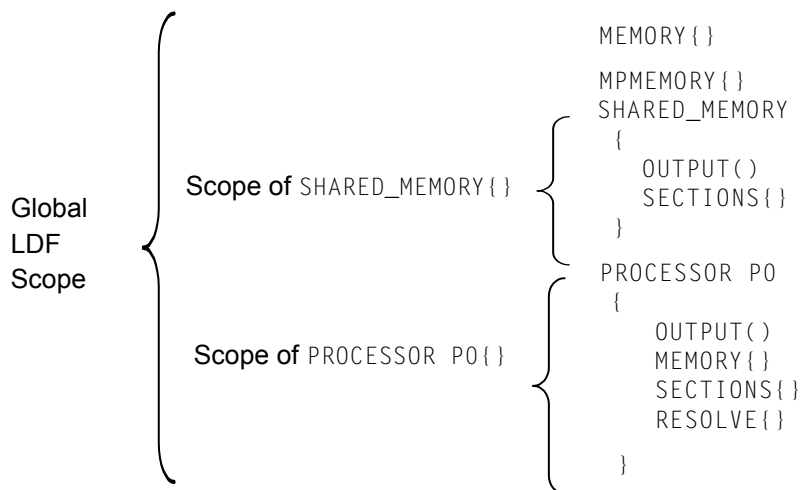


Figure 2-1. LDF Command Scoping Example

LDF Expressions

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

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

Table 2-1. Valid Items in Expressions

Convention	Description
<code>.</code>	(in an address expression) refers to the current location counter (described on page 2-17)
<code>0xnumber</code>	A 0x prefix indicates a hexadecimal number
<code>number</code>	A number without a prefix is a decimal number
<code>number^k</code> (or <code>K</code>)	A decimal number multiplied by 1024
<code>[(B or b)#number</code>	A binary number

LDF Keywords, Commands, and Operators

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

- “Miscellaneous LDF Keywords” on page 2-13
- “LDF Operators” on page 2-13
- “LDF Macros” on page 2-18
- “LDF Commands” on page 2-21



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

Table 2-2. LDF File Keywords Summary

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BOOT	COMAP
DEFINED		
ELIMINATE	ELIMINATE_SECTIONS	END
FALSE	FILL	FIRST_FIT
INCLUDE	INPUT_SECTION_ALIGN	INPUT_SECTIONS
KEEP	LENGTH	LINK_AGAINST
MAP	MEMORY	MEMORY_SIZEOF
NUMBER_OF_OVERLAYS	OUTPUT	
OVERLAY_GROUP	OVERLAY_ID	OVERLAY_INPUT
OVERLAY_OUTPUT	PACKING	PLIT
PLIT_DATA_OVERLAY_IDS	PLIT_SYMBOL_ADDRESS	PLIT_SYMBOL_OVERLAYID
PROCESSOR	RAM	

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

RESOLVE	RESOLVE_LOCALLY	ROM
SEARCH_DIR	SECTIONS	SHARED_MEMORY
SHT_NOBITS	SIZE	SIZEOF
START	TRUE	TYPE
VERBOSE	WIDTH	XREF

Miscellaneous LDF Keywords

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

Table 2-3. Miscellaneous LDF File Keywords

Keyword	Description
BOOT	Boot memory from which a DSP can be booted
DATA	Data memory, the default memory space for all variables
FALSE	A constant with a value of 0
PM	Program memory, the memory space for functions
TRUE	A constant with a value of 1
XREF	A cross-reference option setting

For more information about other .LDF file keywords, see [“LDF Operators” on page 2-13](#), [“LDF Macros” on page 2-18](#) and [“LDF Commands” on page 2-21](#).

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 above 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 32-bit integer constant in one of these forms: hexadecimal, decimal, or decimal optionally followed by “K” (kilo [x1024]) or “M” (Mega [x1024x1024]).
- A period, indicating the current location (see [“Location Counter \(.\)” on page 2-17](#)).

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:

```
Program
{
    INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
}> MEM_PROGRAM

ctor
{
    INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
}> MEM_PROGRAM
```

The LDF may contain the command:

```
ldf_start_ctor = ADDR(ctor)
```

The linker generates the constant `ldf_start_ctor` and assigns it the start address of the `ctor` output section.

DEFINED() Operator

Syntax:

`DEFINED(symbol)`

The linker returns a 1 when the symbol appears in the linker's global symbol table, and returns a 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 defines the global symbol `test`, the following statement sets the constant `test_present` 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 you require a segment's size in order to move the current location counter to an appropriate location.

Example

This example (from a default LDF) 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.

SZIOF() Operator

Syntax:

```
SZIOF(section_name)
```

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

Example

The following LDF fragment defines constant `_sizeofdata1`, whose value is the size of the `data1` section.

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

Location Counter (.)

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

Observe these rules.

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

LDF Macros

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

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

The linker:

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

LDF macros funnel input from the linker command line into predefined macros and provide support for user-defined macro substitutions. Linker macros are available globally in the LDF, regardless of where they are defined. For more information, see [“Command Scoping” on page 2-10](#) and [“LDF Macros and Command-Line Interaction” on page 2-21](#).

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

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

Refer to your DSP's *VisualDSP++ 3.0 C/C++ Compiler and Library* manual and *VisualDSP++ 3.0 Assembler and Preprocessor* manual for more information.

Built-In LDF Macros

The linker provides the following LDF macros.

- `$COMMAND_LINE_OBJECTS`

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

- `$COMMAND_LINE_LINK_AGAINST`

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

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

- `$COMMAND_LINE_OUTPUT_FILE`

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

- `$COMMAND_LINE_OUTPUT_DIRECTORY`

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

```
OVERLAY_OUTPUT ( $COMMAND_LINE_OUTPUT_DIRECTORY\OVL1.OVL )
```

- `$ADI_DSP`

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

User-Defined Macros

The linker supports user-defined macros for file lists. Use the following syntax to define `$macro` as a comma-delimited list of files.

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

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

LDF Macros and Command-Line Interaction

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

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

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

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

LDF Commands

Commands in the `.LDF` file (LDF commands) define the target system and specify the order in which the linker processes output for that system. LDF commands operate within a scope, influencing the operation of other commands that appear within the range of that scope. [For more information, see “Command Scoping” on page 2-10.](#)

The linker supports the following LDF commands:

- [“ALIGN\(\)” on page 2-22](#)
- [“ARCHITECTURE\(\)” on page 2-23](#)
- [“ELIMINATE\(\)” on page 2-23](#)
- [“ELIMINATE_SECTIONS\(\)” on page 2-24](#)

- “INCLUDE()” on page 2-24
- “INPUT_SECTION_ALIGN()” on page 2-24
- “KEEP()” on page 2-26
- “LINK_AGAINST()” on page 2-26
- “MAP()” on page 2-27
- “MEMORY{}” on page 2-27
- “MPMEMORY{}” on page 2-30
- “OVERLAY_GROUP{}” on page 2-32
- “PACKING()” on page 2-37
- “PAGE_INPUT()” on page 2-40
- “PAGE_OUTPUT()” on page 2-41
- “PLIT{}” on page 2-42
- “PROCESSOR{}” on page 2-48
- “RESOLVE()” on page 2-50
- “SEARCH_DIR()” on page 2-50
- “SECTIONS{}” on page 2-51
- “SHARED_MEMORY{}” on page 2-56

ALIGN()

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

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 processor in the target system. An `.LDF` file may contain only one `ARCHITECTURE()` command. The command must appear with global LDF scope, applying to the entire Linker Description File.

The syntax for this command is:

```
ARCHITECTURE(processor)
```

The `ARCHITECTURE()` command is case sensitive. Thus, `ADSP-2191` is valid, but `adsp-2191` is not.

If you do not specify the target processor with the `ARCHITECTURE()` command, the target processor must be identified in the linker command line (`linker -proc <architecture> ...`). Otherwise, the linker cannot link your program. If processor-specific `MEMORY{}` commands in the LDF 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 <target architecture>
```

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 linker's `-e` command-line switch.

ELIMINATE_SECTIONS()

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

INCLUDE()

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

Each LDF must specify the same `Architecture()`, though only one is obligated to specify an architecture. Normally, that is the top-level LDF, which includes the other LDFs.

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*. *number*, which must be a power of 2, is a word boundary (address). Valid values for *number* depend on the word size of the memory segment receiving the output section being aligned.

The linker fills holes created by `INPUT_SECTION_ALIGN()` commands with zeroes (by default), or with the value specified with the preceding `FILL` command valid for the current scope. See `FILL` under “[SECTIONS{}](#)” on [page 2-51](#)

The `INPUT_SECTION_ALIGN()` command is valid only within the scope of an output section. For more information, see [“Command Scoping” on page 2-10](#). For more information on output sections, see the syntax description for [“SECTIONS{” on page 2-51](#).)

Example

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

```
SECTIONS
{
    code
    {
        INPUT_SECTION_ALIGN(4)

        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(program))
        INPUT_SECTIONS ( e.doj(program))

    } >M2Code
}
```

KEEP()

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

LINK_AGAINST()

The `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.

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

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

where:

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

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

Override the search order for a specific variable or label by using the `RESOLVE()` command (see [“RESOLVE\(\)” on page 2-50](#)), which directs the linker to ignore `LINK_AGAINST()` for a specific symbol. `LINK_AGAINST()` for other symbols still applies. Example LDFs that contain the `LINK_AGAINST()` and `RESOLVE()` commands are available in [Listing 2-6 on page 2-63](#).

MAP()

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

The `MAP()` command corresponds to and is overridden by the linker’s `-Map <filename>` command-line switch. In VisualDSP++, if a project’s options (**Link** page of **Project Options** dialog box) specify the generation of a symbol map, the linker runs with `-Map <projectname>.map` asserted, and the LDF’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, you may then use the memory segment names to place program sections via the `SECTIONS{}` command.

The LDF must contain a `MEMORY{}` command for global memory on your target system and may contain a `MEMORY{}` command that applies to each processor’s scope. There is no limit to the number of memory segments you can declare within each `MEMORY{}` command. [For more information, see “Command Scoping” on page 2-10.](#)

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

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

The syntax for the `MEMORY{ }` command appears in [Figure 2-2 on page 2-28](#), followed by a description of each part of a *segment_declaration*.

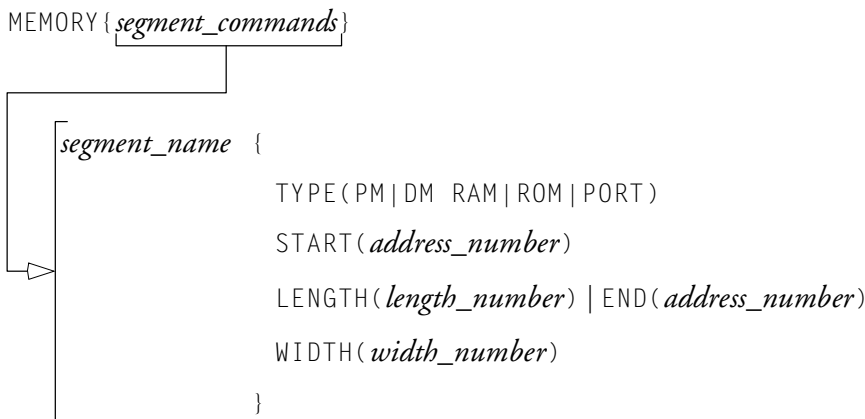


Figure 2-2. Syntax Tree of the `MEMORY{ }` Command

Segment Declarations

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

Each *segment_declaration* must contain a *segment_name*, `TYPE()`, `START()`, `LENGTH()` or `END()`, and a `WIDTH()`. [Table 2-4 on page 2-29](#) describes the make-up of a segment declaration.

Table 2-4. Parts of a Segment Declaration

Item	Description
<i>segment_name</i>	Identifies the memory region. <i>segment_name</i> must start with a letter, underscore, or point, and may include any letters, underscores, digits, and points, and must not conflict with LDF keywords.
TYPE()	Identifies the architecture-specific type of memory within the memory segment. (Note: not all target processors support all types of memory.) The linker stores this information in the executable for use by other development tools. Specify two parameters: memory usage (PM for program memory, or DM for data memory), and functional or hardware locus (RAM or ROM). In ADSP-218x LDF files, the PM/DM declarator specifies the memory type; in ADSP-219x LDF files, they are used only to distinguish between a 16-bit or 24-bit logical data width. The RAM declarator specifies segments that need to be booted. ROM segments are not booted. They are executed/loaded directly from off-chip PROM space.
START(<i>address_number</i>)	Specifies the memory segment's start address. The <i>address_number</i> must be an absolute address or an expression that evaluates to an absolute address.
LENGTH(<i>length_number</i>) or END(<i>address_number</i>)	Identifies the length of the memory segment (in bytes) or specifies the segment's end address. When stating the length, <i>length_number</i> is the number of addressable words within the region or an expression that evaluates to the number of words. When stating the end address, <i>address_number</i> is an absolute address or an expression that evaluates to an absolute address, such as <code>START_1 + LENGTH_1 = END_1</code> .
WIDTH(<i>width_number</i>)	Identifies the physical width (number of bits) of the on-chip or off-chip memory interface. <i>width_number</i> must be a number or an expression that evaluates to a number.

MPMEMORY{}

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

This command specifies the offset of each processor's physical memory in a multiprocessor target system. After declaring the processor names and memory segment offsets with the `MPMEMORY{}` command, the linker uses the offsets during multiprocessor linking. Refer to [“Memory Management Using Overlays” on page 1-42](#) for a detailed description of overlay functionality.

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

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

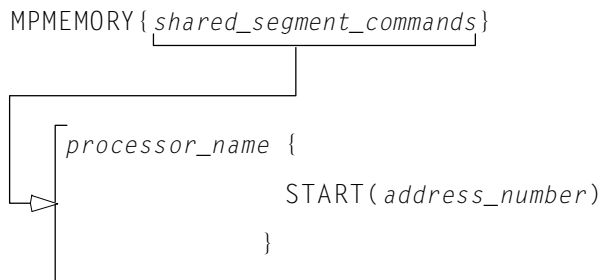


Figure 2-3. `MPMEMORY{}` Command Syntax Tree

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

- *shared_segment_commands*. Contains *processor_name* declarations with a `START{ }` address for each processor's offset in multiprocessor memory. Processor names follow the same rules as any linker label. For more information, refer to [“LDF Expressions” on page 2-11](#).
- *processor_name{placement_commands}* Applies the *processor_name* offset for multiprocessor linking. Refer to [“PROCESSOR{ }” on page 2-48](#) for more information.

OVERLAY_GROUP{}

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 Management Using Overlays” on page 1-42](#) for a detailed description of overlay functionality.

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

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

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

[Figure 2-4 on page 2-33](#) demonstrates grouped overlays.

In the simplified examples in [Listing 2-2 on page 2-35](#) and [Listing 2-3 on page 2-36](#), the functions are written to *overlay files* (`.OVL`). It does not matter whether the functions are disk files or memory segments (except to the DMA transfer that brings them in). Overlays are active only while executing in run-time memory, which is located in the program memory segment.

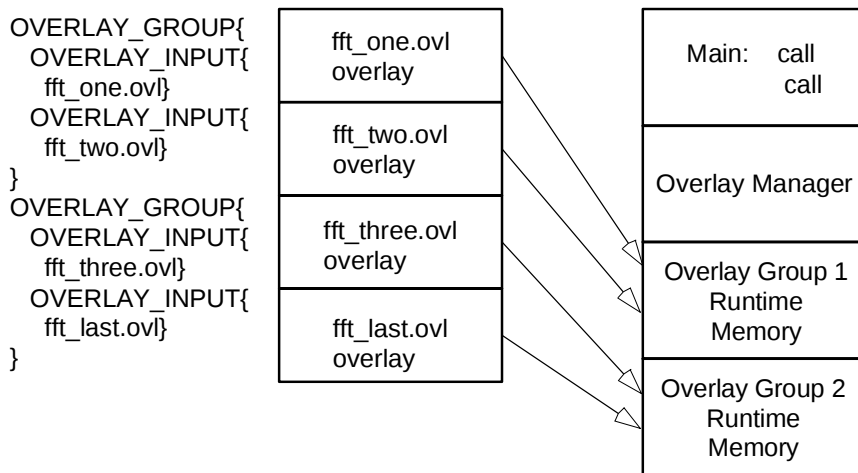


Figure 2-4. Overlays, Grouped

Ungrouped Overlay Execution

In [Listing 2-2 on page 2-35](#), as the FFT progresses, and the overlay functions are called in turn, they are brought into run-time memory in sequence: four function transfers. [“Overlays, Not Grouped” on page 2-34](#) 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.

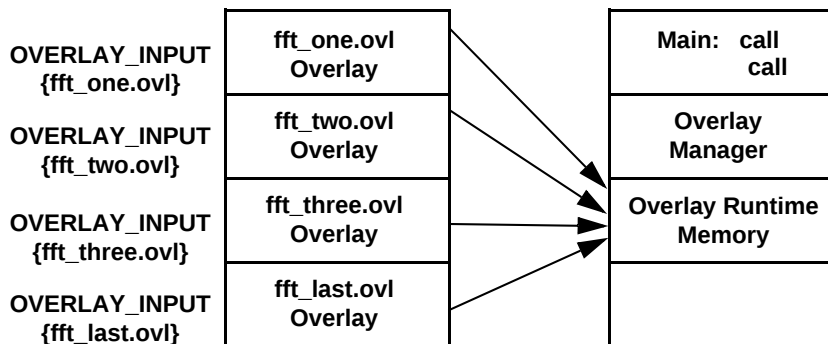


Figure 2-5. Overlays, Not Grouped

Listing 2-2. LDF Overlays, Not Grouped

```
// This is part of the SECTIONS command for processor P0.  
// Declare which functions reside in which overlay.  
// The overlays have been split into different segments  
// in one file, or into different files.  
// The overlays declared in this section (seg_pmco)  
// will run in segment seg_pmco.  
  
OVERLAY_INPUT {      // Overlays to live in section ovl_code  
    ALGORITHM        ( ALL_FIT )  
    OVERLAY_OUTPUT ( fft_one.ovl)  
    INPUT_SECTIONS ( Fft_1st.doj(pmco) ) } >ovl_code  
  
OVERLAY_INPUT {  
    ALGORITHM        ( ALL_FIT )  
    OVERLAY_OUTPUT ( fft_two.ovl)  
    INPUT_SECTIONS ( Fft_2nd.doj(pmco) ) } >ovl_code  
  
OVERLAY_INPUT {  
    ALGORITHM        ( ALL_FIT )  
    OVERLAY_OUTPUT ( fft_three.ovl)  
    INPUT_SECTIONS ( Fft_3rd.doj(pm1_co) ) } >ovl_code  
  
OVERLAY_INPUT {  
    ALGORITHM        ( ALL_FIT )  
    OVERLAY_OUTPUT ( fft_last.ovl)  
    INPUT_SECTIONS ( Fft_last.doj(pm2_co) ) } >ovl_code
```

Grouped Overlay Execution

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

Listing 2-3. LDF Overlays, Grouped

```
OVERLAY_GROUP {      // Declare first overlay group
    OVERLAY_INPUT {   // Overlays to live in section ovl_code
        ALGORITHM      ( ALL_FIT )
        OVERLAY_OUTPUT ( fft_one.ovl)
        INPUT_SECTIONS ( Fft_1st.doj(pm_code) )
    } >ovl_code
    OVERLAY_INPUT {
        ALGORITHM      ( ALL_FIT )
        OVERLAY_OUTPUT ( fft_two.ovl)
        INPUT_SECTIONS ( Fft_mid.doj(pm_code) )
    } >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(pm1_code) )
    } >ovl_code
    OVERLAY_INPUT {
        ALGORITHM      ( ALL_FIT )
        OVERLAY_OUTPUT ( fft_last.ovl)
        INPUT_SECTIONS ( Fft_last.doj(pm2_code) )
    } >ovl_code
}
```

PACKING()

DSPs exchange data with their environments (on-chip or off-chip) through several buses. 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 amounts of memory is implementation-specific. Refer to your DSP's data sheet and *Hardware Reference* manual. Refer to [“Memory Management Using Overlays” on page 1-42](#) for a detailed description of overlay functionality

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 order to place 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, 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, 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 follow these rules.
 - Parameters are whitespace-delimited
 - The total number of non-null bytes is `number_of_bytes`
 - If null bytes are included, they are labeled B0

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 2-5. Unpacked Order vs. Native Storage Order

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

Efficient Packing

The packing in [“Unpacked Order vs. Native Storage Order” on page 2-38](#) 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 [“Unpacked Order vs. Native Storage Order” on page 2-38](#), it is far more simple to program memory access by directing the linker to add unused (null) bytes to your executable. This simplifies alignment and sequencing.

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

B2, B1, B4, B3, B6, B5

Say 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, generating 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 “live” external memory. Use an explicit `PACKING()` command when the width or byte order of stored data (executables or overlays in “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, 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 when you change your target or memory configuration.

PAGE_INPUT()

The `PAGE_INPUT()` command supports paged memory architectures for ADSP-218x DSPs.

Use `PAGE_INPUT()` to specify overlays for memory pages. This command can be used instead of `OVERLAY_INPUT` (see [“OVERLAY_GROUP{}” on page 2-32](#)) in any location in the `.LDF` file. Refer to [“Memory Management Using Overlays” on page 1-42](#) 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 based on the type of memory selected for “run” and “live” space for the overlay. The linker determines the page value 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 there is no internal memory at that address (for example, 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 on 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 2-32](#)) in any location in the .LDF file. Refer to [“Memory Management Using Overlays” on page 1-42](#) for a detailed description of overlay functionality.

PLIT{}

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

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

`PLIT{}` commands provide a template from which the linker generates assembly code when a symbol resolves to a function in overlay memory. The code typically handles a call to a function in overlay memory by calling an overlay memory manager. Refer to [“Memory Management Using Overlays” on page 1-42](#) for a detailed description of overlay and `PLIT` functionality.

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

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

PLIT Syntax

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

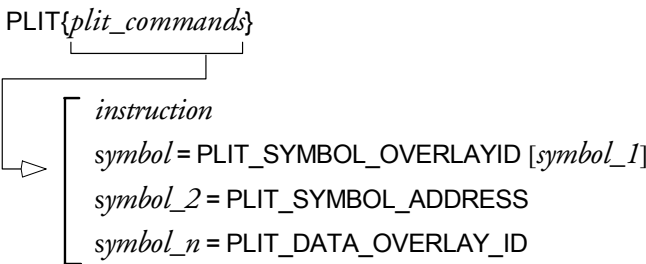


Figure 2-6. Syntax Tree of the `PLIT{}` Command

Table 2-6. Parts of the `PLIT` Command

Item	Description
<i>instruction</i>	None, one, or multiple assembly instructions. The instructions may occur in any reasonable order in the command structure and may precede or follow symbols. The following two constants contain information about <i>symbol</i> and the overlay in which it occurs. You must supply instructions to handle that information.
PLIT_SYMBOL_OVERLAYID	Returns the overlay ID of the resolved symbol. <i>symbol_1</i> is a register name.
PLIT_SYMBOL_ADDRESS	Returns the absolute address of the resolved symbol in run-time memory. <i>symbol_2</i> is a register name.
PLIT_DATA_OVERLAY_ID	Returns the address of an array containing the IDs of overlays that hold data used by the resolved symbol's function. The array terminates with the null ID "0". <i>symbol_n</i> is typically a register name or memory location, which is loaded with that (start) address.

Command Evaluation and Setup

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

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

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

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

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

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

Allocating Space for PLITs

The `.LDF` file must allocate space in memory to hold PLITs built by the linker. Typically, that memory resides in the program code (program¹) memory segment. A typical LDF declaration for that purpose appears below.

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

¹ Whatever you name your program code Memory Segment.

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

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

PLIT Examples

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

Simple PLIT – States are Not Saved

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

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

```
PLIT
{
    AX0 = PLIT_SYMBOL_OVERLAY_IDS;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

A PLIT that Saves Register Contents

The following `PLIT{}` command saves the contents of `AX0` and `AX1` in data memory before being overwritten.

```
PLIT
{
    DM(save_AX0) = AX0;
    DM(save_AX1) = AX1;
    AX0 = PLIT_SYMBOL_OVERLAYID;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

As a general case, minimize overlay transfer traffic. Design code so overlay functions are imported and use minimal (perhaps zero) reloading to improve performance.

What PLIT Does – 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. It 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 to write an overlay manager, the linker generates `PLIT_SYMBOL` constants for each symbol in an overlay.

The overlay manager can also:

- Be helped by manual intervention. Save the target's state on the stack or in memory before loading and executing an overlay function, so it continues correctly on return. However, you can

implement this feature within the PLIT section of your LDF.

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

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

PROCESSOR{}

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

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

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

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

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

The `PROCESSOR{}` command syntax is defined as follows:

- `processor_name`

Assigns a name to the processor. Processor names follow the same rules as linker labels. [For more information, see “LDF Expressions” on page 2-11.](#)

- `OUTPUT(file_name.DXE)`

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

- `MEMORY{segment_commands}`

Defines memory segments that apply only to this processor. Use command scoping to define these memory segments outside the `PROCESSOR{}` command. [For more information, see “Command Scoping” on page 2-10 and “MEMORY{” on page 2-27.](#)

- `PLIT{plit_commands}`

Defines procedure linkage table (PLIT) commands that apply only to this processor. [For more information, see “PLIT{” on page 2-42.](#)

- `SECTIONS{section_commands}`

Defines sections for placement within the executable (.DXE). [For more information, see “SECTIONS{” on page 2-51.](#)

- `RESOLVE(symbol, resolver)`

Use this command to ignore a `LINK_AGAINST()` command. For details, see [“RESOLVE{” on page 2-50.](#)

RESOLVE()

Use the `RESOLVE(symbol_name, resolver)` command, which ignores a `LINK_AGAINST()` for a specific symbol. This overrides the search order for a specific variable or label. Refer to the [“LINK_AGAINST\(\)” on page 2-26](#) for more information.

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



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

SEARCH_DIR()

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

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

Example

```
ARCHITECTURE (ADSP-2191)
MAP (SINGLE-PROCESSOR.MAP)           // Generate a MAP file

SEARCH_DIR( $ADI_DSP\219x\lib; ABC\XYZ )
// $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
// 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 2-8 shows syntax for the SECTIONS{} command.

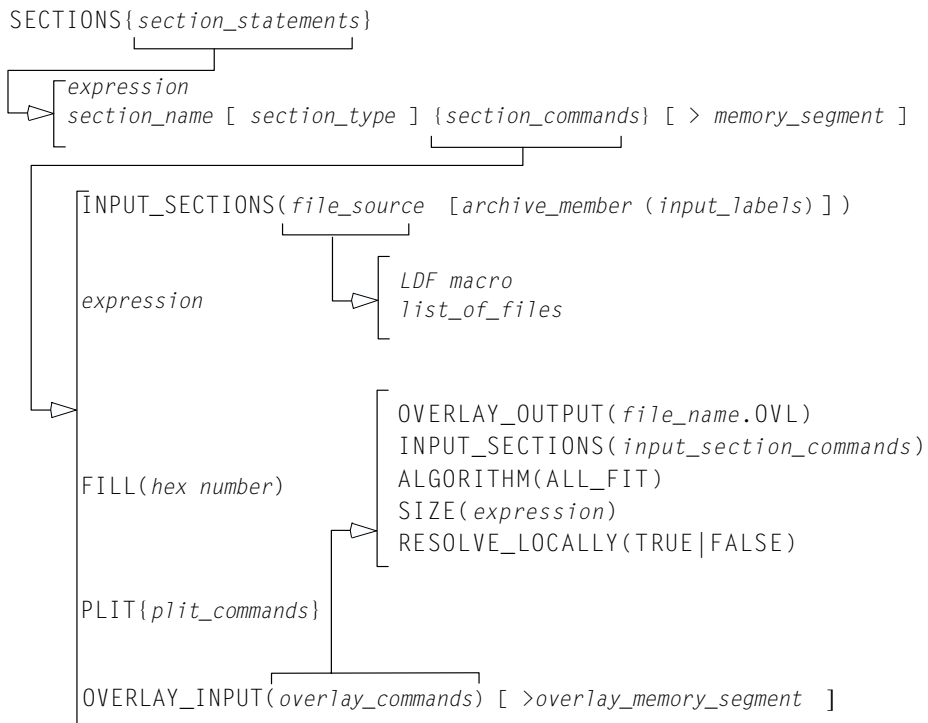


Figure 2-8. Syntax Tree of the SECTIONS{} Command

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

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

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

expressions

or

section_declarations

Use *expressions* to manipulate symbols or to position the current location counter. Refer to [“LDF Expressions” on page 2-11](#).

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

Figure 2-9. Parts of a Section Declaration

Item	Description
<i>section_name</i>	Must start with a letter, underscore, or period and may include any letters, underscores, digits, and points. A <i>section_name</i> must not conflict with any LDF keywords. The special <i>section_name</i> , <code>.plt</code> , indicates the procedure linkage table (PLT) section that the linker generates when resolving symbols in overlay memory. Place this section in non-overlay memory to manage references to items in overlay memory.
<i>section_type</i>	(optional) assigns an ELF section type. The only valid section type keyword is <code>SHT_NOBITS</code> (section header type no bits). This section type contains uninitialized data, so even if it is large, it can download quickly. Space is allocated but not written. For an example of <code>SHT_NOBITS</code> , see Listing 2-6 on page 2-63 .

Figure 2-9. Parts of a Section Declaration

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

INPUT_SECTIONS()

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

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

expression

In a *section_command*, an *expression* manipulates symbols or positions the current location counter. See [“LDF Expressions” on page 2-11](#) for details.

FILL(hex number)

In a *section_command*, a `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 zeroes. Specify only one `FILL()` command per output section. For example,

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

PLIT{plit_commands}

In a *section_command*, a `PLIT{}` command declares a locally scoped procedure linkage table (PLIT). It contains its own labels and expressions. [For more information, see “PLIT{” on page 2-42.](#)

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 [“Example — Linking Overlay Memory for an ADSP-2191 System” on page 2-74.](#)

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

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

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

The *overlay_commands* portion of an `OVERLAY_INPUT()` command follows these rules.

- `OVERLAY_OUTPUT()` outputs an overlay file (`.OVL`) for the overlay with the specified name. The `OVERLAY_OUTPUT()` in an `OVERLAY_INPUT()` command must appear before any `INPUT_SECTIONS()` for that overlay.
- `INPUT_SECTIONS()` has the same syntax within an `OVERLAY_INPUT()` command as when it appears within an *output_section_command*, except that a `.PLIT` section may not be placed in overlay memory. [For more information, see “INPUT_SECTIONS\(\)” on page 2-53.](#)
- `OVERLAY_ID()` returns the overlay ID of the resolved symbol.
- (not currently available) `NUMBER_OF_OVERLAYS()` returns the number of overlays that the current link generates when using the `FIRST_FIT` or `BEST_FIT` overlay-placement `ALGORITHM()`.
- `ALGORITHM()` directs the linker to use the specified overlay linking algorithm. The only currently available linking algorithm is `ALL_FIT`.

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

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

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

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

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

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

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

SHARED_MEMORY{ }



`SHARED_MEMORY{ }` is used only with ADSP-2192 DSPs.

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

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

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

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

The syntax for the `SHARED_MEMORY{ }` command appears in [Figure 2-10](#) followed by definitions of its components in [Table 2-7 on page 2-57](#).

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

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

Figure 2-10. SHARED_MEMORY{} Command Syntax

Table 2-7. Parts of the SHARED_MEMORY{} Command

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

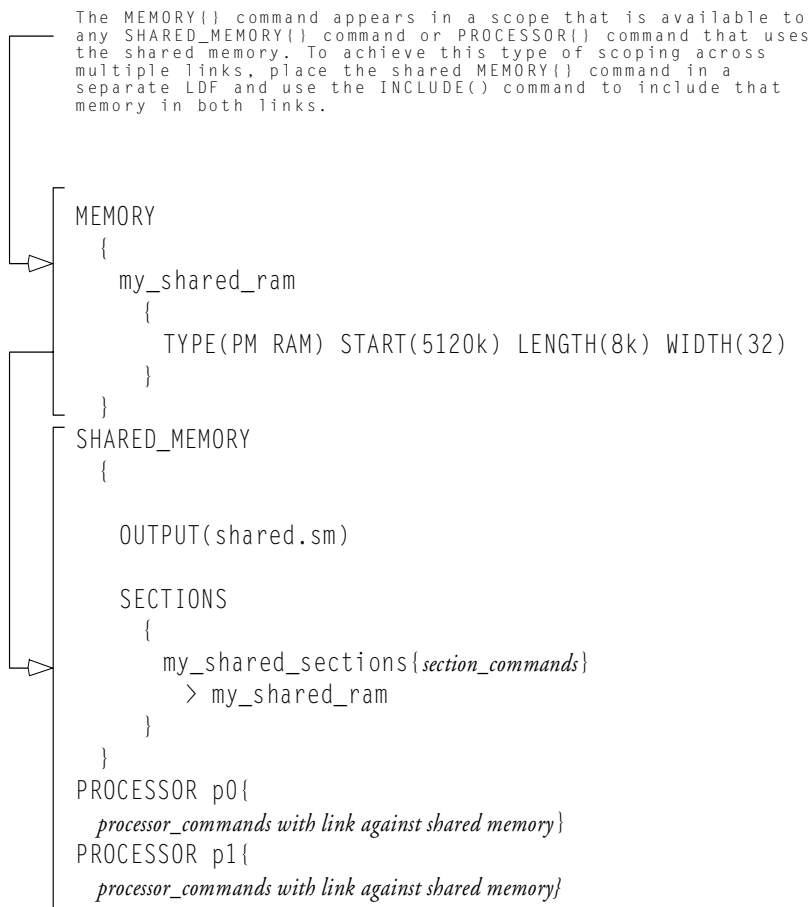


Figure 2-11. LDF Scopes for SHARED_MEMORY{}

LDF Programming Examples

This section shows several typical LDFs. As you modify these examples, refer to the syntax descriptions in [“LDF Commands” on page 2-21](#).

This section provides the following examples:

- [“Example — Linking for a Single-Processor ADSP-219x System” on page 2-61](#)
- [“Example — Linking Large Uninitialized Variables” on page 2-63](#)
- [“Example — Linking an Assembly Source File” on page 2-65](#)
- [“Example — Linking a Simple C-Based Source File” on page 2-67](#)

LDF Programming Examples

- “Example — Linking Overlay Memory for an ADSP-2191 System” on page 2-74
- “Example — Linking an ADSP-219x MP System With Shared Memory” on page 2-77
- “Example — Overlays Used With ADSP-218x DSPs” on page 2-81



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. Examples are in the following directories.

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

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



A variety of per-processor default .LDF files come with the development software, providing 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
```

Example — Linking for a Single-Processor ADSP-219x System

When linking an executable for a single-processor system, the LDF describes the processor's memory and places code for that processor. The LDF in [Listing 2-4](#) 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
- `$OBS` and `$LIBS` macros get object (`.DOJ`) and library (`.DLB`) file input
- `MAP()` outputs a map file
- `MEMORY{}` defines memory for the processor
- `PROCESSOR{}` and `SECTIONS{}` defines a processor and place program sections for that processor's output file, using the memory definitions

Listing 2-4. 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
```

LDF Programming Examples

```
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
```

Example — Linking Large Uninitialized Variables

When linking an executable file that contains large uninitialized variables, use the `SHT_NOBITS` section qualifier (section header type no bits) 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 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. Long download times may occur when booting from a loader file (because of the increased file size).

[Listing 2-6](#) shows the use of the `SHT_NOBITS` section to avoid initialization of a segment.

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



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

Listing 2-5. Large Uninitialized Variables: Assembly Source

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

Listing 2-6. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-219x)
$OBJECTS = $COMMAND_LINE_OBJECTS; // Libraries & objects from
```

LDF Programming Examples

```
                                // 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 )
    // SHT_NOBITS section is not written to output file
    SECTION {
        extram_output SHT_NOBITS {
            INPUT_SECTIONS ( $OBJECTS ( extram_area ) ) } >mem_extram;
        } //end section
    } // end processor P0
```

Example — Linking an Assembly Source File

[Listing 2-8 on page 2-66](#) shows an example .LDF (for an ADSP-2191) that describes a simple memory placement of an assembly source file ([Listing 2-7](#)). 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 2-7. 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.

LDF Programming Examples

Listing 2-8. 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
```

Example — Linking a Simple C-Based Source File

[Listing 2-10](#) shows an example LDF that describes the memory placement of a simple C source file ([Listing 2-9](#)).

Listing 2-9. Simple C Source File

```
int x_input[256];

main()
{
    int i;

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

    } // end main
```

Listing 2-10. 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"
```

LDF Programming Examples

```
// 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, libcppprt.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 facilitate 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) }
```

LDF Programming Examples

```
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

    // The 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 ( $OBJECTS(IVpwrdown)$LIBRARIES(IVpwrdown ) )
        } > 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 ) )
```

LDF Programming Examples

```
        } > 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 {
```

LDF Programming Examples

```
        __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
```

Example — Linking Overlay Memory for an ADSP-2191 System

When linking 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 2-11](#) shows an example overlay memory .LDF. For more information, see the comments in the listing.

Listing 2-11. 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{
        dx_seg_rth{
```

LDF Programming Examples

```
INPUT_SECTIONS("2191_ASM_Interrupt_Table.doj"(interrupts))
} >mem_seg_rth

dxseg_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

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

dxseg_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

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

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

Example — Linking an ADSP-219x MP System With Shared Memory

When linking 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 2-12](#) shows a multiprocessor system and shared memory LDF.

Listing 2-12. 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”
```

LDF Programming Examples

```
// 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 {
        dxm_pmco
            { INPUT_SECTIONS($PSH0_OBJECTS(seg_pmco)
                          $LIBRARIES(seg_pmco)) }
            > seg_pmco
        dxm_pmda
            { INPUT_SECTIONS($PSH0_OBJECTS(seg_pmda)
                          $LIBRARIES(seg_pmda)) }
            > seg_pmda
        dxm_dmda
            { INPUT_SECTIONS($PSH0_OBJECTS(seg_dmda)
                          $LIBRARIES(seg_dmda)) }
            > seg_dmda
        dxm_init
            { INPUT_SECTIONS($PSH0_OBJECTS(seg_init)
                          $LIBRARIES(seg_init)) }
            > seg_init
        dxm_rth
            { INPUT_SECTIONS($PSH0_OBJECTS(seg_rth)
```

LDF Programming Examples

```

        $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
}

```

Example — 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 (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

```

LDF Programming Examples

```
.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
```


3 EXPERT LINKER

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

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

This chapter contains:

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

Expert Linker Overview

The Expert Linker (EL) 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

EL 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 (`.DXX`) 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.

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

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

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

The Expert Linker window appears.

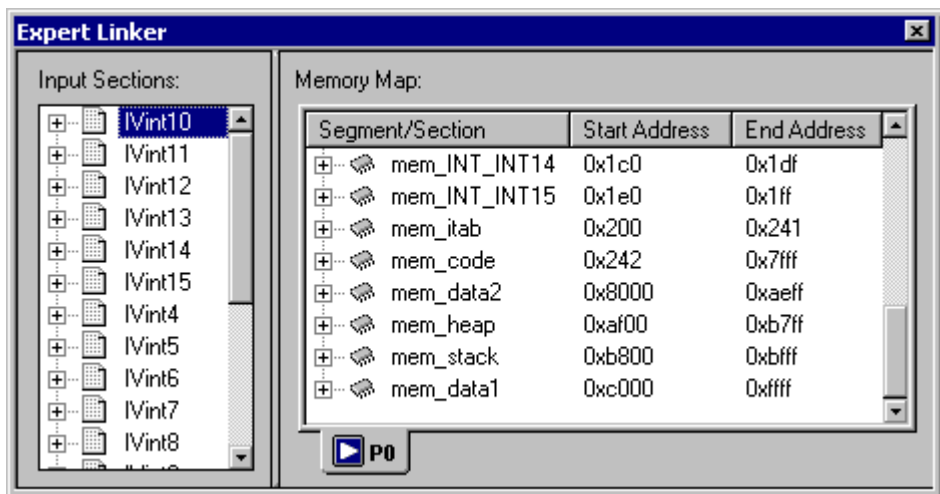


Figure 3-1. Expert Linker Window

Launching the Create LDF Wizard

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

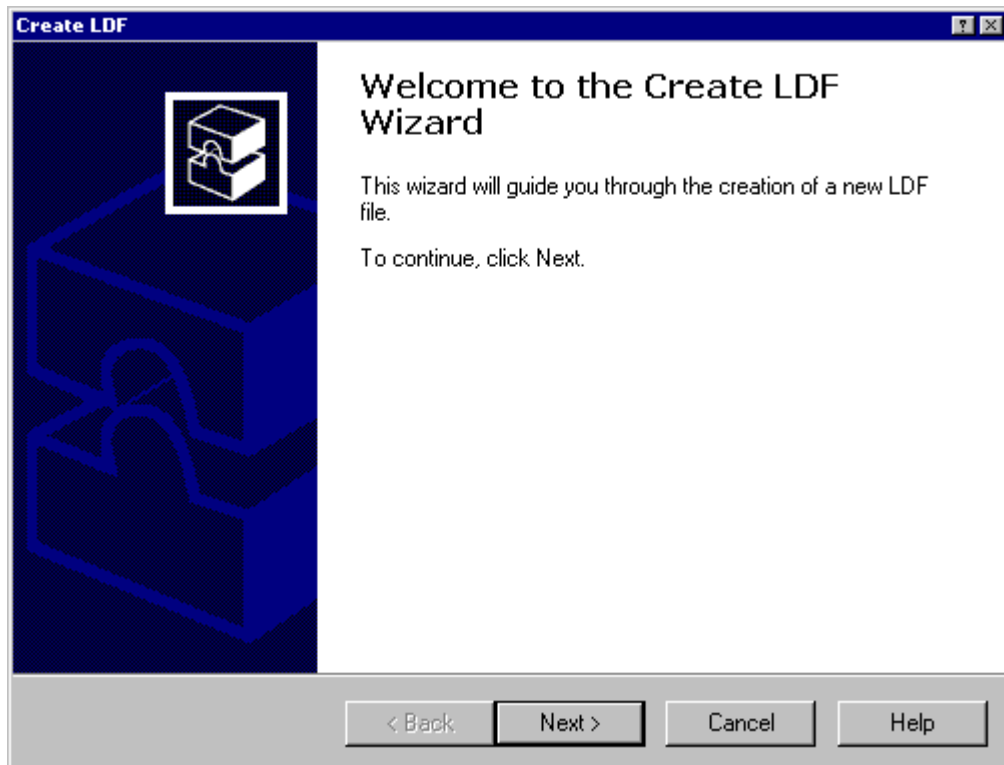


Figure 3-2. Welcome Page of the Create LDF Wizard

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

Step 1: Specifying Project Information

The first wizard window is displayed.

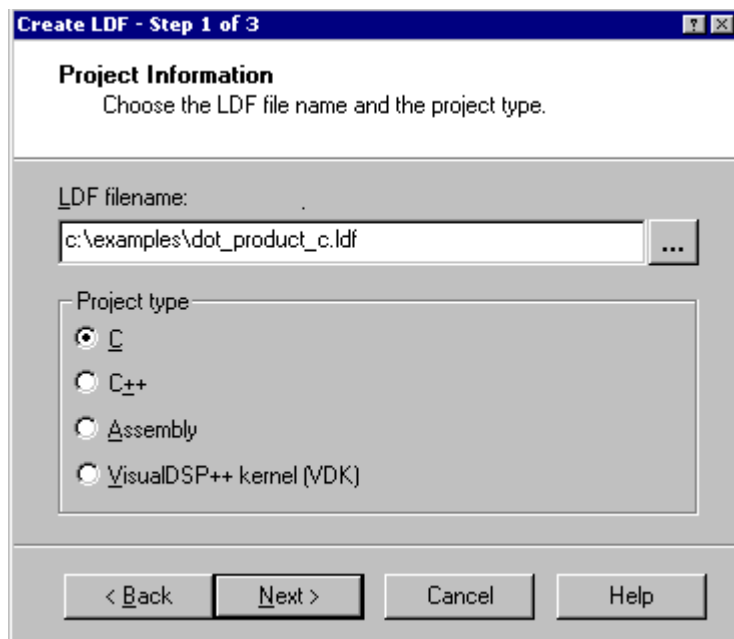


Figure 3-3. Selecting File Name and Project Type

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

Launching the Create LDF Wizard

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

Press **Next**.

Step 2: Specifying System Information

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

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

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

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

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

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

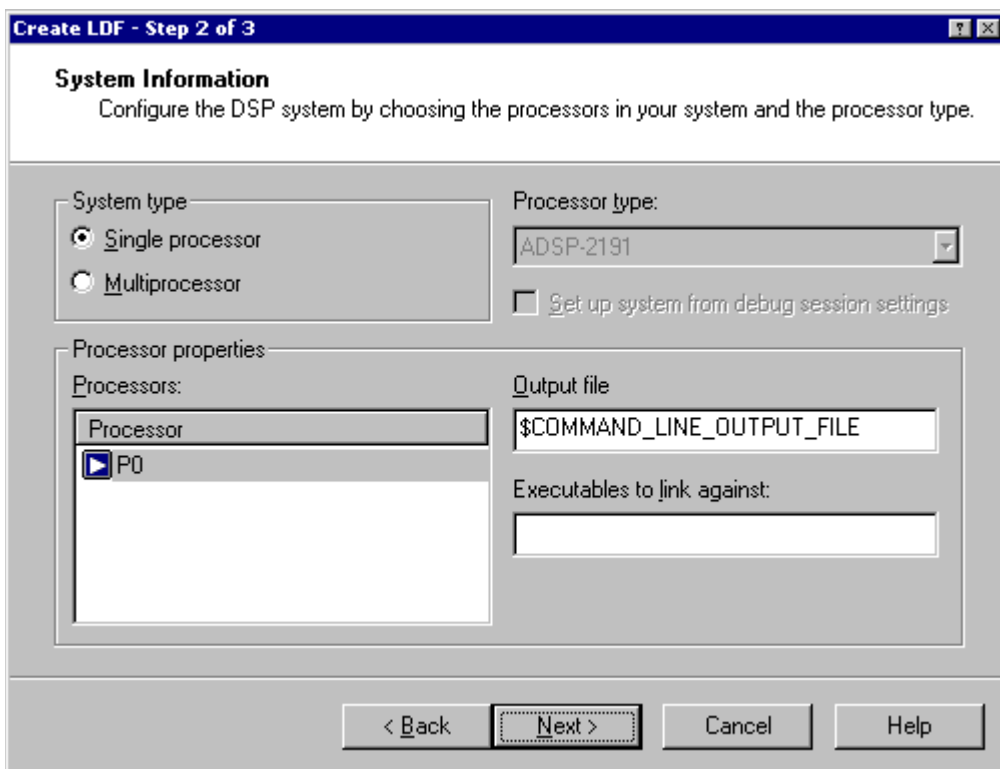


Figure 3-4. Selecting System and Processor Types

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

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

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

Step 3: Completing the LDF Wizard

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

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

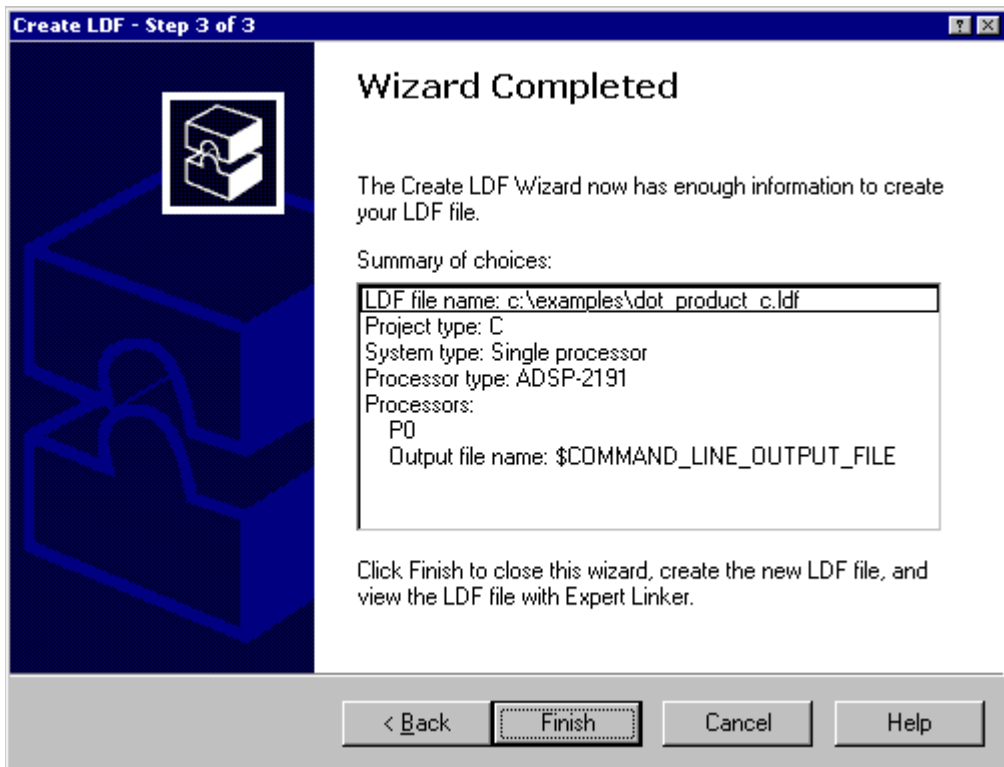


Figure 3-5. Wizard Completed Page of the Create LDF Wizard

Expert Linker Window Overview

The Expert Linker window contains two panes:

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

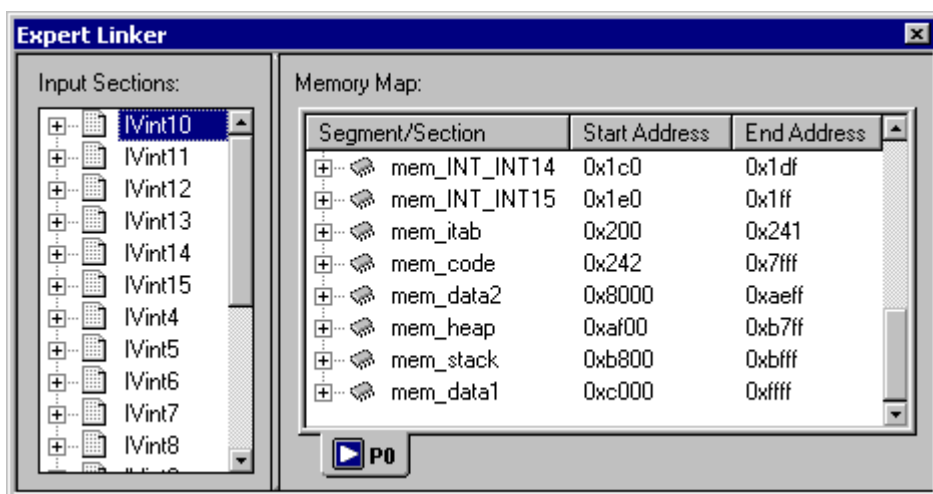


Figure 3-6. Expert Linker Window

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

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

Using the Input Sections Pane

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

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

Using the Input Sections Pane

The **Input Sections** pane ([Figure 3-6 on page 3-9](#)) initially displays a list of all the input sections, referenced by the .LDF file, and all input sections contained in the object files and libraries. Under each input section, there may be a list of LDF macros, libraries, and object files contained in that input section. You can add or delete input sections, LDF macros, or objects/library files in this pane.

Using the Input Sections Menu

Right-click an object in the **Input Sections** pane, a menu appears.

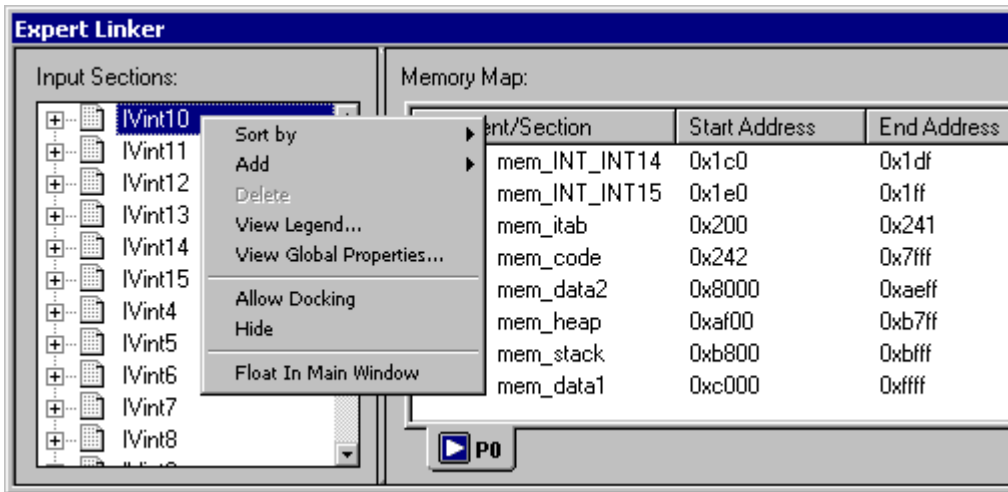


Figure 3-7. Input Sections Right-Click Menu

The main menu functions include:

- **Sort by** — Allows you to sort objects by input sections or LDF macros. These selections are mutually exclusive.
- **Add** — Allows you to add 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** — Allows you to delete the selected object (input section, object/library file, or LDF macro).
- **View Legend** — Displays the **Legend** dialog box showing icons and colors used by the Expert Linker.

Using the Input Sections Pane

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

Mapping an Input Section to an Output Section

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

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

If an input section is dragged onto a memory segment with no output section in it, an output section with a default name is automatically created and displayed.

A red “x” on an icon (for example, ) indicates that the object/file is not mapped. Once an input section has been completely mapped (all object files that contain the section are mapped), the icon next to the input section changes to indicate that it has been mapped; the 'x' disappears. See [Figure 3-8 on page 3-13](#).

While dragging the input section, the icon changes to a circle with a diagonal slash if it is over an object where you are not allowed to drop the input section.

Viewing Icons and Colors

Use the **Legend** dialog box to displays all possible icons in the tree pane and a short description of each icon ([Figure 3-8 on page 3-13](#)).

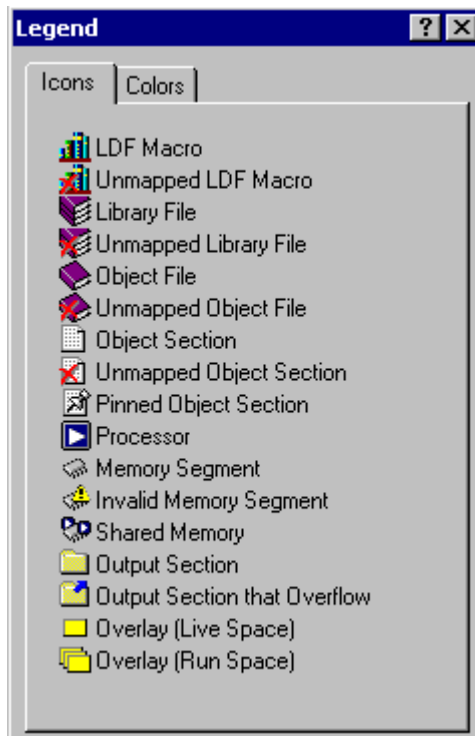


Figure 3-8. Legend Dialog Box – Icons Page



The red x on an icon indicates this object/file was not mapped yet.

Click the **Colors** tab to view the **Colors** page ([Figure 3-9 on page 3-14](#)). It contains a list of colors used in the graphical memory map view; each item's color can be customized. The list of displayed objects depends on the DSP family.

Using the Input Sections Pane

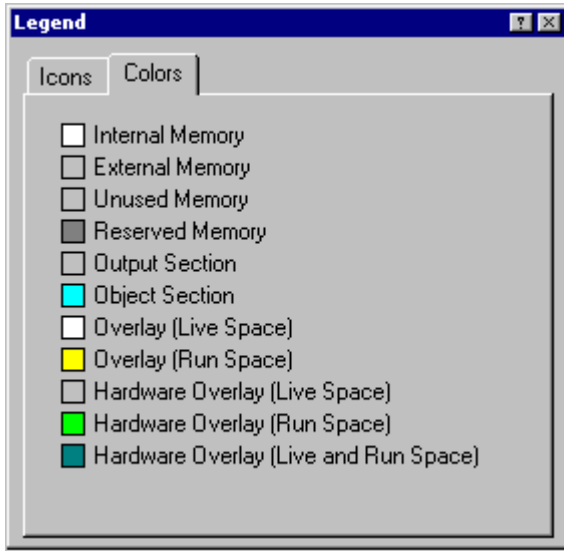


Figure 3-9. Legend Dialog Box – Colors Page

To change a color:

1. Double-click the color. You can also right-click on a color and select **Properties**.

The system displays the **Select a Color** dialog box ([Figure 3-10](#)).

2. Select a color and click **OK**.

Click **Other** to select other colors from the advanced palette.

Click **Reset** to reset all memory map colors to the default colors.

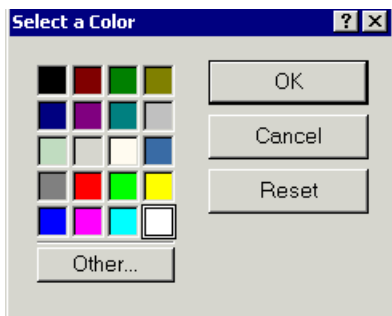


Figure 3-10. Selecting Colors

Sorting Objects

You can sort objects in the **Input Sections** pane by input sections (default) or by LDF macros, like `$OBJECTS` or `$COMMAND_LINE_OBJECTS`. The **Input Sections** and **LDF Macros** menu selections are mutually exclusive—only one can be selected at a time. For example,

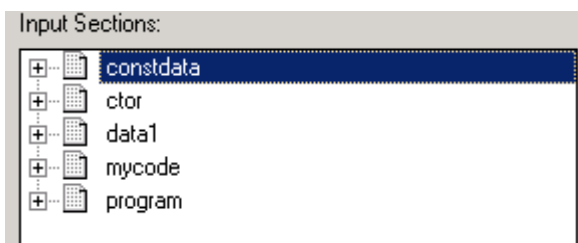


Figure 3-11. Expert Linker Window – Sorted by Input Sections

Other macros, object files, or libraries may appear under each macro. Under each object file are input sections contained in that object file.



When the tree is sorted by LDF macros, only input sections can be dragged onto output sections.

Using the Memory Map Pane

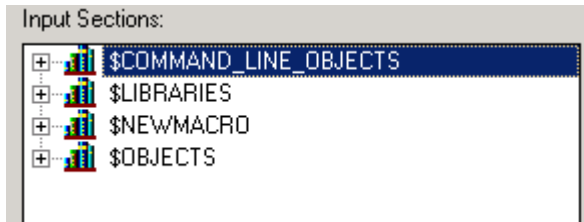


Figure 3-12. Expert Linker Window – Sorted by LDF Macros

Using the Memory Map Pane

In an .LDF file, the linker's `MEMORY()` command defines the target system's physical memory. Its argument list partitions memory into memory segments and assigns labels to each, specifying start and end addresses, memory width, and memory type (such as program, data, stack, and so on). It thereby connects your program to the target system. The `OUTPUT()` command directs the linker to produce an executable (.DXX) file and specifies its file name.

For ADSP-218x DSPs, a combo box permits the selection of PM, DM, or BM memory space.

The **Memory Map** pane has tabbed pages. You can page through the memory maps of the processors and shared memories to view their makeup. There are two viewing modes—a **tree** view and a **graphical** view.

Select these views and other memory map features by means of the right-click (context) menu. All procedures involving memory map handling assume that the Expert Linker window is open.

The **Memory Map** pane displays a tooltip when you move the mouse cursor over an object in the display. The tooltip shows the object's name, address, and size. The system also uses representations of overlays, which display in “run” space and “live” space.

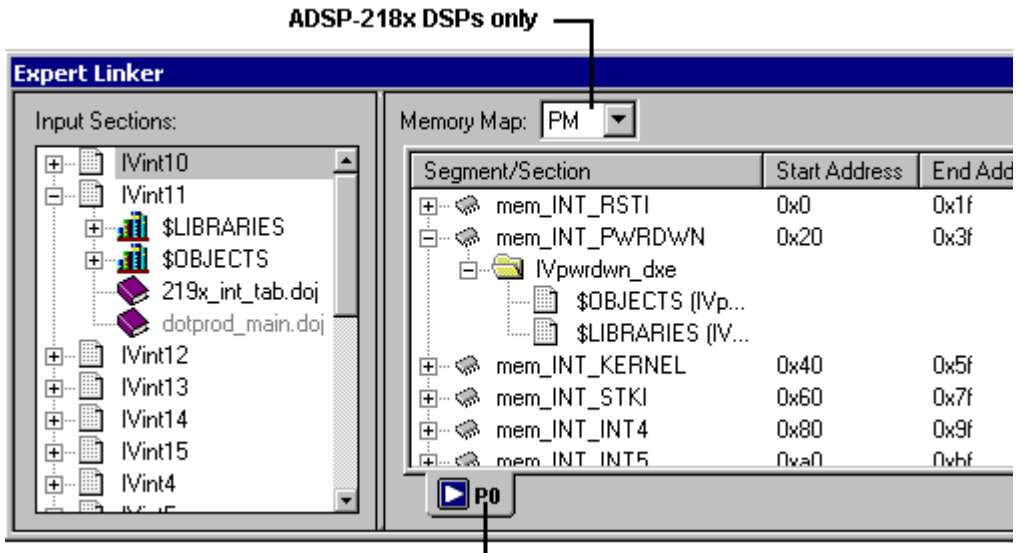


Figure 3-13. Expert Linker Window – Memory Map

Invalid Memory Segment Notification:

When a memory segment is invalid (for example, when a memory range overlaps another memory segment), the memory width is invalid. The tree shows an **Invalid Memory Segment** icon (see also [Figure 3-8 on page 3-13](#)). Move the mouse pointer over the icon, and a tooltip displays a message describing why the segment is invalid.

Using the Context Menu

Display the context menu by right-clicking in the **Memory Map** pane. The menu allows you to select and perform major functions. The available right-click menu commands are listed below.

Using the Memory Map Pane

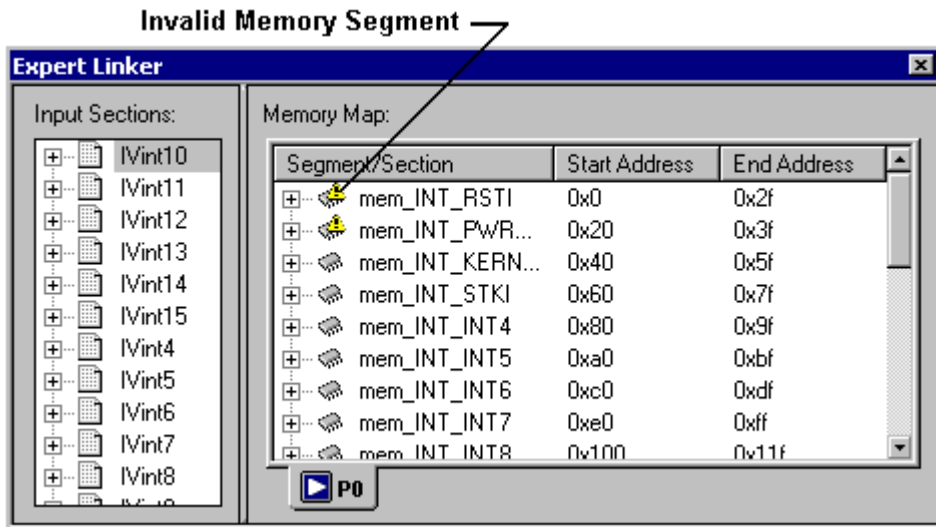


Figure 3-14. Memory Map With Invalid Memory Segments

View Mode

- **Memory Map Tree** — Displays the memory map in a tree representation (see [Figure 3-15 on page 3-21](#)).
- **Graphical Memory Map** — Displays the memory map in graphical blocks (see [Figure 3-16 on page 3-23](#)).

View

- **Mapping Strategy (Pre-Link)** — Displays the memory map that shows where you plan to place your object sections.
- **Link Results (Post-Link)** — Displays the memory map that shows where the object sections were actually placed.

New

- **Memory Segment** — Allows you to specify the name, address range, type, size, and so on for memory segment 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** (output section to overflow or **None**), **Packing**, and **Number of bytes** (number of bytes to be re-ordered at one time. This value does not include the number of null bytes inserted into memory).

- **Shared Memory** — Adds a shared memory to the memory map.
- **Overlay** — Invokes a dialog box that allows you to add a new overlay to the selected output section or memory segment. The selected output section is the new overlay's run space (see [Figure 3-43 on page 3-58](#)).

Delete — Deletes the selected object.

Pin to Output Section — Appears only when you right-click on an object section that is part of an output section specified to overflow to another output section. Pinning an object section to an output section prevents it from overflowing to another output section.

View Section Contents — Available only after linking or building the project and then right-clicking on an input or object section. Invokes a dialog box that displays the contents of the input or output section (see [Figure 3-27 on page 3-36](#)).

Add Hardware Page Overlay Support — (ADSP-218x only) Automatically sets up hardware overlay live and run spaces for available hardware pages. Expert Linker checks whether any memory segments are currently defined in all of the hardware pages. If there are, you are queried before

Using the Memory Map Pane

the segments are deleted. Expert Linker then creates a memory segment containing an overlay (live space) in each hardware page and creates a memory segment containing all of the overlay run spaces in hardware page 0. Lastly, Expert Linker creates a default mapping for each overlay to map objects containing the “pmpage0” section to the hardware overlay on PM page 0, the “pmpage1” section to PM page 1, the “dmpage0” section to DM page 0, and so on.

View Symbols — (Available only after linking the project and then right-clicking on a processor, overlay, or input section) Invokes a dialog box that displays the symbols for the project, overlay, or input section (see [Figure 3-30 on page 3-39](#)).

Properties — Displays a **Properties** dialog box for the selected object. The **Properties** menu is context-sensitive; different properties display for different objects. Right-click a memory segment and choose **Properties** to specify a memory segment's attributes (name, start address, end address, size, width, memory space, PM/DM/(BM), RAM/ROM, and internal/external flag).

View Legend — Displays the **Legend** dialog box displaying tree view icons and a short description of each icon. The **Colors** page lists the colors used in the graphical memory map. You can customize each object's color. See [Figure 3-8 on page 3-13](#) and [Figure 3-9 on page 3-14](#).

View Global Properties — Displays a **Global Properties** dialog box that lists the map file generated after linking the project. It also provides access to some processor and setup information (see [Figure 3-31 on page 3-41](#)).

Tree View Memory Map Representation

In the tree view (right-click and choosing **View Mode -> Memory Map Tree**), the memory map displays with memory segments at the top-level.

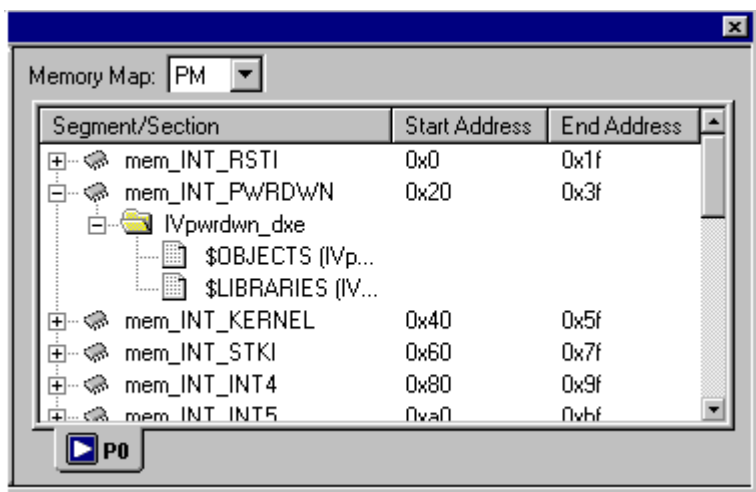


Figure 3-15. Expert Linker Window – Memory Map

Each memory segment may have one or more output sections under it. Input sections mapped to an output section display under that output section.

The start address and size of the memory segments display in separate columns. If available, the start address and the size of each output section display (for example, after linking the project).

Graphical View Memory Map Representation

In the graphical view (selected by right-clicking and choosing **View Mode** -> **Graphical Memory Map**), the graphical memory map displays the processor's hardware memory map (refer to your DSP's *Hardware Reference* manual or data sheet). Each hardware memory segment contains a list of user-defined memory segments.

View the memory map from two perspectives—pre-link view and post-link view (see [“Specifying Pre- and Post-Link Memory Map View” on page 3-28](#)).

[Figure 3-16 on page 3-23](#) shows an example of a graphical memory map.

In graphical view, the memory map comprises blocks of different colors, representing memory segments, output sections, objects, and so on. The memory map is drawn with the following rules:

- An output section is represented as a vertical header with a group of object to the right of it.
- A memory segment's border and text change to red (from its normal black color) to indicate that it is invalid. When you move the mouse pointer over the invalid memory segment, a tooltip displays a message, describing why the segment is invalid.
- The height of the memory segments is not scaled as a percentage of the total memory space. However, the width of the memory segments is scaled as a percentage of the widest memory.
- Object sections are drawn as horizontal blocks stacked on top of each other. Before linking, the object section sizes are not known and display in equal sizes within the memory segment. After link-

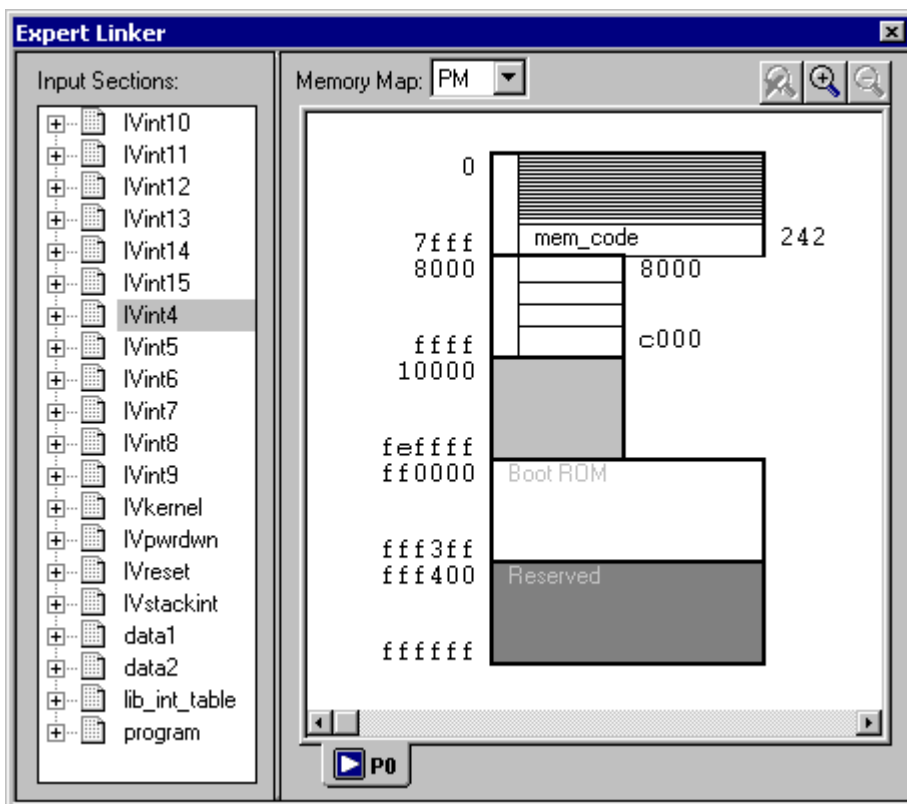


Figure 3-16. Graphical Memory Map Representation

ing, the height of the objects is scaled as a percentage of the total memory segment size. Object section names display only when there is enough room for display.

- Addresses are ordered in ascending order from top to bottom.

Using the Memory Map Pane

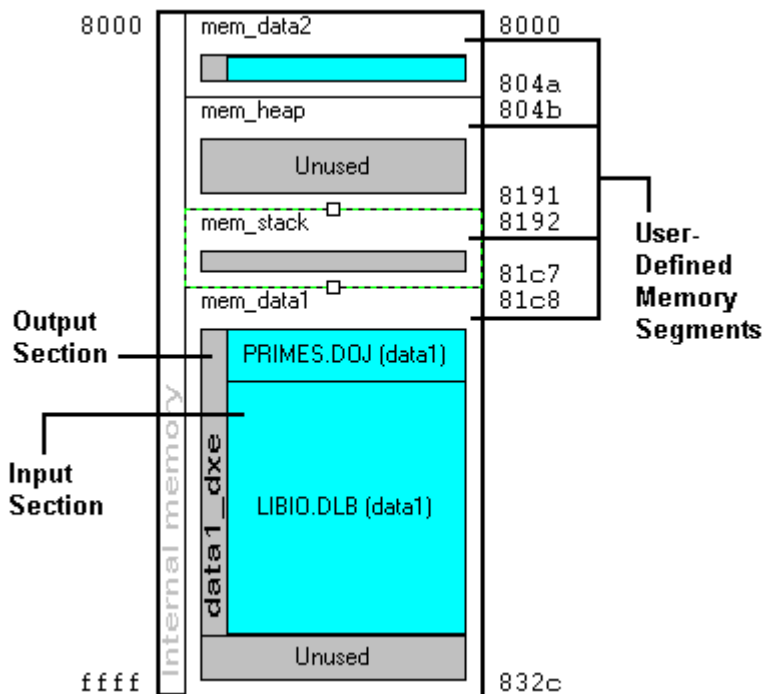


Figure 3-17. Viewing Sections and Segments in Memory Map

Three buttons at the top right of the Memory Map pane permit zooming. If there is not enough room to display the memory map when zoomed in, horizontal and/or vertical scroll bars allow you to view the entire memory map (for more information, see [“Zooming In and Out on the Memory Map”](#) on page 3-28).

You can drag and drop any object except memory segments.

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

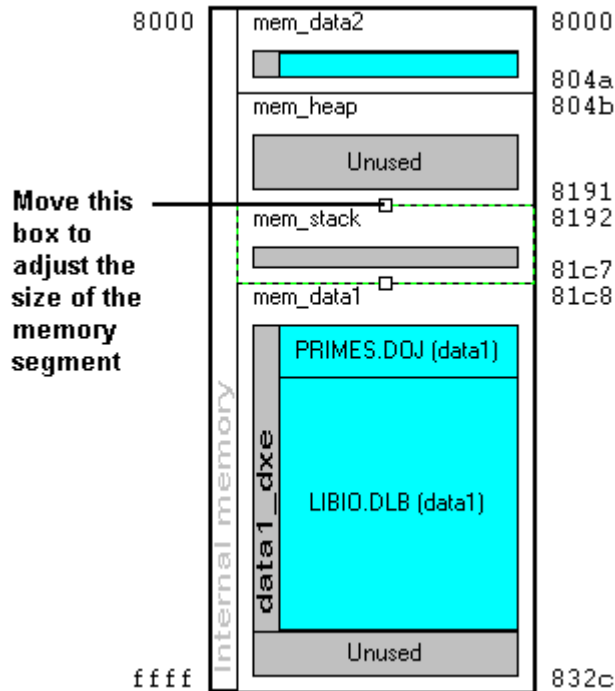


Figure 3-18. Adjusting the Size of a Memory Segment

When the mouse pointer is on top of the box, the resize cursor appears as



When an object is selected in the memory map, it highlights as shown in [Figure 3-20 on page 3-27](#). If you move the mouse pointer over an object in the graphical memory map, a yellow tooltip appears displaying the information about the object (such as name, address, and size).

Using the Memory Map Pane

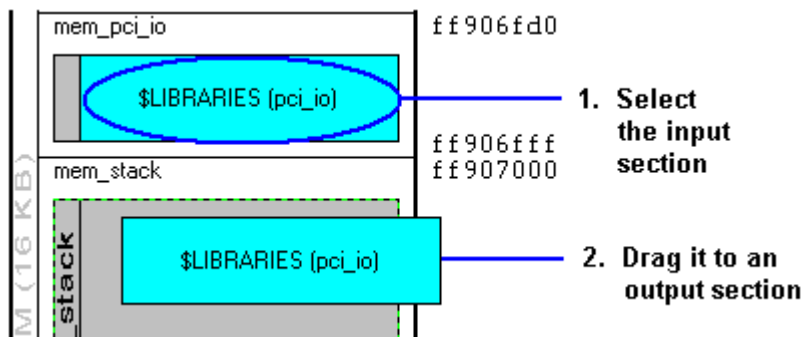


Figure 3-19. Dragging and Dropping an Object

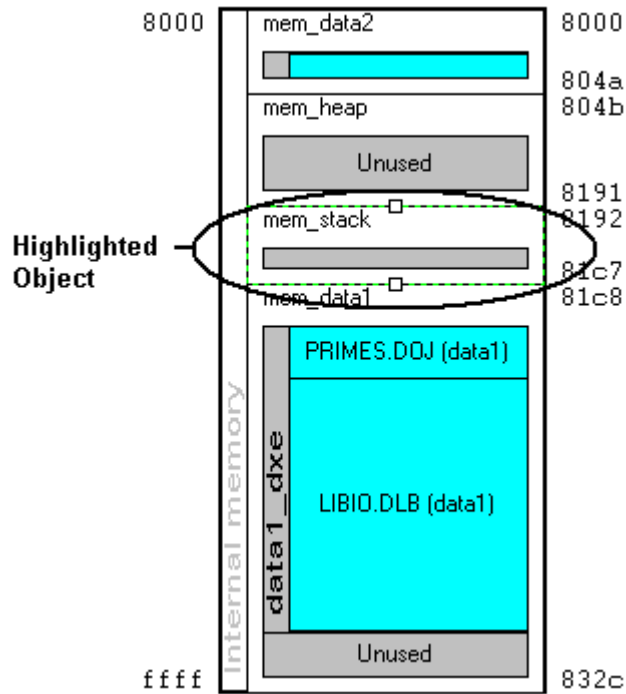


Figure 3-20. A Highlighted Memory Segment in the Memory Map

Specifying Pre- and Post-Link Memory Map View

View the memory map from two perspectives: pre-link view and post-link view. Pre-link view is typically used to place input sections. Post-link view is typically used to view where the input sections were placed after linking your project. Other information is available after linking (such as the sizes of each section, symbols, and the contents of each section).

- To enable pre-link view from the **Memory Map** pane, right-click and choose **View and Mapping Strategy (Pre-Link)**. [Figure 3-21 on page 3-29](#) illustrates memory map before linking.
- To enable post-link view from the **Memory Map** pane, right-click and choose **View and Link Results (Post-Link)**. [Figure 3-22 on page 3-30](#) illustrates memory map after linking.

Zooming In and Out on the Memory Map

From the **Memory Map** pane, you can zoom in or out incrementally or zoom in or out completely. Three buttons at the top right of the pane perform zooming operations. Horizontal and/or vertical scroll bars appear when there is not enough room to display a zoomed memory map in the **Memory Map** pane.

To:

- Zoom in, click on the magnifying glass icon with the + sign above the upper right corner of the memory map window.
- Zoom out, click on the magnifying glass icon with the - sign above the upper right corner of the memory map window.
- Exit zoom, click on the magnifying glass icon with the 'x' above the upper right corner of the memory map window.

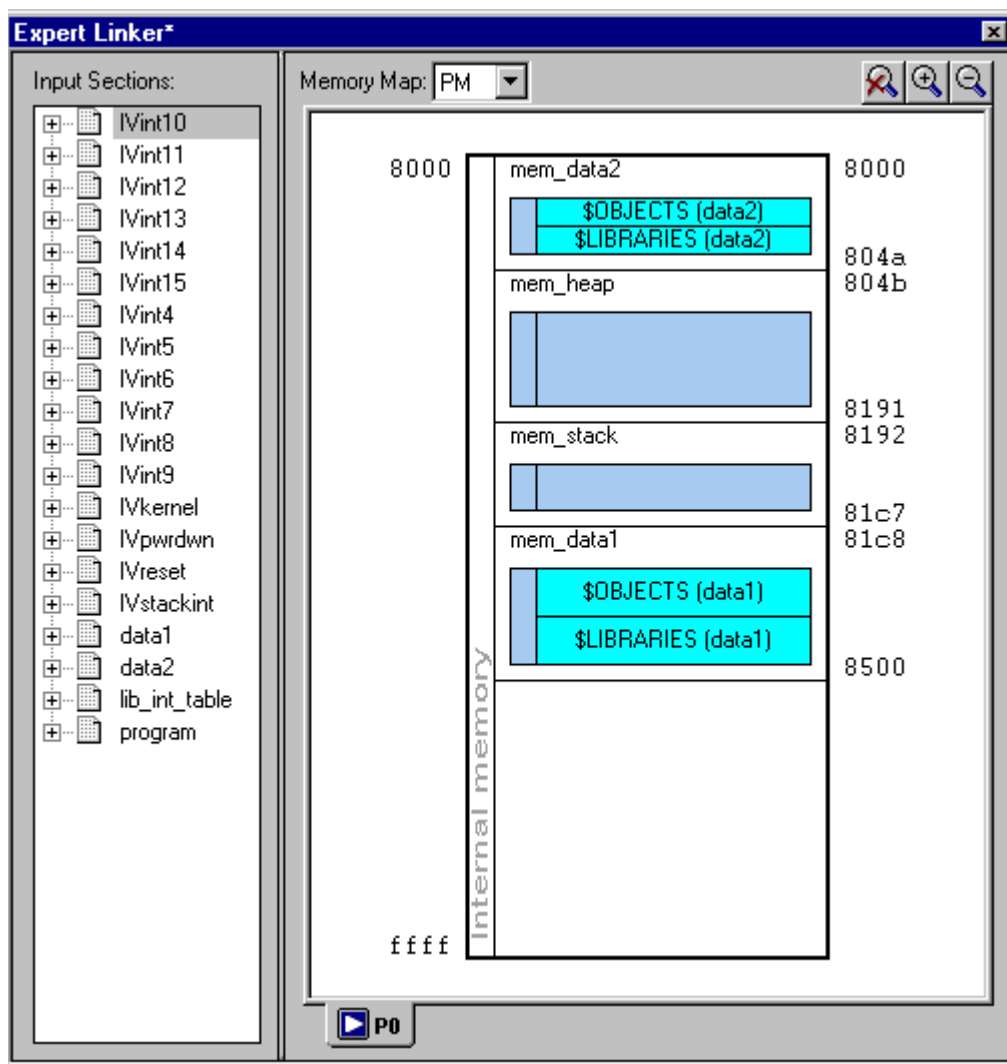


Figure 3-21. Memory Map Pane in Pre-Link View

Using the Memory Map Pane

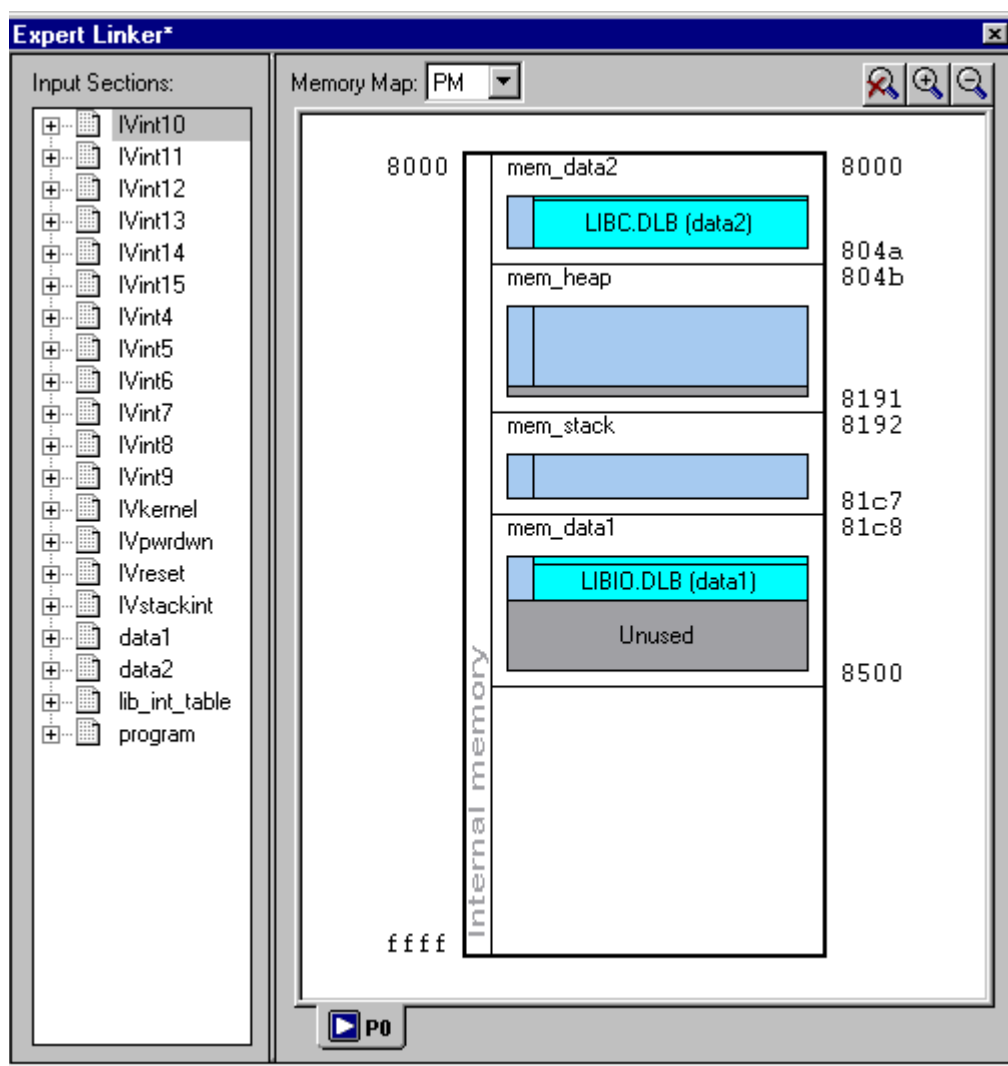


Figure 3-22. Memory Map Pane in Post-Link View



Figure 3-23. Memory Map – Zoom Options

- View a memory object by itself, double-click on the memory object.
- View the memory object containing the current memory object, double-click on the white space around the memory object

Inserting a Gap into a Memory Segment

A gap may be inserted into a memory segment in the graphical memory map.

To insert a gap:

1. Right-click on a memory segment.
2. Choose **Insert gap**. The **Insert Gap** dialog box appears.

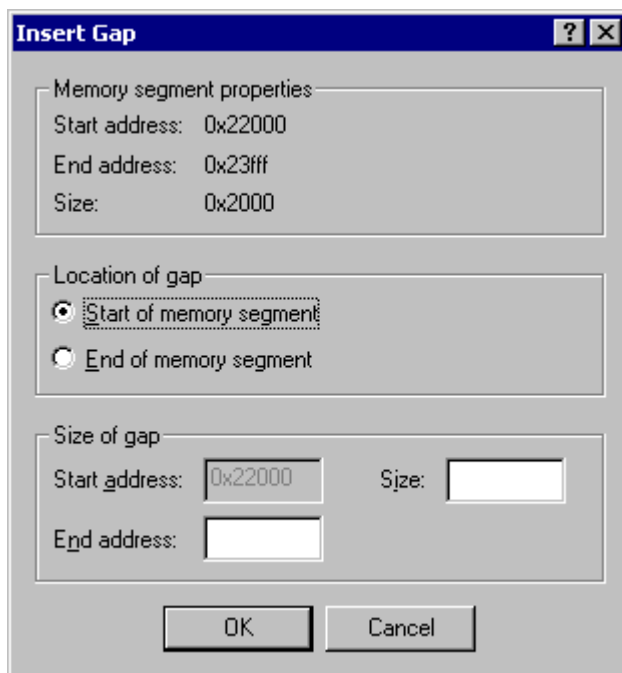


Figure 3-24. Insert Gap Dialog Box

You may insert a gap at the start of the memory segment or the end of it. If the start is chosen, the **Start address** for the gap is grayed out and you must enter an end address or size (of the gap). If the end is chosen, the **End address** of the gap is grayed out, and you must enter a start address or size.

Working with Overlays

Overlays appear in the memory map window in two places: “run” space and “live” space. Live space is where the overlay is stored until it is swapped into run space. Because multiple overlays can exist in the same “run” space, the overlays display as multiple blocks on top of each other in cascading fashion.

[Figure 3-25](#) shows an overlay in live space, and [Figure 3-26](#) shows an overlay in run space.

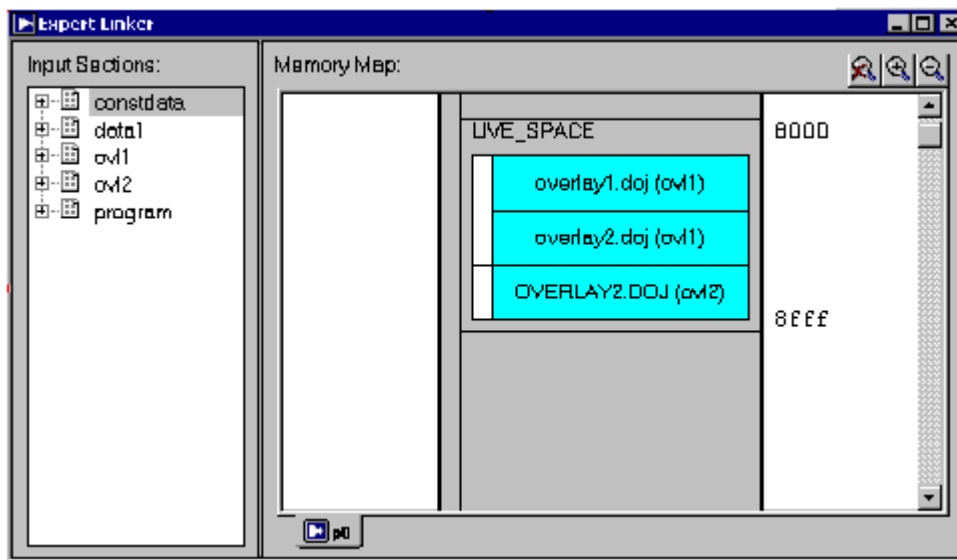


Figure 3-25. Graphical Memory Map Showing Overlay Live Space

Using the Memory Map Pane

Overlays in a “run” space display one at a time in the graphical memory map. The scroll bar next to an overlay in “run” space allows you to specify an overlay to be shown on top. You can drag the overlay on top to another output section to change the “run” space for an overlay.

Click the up arrow or down arrow button in the header to display previous or next overlay in “run” space. Click the browse button to display the list of all available overlays. The header shows the number of overlays in this run space and the current overlay number.

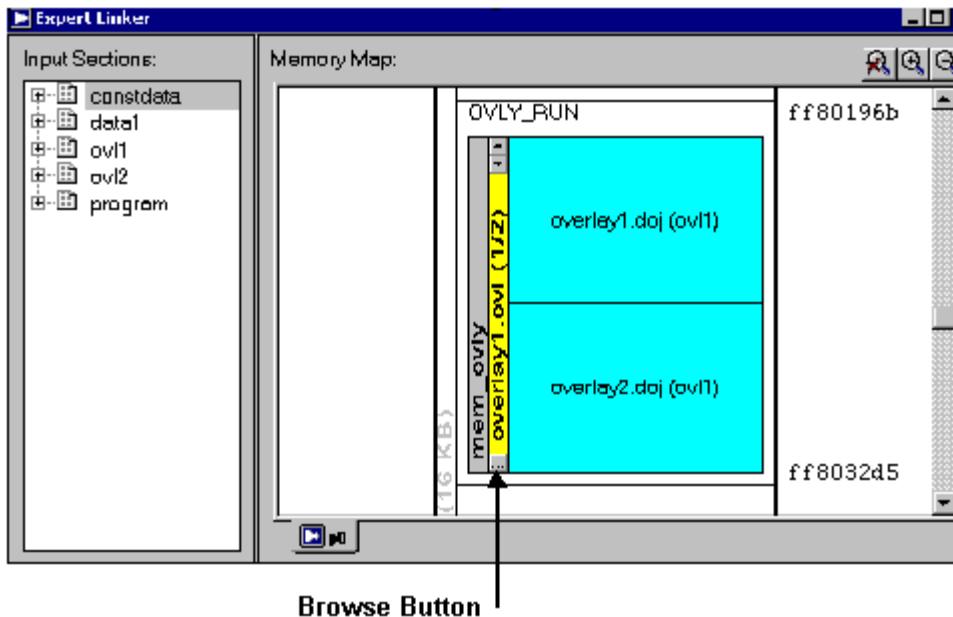


Figure 3-26. Graphical Memory Map Showing Overlay Run Space

To create an overlay in the “run” space:

1. Right-click on an output section.
2. Choose New -> Overlay.

3. Select the “live” space from the **Overlay Properties** dialog box. The new overlay displays in the “run” and “live” spaces in two different colors in the memory map.
4. Drag the “live” space overlay to a different output section. This can change the “live” to “run” space.

Viewing Section Contents

You can view the contents of an input section or an output section. You must specify the particular memory address and the display’s format.

This capability employs the `elfdump.exe` utility to obtain the section contents and display it in a window similar to a memory window in VisualDSP++. Multiple **Section Contents** dialog boxes may be displayed.

For example, 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 **Output Section Contents** dialog box appears.

By default, the memory section content displays in **Hex** format.
3. Right-click anywhere in the section view to display a menu with these selections:
 - **Go To** — Allows you to display 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 3-28 on page 3-37](#) and [Figure 3-29 on page 3-38](#) illustrate other memory data formats available for the selected output section.

Using the Memory Map Pane

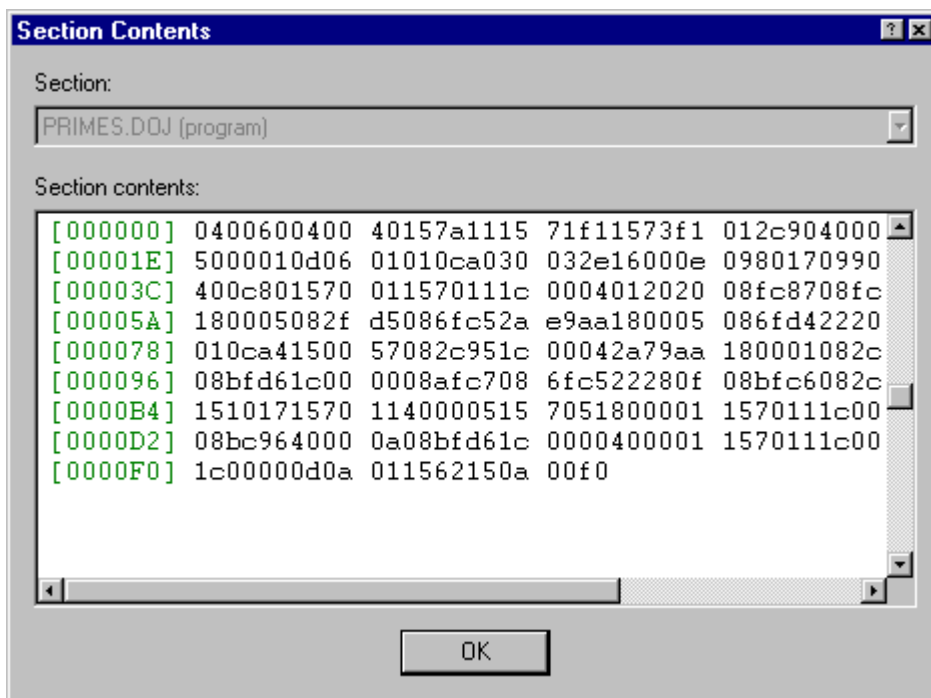


Figure 3-27. Output Section Contents in Hex Format

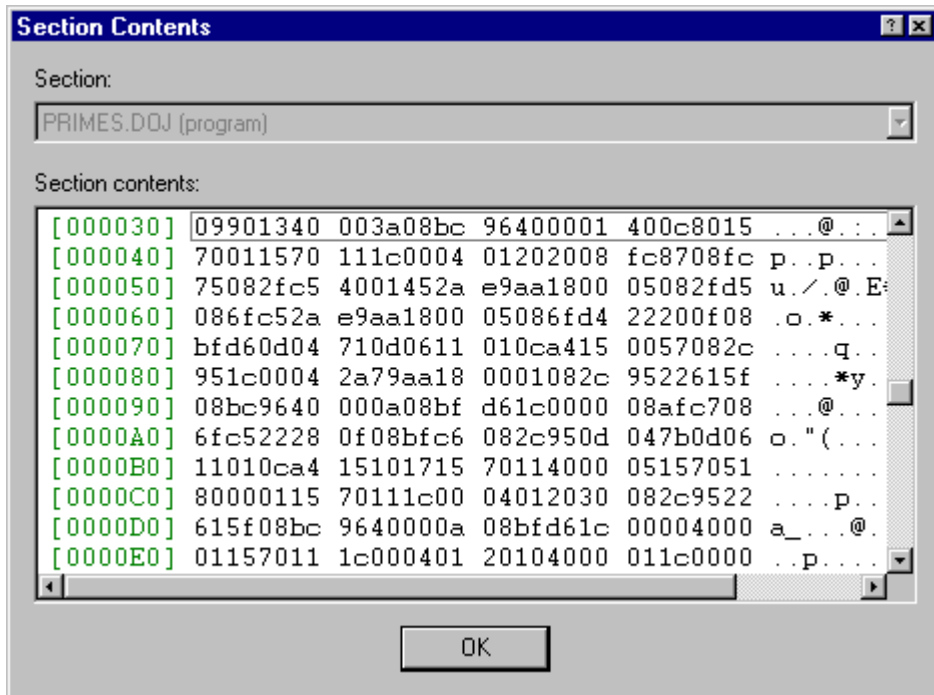


Figure 3-28. Output Section Contents in Hex and ASCII Format

Viewing Symbols

Symbols can be displayed per processor program (.DXE), per overlay (.OVL), or per input section. Initially, symbol data is in the same order that it appears in the linker's map output. Sort symbols by name, address, and so on by clicking the column headings.

Using the Memory Map Pane

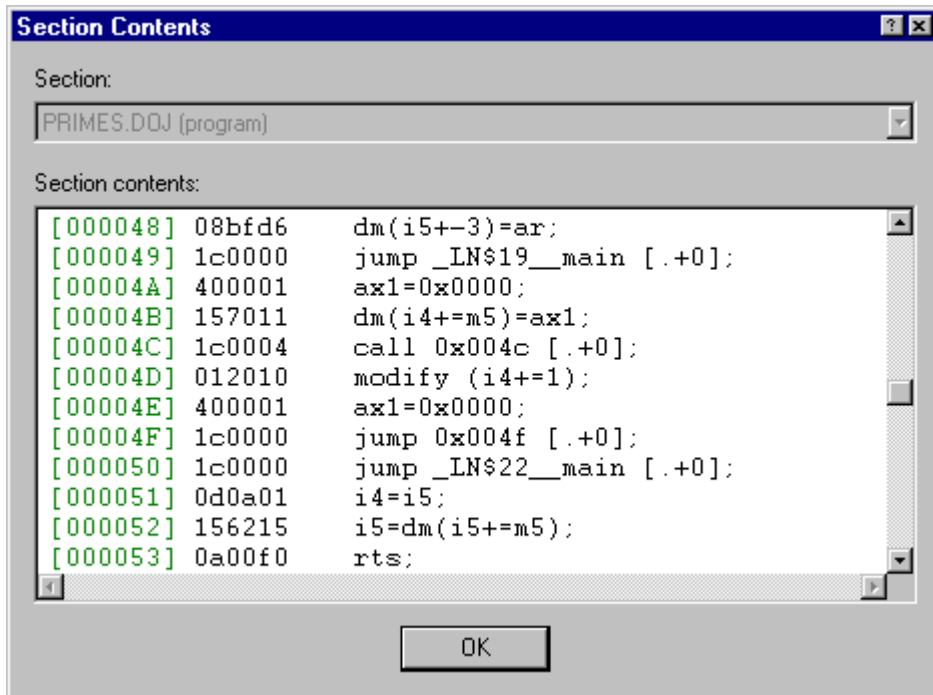


Figure 3-29. Output Section Contents in Hex and Assembly Format

To view symbols:

1. In the post-link view of the **Memory Map** pane, select the item (memory segment, output section, or input section) whose symbols you want to view.
2. Right-click and choose **View Symbols**.

The **View Symbols** dialog box displays the selected item's symbols. The symbol's address, size, binding, file name, and section appear beside the symbol's name.

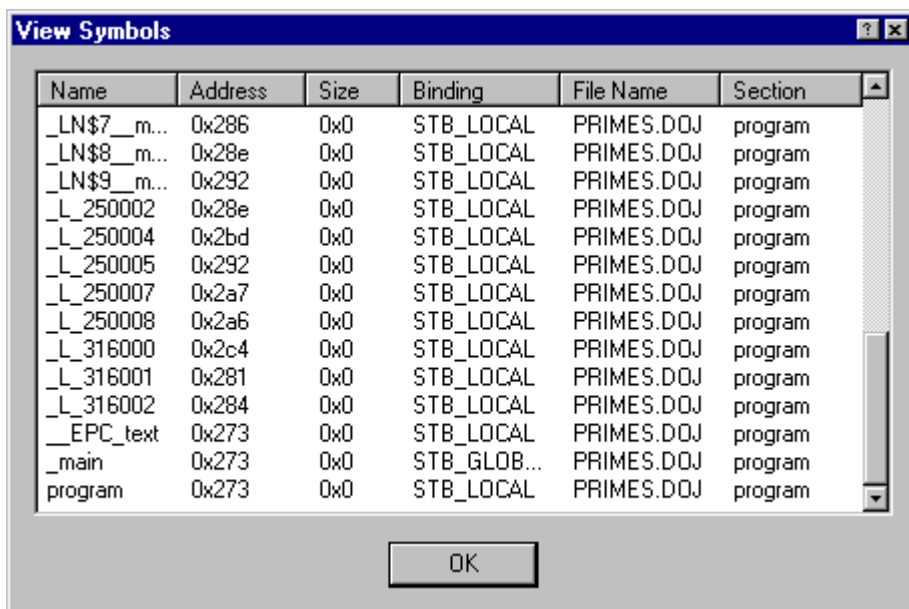


Figure 3-30. View Symbols Dialog Box

Managing Object Properties

You can display different properties for each type of object. Since different objects may share certain properties, their **Properties** dialog boxes share pages. The following procedures assume the Expert Linker window is open.

To display a **Properties** dialog box, right-click an object and choose **Properties**. You may choose these functions:

- [“Managing Global Properties” on page 3-41](#)
- [“Managing Processor Properties” on page 3-42](#)
- [“Managing PLIT Properties for Overlays” on page 3-44](#)
- [“Managing Elimination Properties” on page 3-45](#)
- [“Managing Symbols Properties” on page 3-47](#)
- [“Managing Memory Segment Properties” on page 3-51](#)
- [“Managing Output Section Properties” on page 3-53](#)
- [“Managing Packing Properties” on page 3-55](#)
- [“Managing Alignment and Fill Properties” on page 3-56](#)
- [“Managing Overlay Properties” on page 3-58](#)
- [“Managing Stack and Heap in DSP Memory” on page 3-60](#)

Managing Global Properties

Use the context menu to display Global Properties:

1. Right-click in a section pane of the Expert Linker window.
2. From the context menu, choose **Global Properties**.
The **Global Properties** dialog box appears.

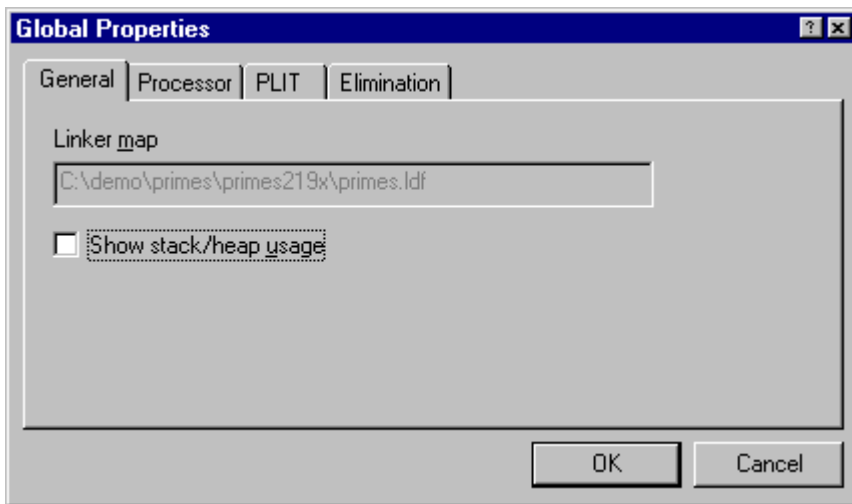


Figure 3-31. General Page of the Global Properties Dialog Box

The **Global Properties** dialog box provides the following selections:

- **Linker map file** displays the map file generated after linking the project. This is a read-only field.
- If **Show stack/heap usage** is selected, after you run a project, Expert Linker shows how much of the stack and heap were used.

Managing Processor Properties

To specify Processor properties:

1. In the **Memory Map** pane, right-click on a **Processor** tab and choose **Properties**.

The **Processor Properties** dialog box appears.

2. Click the **Properties** tab.

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

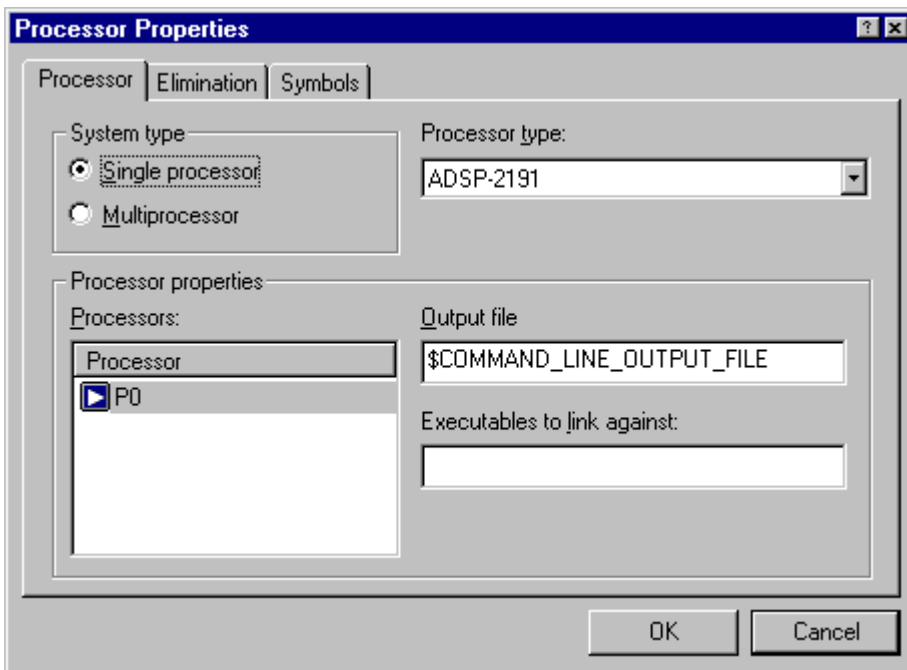


Figure 3-32. Processor Page of the Processor Properties Dialog Box

With a **Processor** tab in focus, you can:

- Specify **System Type** — Use the **Single processor** selection.
- Select a **Processor type** (such as ADSP-2191).
- Specify an **Output file** name — The file name may include a relative path and/or LDF macro.
- Specify **Executables to link against** — Multiple files names are permitted, but must be separated with space characters. Only .SM, .DLB, and .DXE files are permitted. A file name may include a relative path and/or LDF macro.

Additionally, you can rename a processor by selecting the processor, right-clicking, and choosing **Rename Processor**, and typing a new name.

Managing PLIT Properties for Overlays

The **PLIT** tab allows you to view and edit the function template used in overlays. Assembly instructions observe the same syntax-coloring as specified via for editor windows.

 You can enter assembly code only. Comments are not allowed.

To view and edit **PLIT** information:

1. Right-click in the **Input Sections** pane.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. Click the **PLIT** tab.

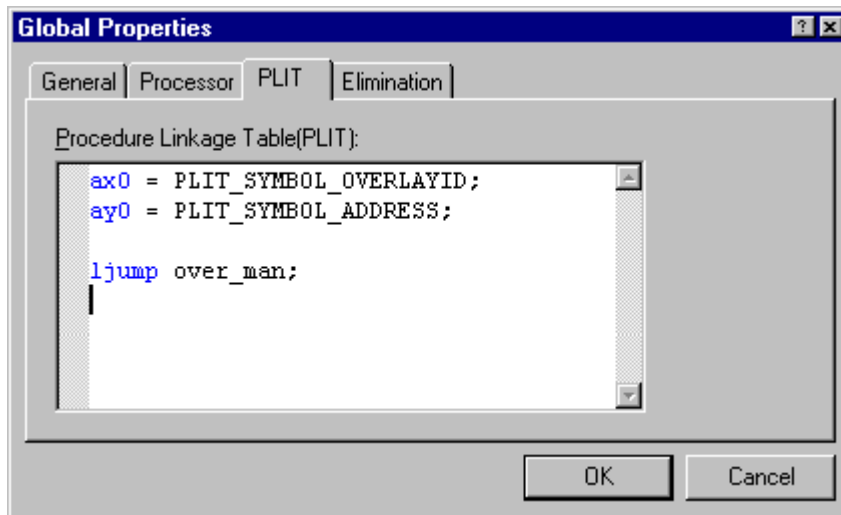


Figure 3-33. PLIT Page of the Global Properties Dialog Box

Managing Elimination Properties

You can eliminate unused code from the target .DxE file. Specify the input sections from which to eliminate code and the symbols you want to keep.

The **Elimination** tab allows you to perform elimination.

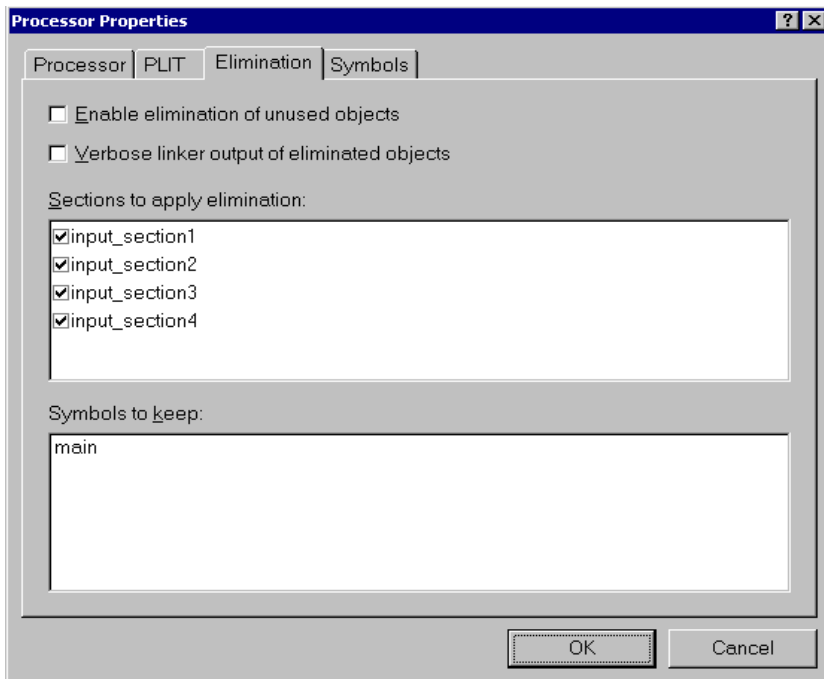


Figure 3-34. Processor Properties Dialog Box – Elimination Tab

Selecting the **Enable elimination of unused objects** option allows you to enable elimination. This check box is grayed out when elimination is enabled through the linker command line or when the .LDF file is read-only.

Managing Object Properties

When **Verbose linker output of eliminated objects** is selected, the eliminated objects will be shown as linker output in the **Output** window's **Build** tab during linking. This check box is grayed out when the **Enable elimination of unused objects** check box is cleared. It is also grayed out when elimination is enabled through the linker command line or when the .LDF file is read-only.

The **Sections to apply elimination** box lists all input sections with a check box next to each section. Elimination applies to the sections that are selected. By default, all input sections are selected.

Symbols to keep is a list of symbols you want to keep. The linker will not remove these symbols. If you right-click in this list box, a menu allows you to:

- Add a symbol by typing in the new symbol name in the edit box at the end of the list
- Remove the selected symbol

Managing Symbols Properties

You can view the list of symbols that the linker is to resolve. 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 (.DXX). 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 EL window.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. To add or remove a symbol, click the **Elimination** tab.
4. Right-click in the **Symbols to keep** window.

Choose **Add Symbol**. In the ensuing dialog box, type new symbol names at the end of the existing list. To delete a symbol, select the symbol, right-click, and choose **Remove Symbol**.

Specifying Symbol Resolution

1. In the **Memory Map** pane, right-click a processor tab.
2. Choose **Properties**. The **Processor** page of the **Processor Properties** dialog box appears. The **Symbols** tab allows you to specify how symbols are to be resolved by the linker.

The symbols can be resolved to an absolute address or to a program file (.DXX). When you right-click in the **Symbols** field, a menu enables you to add or remove symbols.

Managing Object Properties

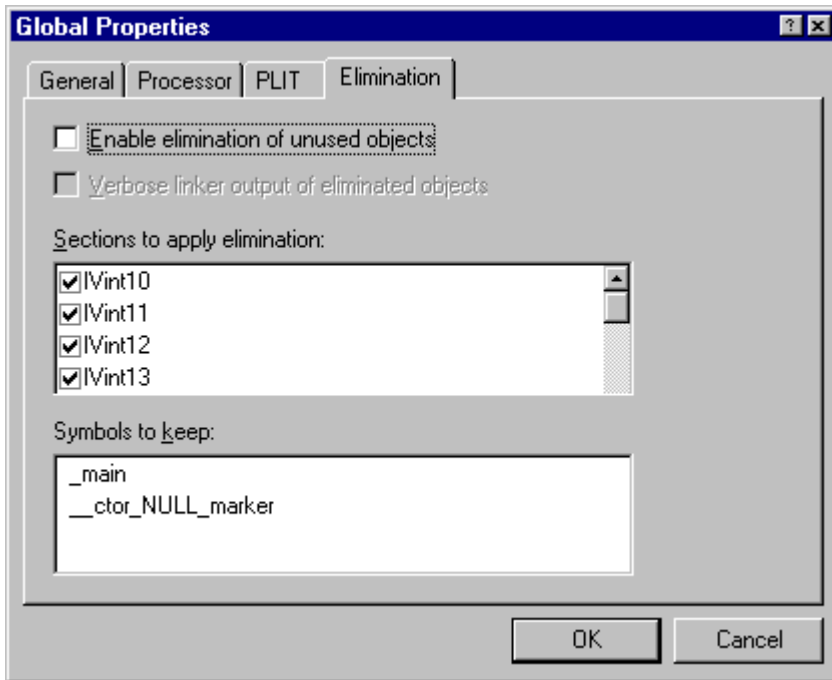


Figure 3-35. Elimination Page of the Global Properties Dialog Box

Choosing **Add Symbol** from the menu invokes the **Add Symbol to Resolve** dialog box allowing you to pick a symbol by either typing the name or browsing for a symbol. Using **Resolve with**, you can also decide whether to resolve the symbol from a known absolute address or file name (.DXE or .SM file).

The **Browse** button is grayed out when no symbol list is available; for example, if the project has not been linked. When this button is active, click it to display the **Browse Symbols** dialog box, which displays a list of all the symbols (see [Figure 3-38 on page 3-50](#)).

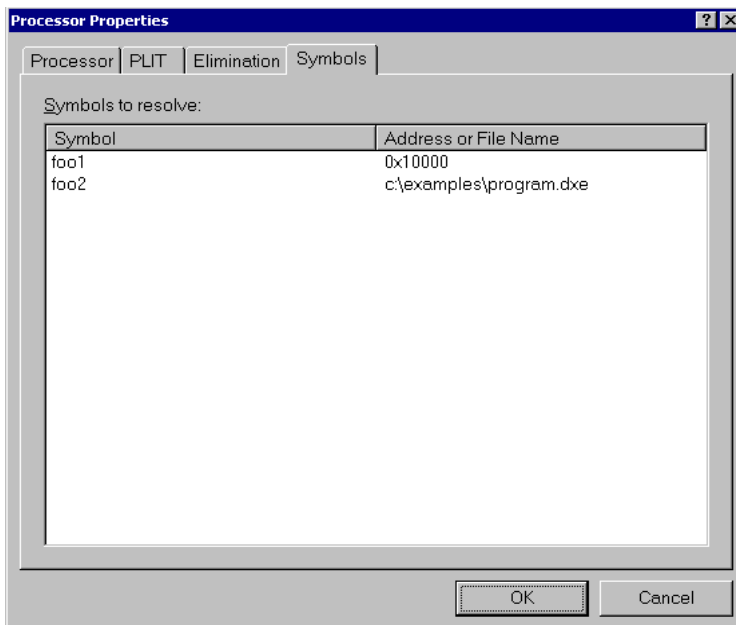


Figure 3-36. Processor Properties Dialog Box – Symbols Tab

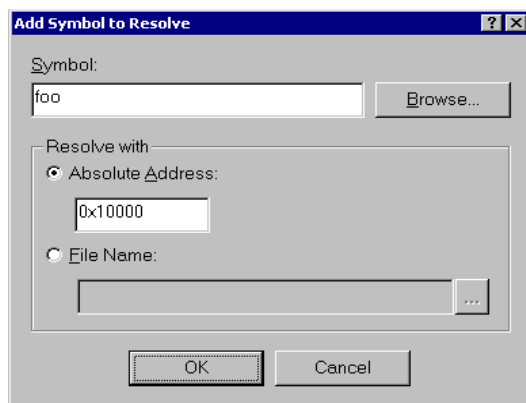


Figure 3-37. Add Symbol to Resolve Dialog Box

Managing Object Properties

You can select a symbol from that list and it will appear in the **Symbol** box of the **Edit Symbol to Resolve** dialog box.

Deleting a Symbol from the Resolve List

Click **Browse** to display the **Symbols to resolve** list in the **Symbols** pane (Figure 3-36 on page 3-49). Select a symbol you want to delete.

Right-click and choose **Remove Symbol**.

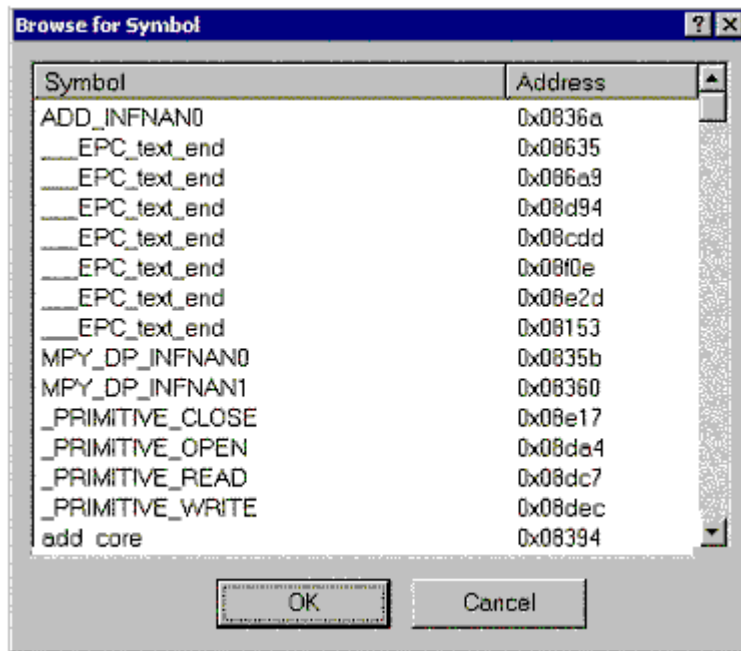


Figure 3-38. Browse for Symbol Dialog Box

Managing Memory Segment Properties

You can specify/change the memory segment's name, start address, end address, size, width, memory space, PM/DM/BM, RAM/ROM, and internal/external flag.



The BM memory space option is available only for ADSP-218x DSPs.

To display the Memory Segment Properties dialog box:

1. Right-click a memory segment (for example, `PROGRAM`) in the **Memory Map** pane.
2. Choose **Properties**. The selected segment properties are displayed.

Managing Object Properties

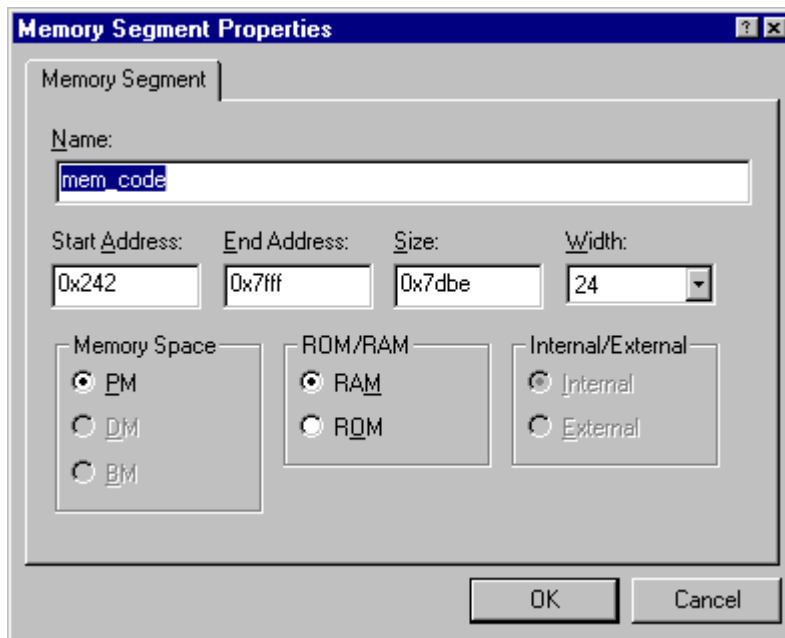


Figure 3-39. Memory Segment Properties Dialog Box

Managing Output Section Properties

The **Output Section** tab allows you to change the output section's name or to set the overflow. Overflow allows objects that do not fit in the current output section to spill over into the specified output section. By default, all objects that do not fit (except objects that are manually pinned to the current output section) overflow to the specified section.

To specify output section properties:

1. Right-click an output section in the **Memory Map** pane.
2. Choose **Properties**.



Figure 3-40. Output Section Properties Dialog Box – Output Section Tab

Managing Object Properties

The selections in the output section/segment list includes “None” (for no overflow) and all output sections. Objects can be pinned to an output section. To do that, right-click the object and then choose **Pin to output section**.

You can:

- Type a name for the output section in **Name**.
- Select an output section into which the selected output section will overflow in **Overflow**. Or select **None** for no overflow. This setting appears in the **Placement** box.

Before linking the project, the **Placement** box indicates the output section’s address and size as “Not available”. After linking, the box displays the output section’s actual address and size.

You should also specify the **Packing** and **Alignment** (with **Fill Value**) properties as needed.

Managing Packing Properties

The **Packing** tab allows you to specify the packing format that the linker uses to place bytes into memory. The choices include **No packing** or **Custom** packing. You can view byte order, which defines the order that bytes will be placed into memory. With Blackfin DSPs, **No packing** is the only packing method available.

To specify packing properties:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties** and click the **Packing** tab.

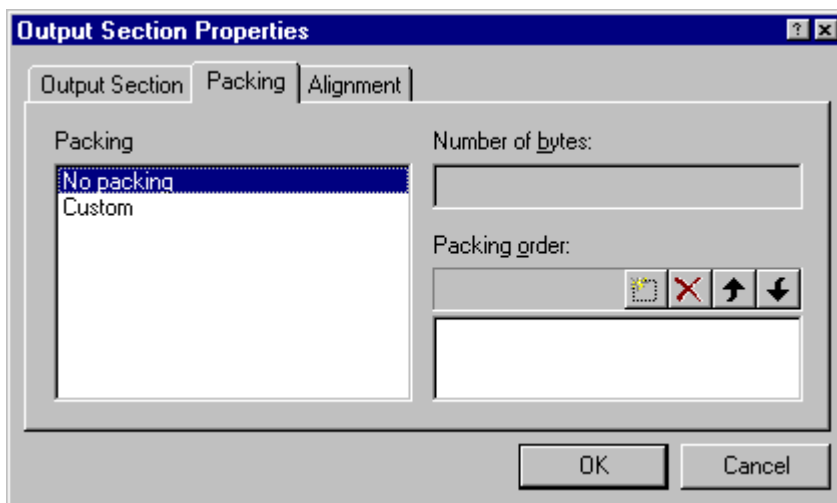


Figure 3-41. Memory Segment Properties Dialog Box – Packing Tab

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 zeroes (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.

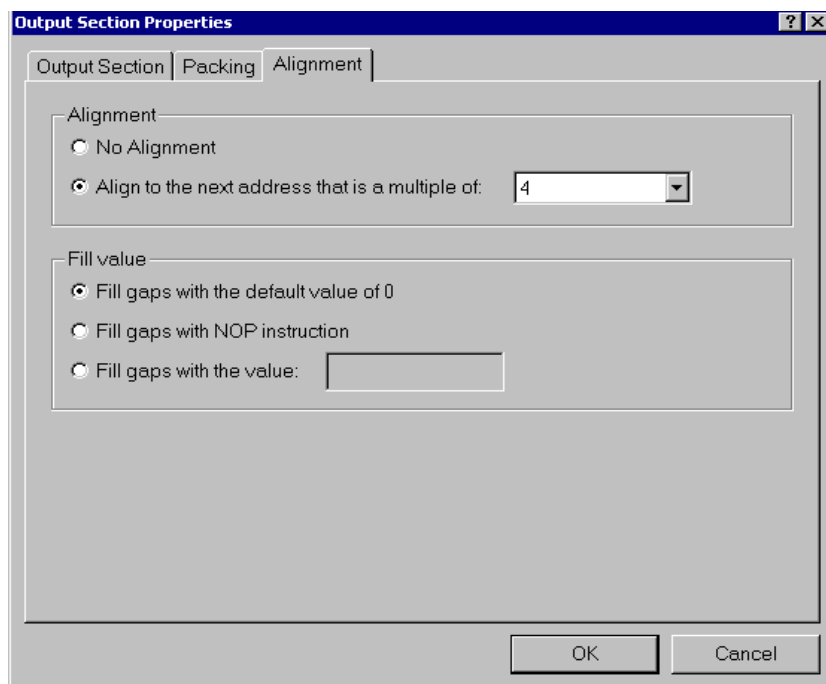


Figure 3-42. Output Section Properties Dialog Box – Alignment Tab

No Alignment specifies no alignment.

If you select **Align to the next address that is a multiple of**, select an integer value from the drop-down list to specify the output section alignment.

When the output section is aligned on an address, there is a gap that is filled by the linker. Based on the processor architecture, the Expert Linker determines the opcode for the NOP instruction.

The **Fill value** is either 0, a NOP instruction, or a user-specified value (a hexadecimal value entered in the entry box).

Managing Overlay Properties

The **Overlay** tab allows you to choose the output file for the overlay, its live memory, and its linking algorithm.

To specify overlay properties:

1. Right-click an overlay object in the **Memory Map** pane.
2. Choose **Properties**.

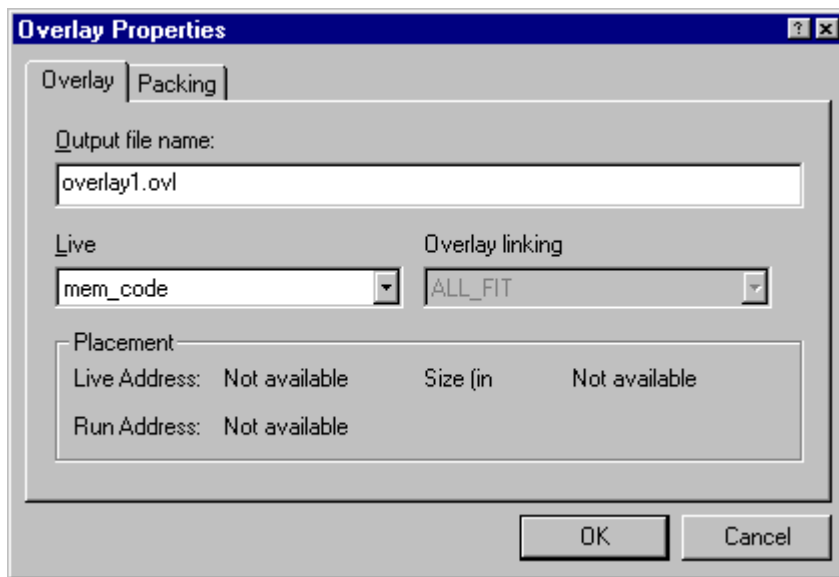


Figure 3-43. Overlay Properties Dialog Box – Overlay Tab

Live Memory contains a list of all output sections or memory segments with one output section. The live memory is where the overlay is stored before it is swapped into memory.

The **Overlay linking algorithm** box permits one overlay algorithm: `ALL_FIT`. Expert Linker does not allow you to change this setting. When using `ALL_FIT`, the linker tries to fit all of the mapped objects into one overlay.

The **Browse** button is available only if the overlay has already been built and the symbols are available. Clicking **Browse** opens the **Browse Symbols** dialog box (see [Figure 3-38 on page 3-50](#)).

You can choose the address for the symbol group or let the linker choose the address.

Managing Stack and Heap in DSP Memory

The Expert Linker shows how much space is allocated for your program's heap and stack. For ADSP-21xx DSPs, be aware of these stack/heap restrictions:

- The heap and stack must be defined in output sections named `sec_heap` and `sec_stack`, respectively.
- The heap and stack must be the only items in those output sections. You cannot place objects into these sections.

Figure 3-44 shows stack/heap output sections in the **Memory Map** pane. Right-click on either of them to display its properties.

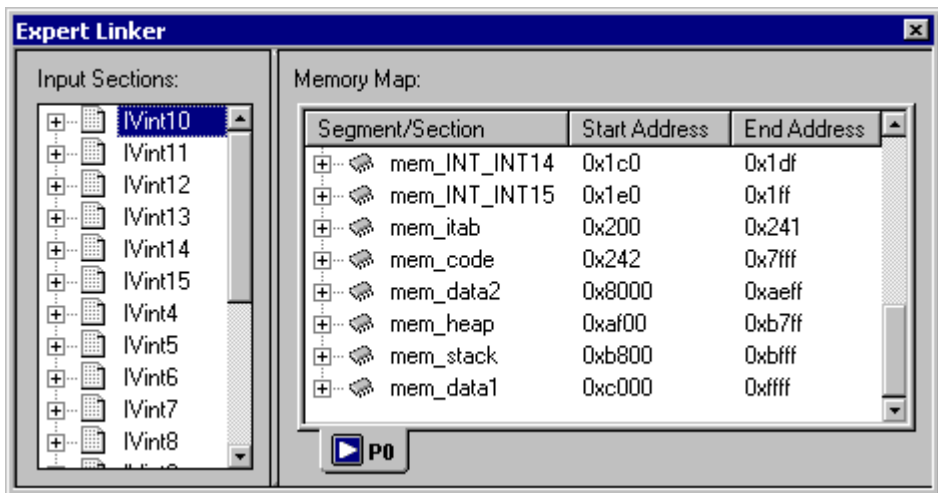


Figure 3-44. Memory Map Window With STACK/HEAP Sections

Use the **Global Properties** dialog box to select **Show stack/heap usage**. This option graphically displays the stack/heap usage in memory (Figure 3-46).

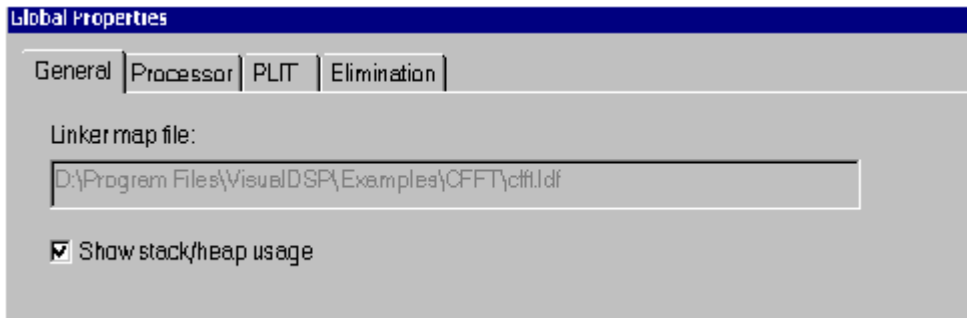


Figure 3-45. Global Properties – Selecting Stack/Heap Usage

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 occurs when the target halts after running the program. Assume this as the start of the unused portion of the stack or heap. The Expert Linker updates the memory map to show how much of the stack and heap are unused.

Use this information to adjust the size of your stack and heap making better use of your DSP memory more if the stack and heap segments use up too much memory.

Use the graphical view (**View Mode -> Graphical Memory Map**) to display stack/heap memory map blocks. [Figure 3-46](#) shows a possible memory map after running a Blackfin DSP project program.

Managing Object Properties

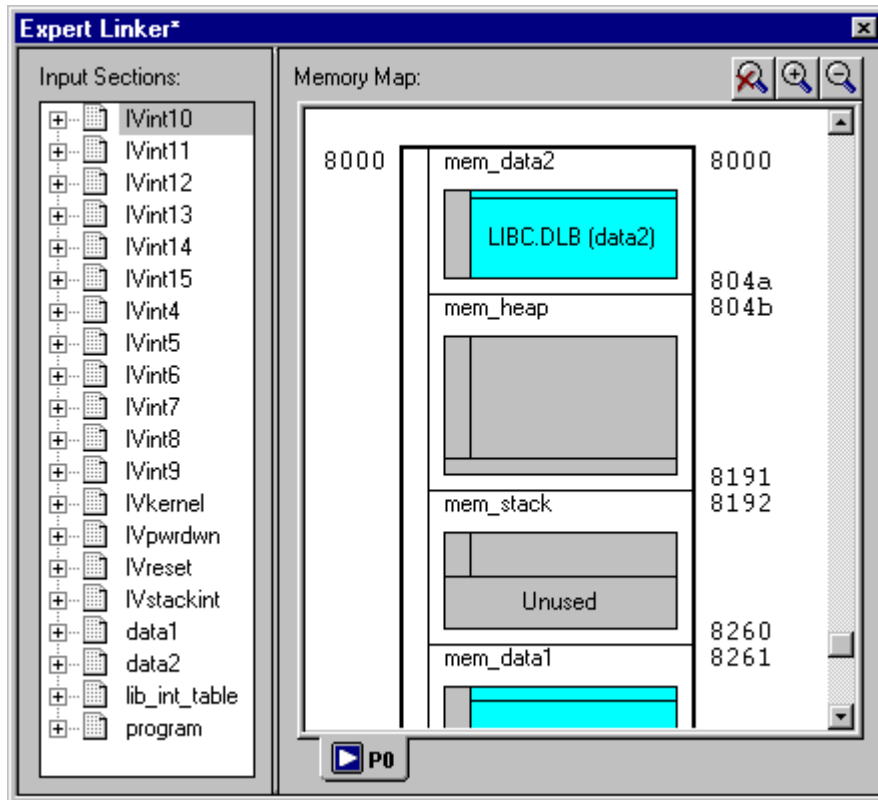


Figure 3-46. Graphical Memory Map Showing Stack/Heap Usage

4 ADSP-218X LOADER/SPLITTER

Use the ADSP-218x loader/splitter (`elfspl21.exe`) to convert executable files into boot-loadable (or non-bootable) files for ADSP-218x DSPs.

A boot-loadable file is transported into (and run from) DSP internal memory. A non-bootable PROM image file executes from DSP external memory.

 The preparation of a non-bootable image is also called splitting (PROM splitting). In most cases, developers working with an ADSP-21xx DSP use the loader instead of the splitter.

To generate a boot-loadable file, specify options via the **Load** page of the **Project Options** dialog box within the VisualDSP++ environment. VisualDSP++ invokes the `elfspl21` utility and builds the output file.

To generate a non-bootable PROM file, you must run the `elfspl21` utility from a command window.

This chapter contains the following information:

- [“ADSP-218x Loader Guide” on page 4-2](#)
- [“ADSP-218x Loader BDMA Command-Line Reference” on page 4-8](#)
- [“ADSP218x Splitter” on page 4-16](#)
- [“Splitter Command-Line Reference” on page 4-17](#)

ADSP-218x Loader Guide

The loader/splitter (`elfspl21.exe`) processes executable (`.EXE`) files, producing a boot-loadable (`.LDR`) file. Loader operations depend on loader options.

Loader options control how the loader processes executable files, letting you select features such as loader kernel, boot type, and file format. These options appear on the loader command line or the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment.

 Option setting on the **Load** page correspond to switches in the command-line version of the loader.

You should be familiar with the following operations:

- [“Boot Types” on page 4-3](#)
- [“Determining Boot Mode” on page 4-4](#)
- [“EPROM Booting \(BDMA\)” on page 4-6](#)
- [“IDMA - Host Booting” on page 4-13](#)
- [“Non-Bootable EPROM Images” on page 4-15](#)
- [“Using the Splitter” on page 4-16](#)

Boot Types

While working within a VisualDSP++ (simulation, emulation, or EZ-KIT Lite) session, you use the executable (.EXE) file, which conforms to the ELF standard.

Upon completing the debug cycle, you must transform the project data to a format readable by the DSP. This process is handled by the loader/splitter (elfspl21.exe) utility.

ADSP-218x DSPs provide these program-execution options:

- EPROM booting (BDMA)
- Host booting (IDMA)
- No booting

EPROM Booting: (BDMA) In this mode, project data is stored in an 8-bit wide PROM. After reset, the DSP performs a special bootstrapping scenario. It reads the PROM's content through the BDMA interface and initializes on-chip and off-chip memories. The elfspl21.exe loader utility generates a PROM image that contains all project data and loader code.

Host Booting: (IDMA) In this mode, the DSP does not start program execution immediately, but waits passively until a host processor (such as a microcontroller or another ADSP-218x) writes project data into the DSP's on-chip memory through the IDMA interface. The elfspl21.exe loader utility processes the project data, but the data may require post-processing because each type of host processor requires its individual data format.

No-Boot Option: In this mode, the DSP does not perform booting. After reset, the DSP starts program execution directly from off-chip 24-bit PROM memory. The splitter capabilities of the elfspl21.exe utility generate a proper PROM hex file. This option is not often used. You must run elfspl21.exe from the command line window.

Determining Boot Mode

To determine the boot mode, an ADSP-218x DSP samples its mode pins after reset. The following tables illustrate how to configure the DSP by pulling the proper pins up or down.

Table 4-1. Modes of Operation - ADSP-2181 and ADSP-2183 Only

MMAP Pin	BMODE Pin	Description
0	0	BDMA is used in default mode to load the first 32 program memory words from byte memory space. Program execution is held off until all 32 words have been loaded.
0	1	IDMA is used to load any internal memory as desired. Program execution is held off until internal program memory location 0 is written to.
1	X	Bootstrap is disabled. Program execution immediately starts from location 0.

Table 4-2. Modes of Operation - ADSP-2184 to ADSP-2189

Mode D	Mode C	Mode B	Mode A	Description
X	0	0	0	BDMA is used to load the first 32 program memory words from byte memory space. Program execution is held off until all 32 words have been loaded. Chip is configured in Full Memory Mode. *
X	0	1	0	No automatic boot operations occur. Program execution starts at external memory location 0. Chip is configured in Full Memory Mode. BDMA can still be used, but the processor does not automatically use or wait for these operations.
0	1	0	0	BDMA is used to load the first 32 program memory words from the byte memory space. Program execution is held off until all 32 words have been loaded. Chip is configured in Host Mode. /IACK has active pull-down.* (REQUIRES ADDITIONAL HARDWARE).

Table 4-2. Modes of Operation - ADSP-2184 to ADSP-2189

Mode D	Mode C	Mode B	Mode A	Description
0	1	0	1	IDMA is used to load any internal memory as desired. Program execution is held off until the host writes to internal program memory location 0. Chip is configured in Host Mode. /IACK has active pull-down. *
1	1	0	0	BDMA is used to load the first 32 program memory words from byte memory space. Program execution is held off until all 32 words have been loaded. Chip is configured in Host Mode; /IACK requires external pull down. (REQUIRES ADDITIONAL HARDWARE)
1	1	0	1	IDMA is used to load any internal memory as desired. Program execution is held off until the host writes to internal program memory location 0. Chip is configured in Host Mode. /IACK requires external pull-down. *



* Considered as standard operating settings. Using these configurations allows for easier design and better memory management.

The following DSPs have the Mode D mode of operation (Mode D pin).

Table 4-3. Processors that Support Mode D Operation

ADSP-2185M	ADSP-2184N
ADSP-2186M	ADSP-2185N
ADSP-2187L	ADSP-2186N
ADSP-2188M	ADSP-2187N
ADSP-2189M	ADSP-2188N
	ADSP-2189N

EPROM Booting (BDMA)

To generate a PROM image for EPROM booting, invoke the `elfspl21.exe` loader utility from the command line or change the VisualDSP++ project type to **Splitter file** and specify options on the **Project Options** dialog box's **Load** page. For more information on IDDE, see the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs Manual* or online Help.

After reset, the ADSP-218x DSP loads the 96 bytes from PROM address 0x0000 into the first 32 locations of on-chip PM memory. Assuming these 96 bytes consist of 32 valid instructions, the DSP executes this piece of program (preloader) afterwards. Usually 32 instructions are not sufficient to load the complete project data; therefore, bootstrapping continues and the preloader loads a set of so-called page loaders beginning at PM address 0x0020. After the preloader terminates, the DSP executes the page loaders, which load the project data page by page.

The loader utility uses a default preloader. You can force the loader with the `-uload` switch to use a customized preloader to reduce wait-states or to implement a boot management scenario as discussed in *Application Note EE-146*.

Example

The following example lists the default preloader together with its opcode. Note that the `BWCOUNT` value is generated dynamically.

```
/* standard preloader (32 instructions)    address  opcodes */
ax0 = 0x0060; dm(0x3fe2) = ax0; /* BEAD  0x0000:400600 93FE20 */
ax0 = 0x0020; dm(0x3fe1) = ax0; /* BIAD  0x0002:400200 93FE10 */
ax0 = 0x0000; dm(0x3fe3) = ax0; /* CTRL  0x0004:400000 93FE30 */
ax0 = 0x0087; dm(0x3fe4) = ax0; /* BWCOUNT0x0006: 400870 93FE40 */
ifc = 0x0008;nop;                /* BDMA IRQ 0x0008: 3C008C 000000 */
imask = 0x0008;                  /* 0x000A: 3C0083 */
idle;                            /* 0x000B: 028000 */
```

```
jump 0x0020; nop; nop; nop;          /* 0x000C: 18020F 000000 */
nop;          nop; nop; nop;          /* 0x0010: 000000 000000 */
nop;          nop; nop; nop;          /* 0x0014: 000000 000000 */
nop;          nop; nop; nop;          /* 0x0018: 000000 000000 */
rti;          nop; nop; nop;          /* 0x001C: 0A001F 000000 */
```

The page loaders are generated dynamically by `elfspl21.exe`, depending on the number of overlay pages to be initialized. A page loader sets up BDMA transfers. Since BDMA transfers cannot target off-chip DM and PM memories (overlay pages 1 and 2) directly, the corresponding page loaders first BDMA-load the data into on-chip memory and copy the data by core instructions afterwards.

The final BDMA sequence has the BCR bit set. It overwrites preloader and page loaders resident in internal PM memory and issues a context reset once it has finished. The program counter resets to 0x0000 and program execution begins.

Refer to the DSP's *Hardware Reference Manual* for a detailed description of the BDMA capabilities. You may debug the EPROM boot process using the VisualDSP++ simulator by loading the `.BNM` file (Settings->Simulator) and then resetting (Debug->Reset) the DSP to start booting.

The loader utility outputs industry-standard file formats like Intel Hex-32 or Motorola S, which are readable by most EPROM burners. For advanced usage, other file formats are supported; refer to [“File Formats” on page A-1](#) for details. The default file extension is `.BNM`.

ADSP-218x Loader BDMA Command-Line Reference

Invoke the `elfspl21.exe` loader utility from the command line to generate a PROM image for BDMA booting, or change the VisualDSP++ project type to Splitter file from Project page of the Project Options dialog box and specify the options on the Load page. For more information on IDDE, see the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs* or online Help. This section discusses the command-line only.

Use the following syntax for the ADSP-218x BDMA loader command line.

```
elfspl21 sourcefile outputfile -switch [-switch ...]
```

where:

- `sourcefile` — Identifies the executable file (.DXE) to be processed for a single-processor boot-loadable file. A file name can include the drive, directory, file name and file extension. Enclose long file names within straight-quotes, "long file name".
- `outputfile` — Specifies the output file. A file name can include the drive, directory, file name and file extension (.bnm).
- `-switch` — Optional switch to be processed. The loader has many switches to select operations and modes.



Command-line switches may occur in any order, except for the `sourcefile` and `outputfile` files, `-2181` (or `-218x`) and `-loader` switches. The `sourcefile` and `outputfile` files must be placed first on the command line. If required, the `-2181` (or `-218x`) and `-loader` switches must always follow the `outputfile` name, and the `-2181` (or `-218x`) switches must always be placed before the `-loader` switch.

Example

```
elfspl21 p0.dxe output.bnm -218x -loader -i -uload test.doj
```

In the above example, the ADSP-218x BDMA loader command line runs the loader with:

- `p0.dxe` — identifies the executable file to process into a boot-loadable file.
- `output.bnm` — Specifies the loader output file name.
- `-218x` — Specifies that the target processor is one of the ADSP-2184 through ADSP-2189 DSPs. Always specify either `-2181` or `-218x` when working with an ADSP-218x family DSP.
- `-loader` — Specifies EPROM booting as the boot-type for the boot-loadable file.
- `-i` — Specifies Intel hex-32 format for the boot-loadable file.
- `-uload` — Specifies the use of `test.doj` as the boot-kernel for the boot-loadable file.

File Names

Many loader switches take a file name as an optional parameter. [Figure 4-5 on page 4-11](#) lists the expected file types, names, and extensions.

File searches are important in the loader's process. The loader supports relative and absolute directory names, default directories. File searches occur as follows:

- *Specified path* — If you include relative or absolute path information in a file name, the loader searches only in that location for the file.
- *Default directory* — If you do not include path information in the file name, the loader searches for the file in the current working directory.

When you provide an input or output file name as a command-line parameter, use the following guidelines:

- Enclose long file names within straight-quotes, "long file name".
- Append the appropriate file extension to each file.

File Extensions

The loader follows the conventions for file extensions in [Table 4-7](#).

Table 4-4. File Extensions

Extension	File Description
.DXE	Executable files and boot-kernel files. The loader recognizes overlay memory (.OVL) files, but does not expect these files on the command line. Place .OVL files in the same directory as the .DXE file that refers to them so the loader can find them when processing the .BNM file.
.BNM	Loader output file for EPROMs
.IDM	Loader output file for IDMA

Loader Switches

Descriptions for each switch appear in [Table 4-5 on page 4-11](#).

Table 4-5. ADSP-218x BDMA elfspl21.exe Command-Line Switches ¹

Switch	Description
sourcefile	This parameter without a switch selects an executable file for single-processor boot-loadable files. For information on shared and overlay memory files, see Table 4-7 on page 4-19 .
outputfile	Specifies the loader's output file.
-h	Outputs the list of command-line switches to standard output and exits
-i	Produces Intel hex format
-s	Produces Motorola S2 format
-byte	Produces byte-stream output format
-2181	When used with -2181 -loader, keeps the image
-218x	PMOVLAY/DMOVLAY pages support. Use in place of -2181 (for example, -2187, -2188, or -2189).
-loader	Includes 218x default boot loader
-noloader	Excludes 218x boot loader. When used with -2181 -loader (or -2181 -loader), generates a byte memory image without a boot loader. This suppresses the preloader and page loaders.
-bdma <i>inputfile</i> <i>start_address</i>	(Use with -2181 -loader) Specifies placement of an additional .DXE file (inputfile) in byte memory, starting at the specified address. The loader returns an error if the specified address is in use by the boot loader, or another additional -bdma specified file. Files specified this way may be BDMA loaded at runtime, but the individual start addresses (and length and target information) must be predefined.
-bdmaload <i>infile</i> <i>start_address</i>	Specifies an additional .DXE file (infile) to be placed in byte memory, starting at the specified address. The address must be a multiple of 0x4000. Unlike the -bdma switch, this option generates a preloader and page loaders for the specified .DXE file. This way, two applications (or more) can be stored in the same EPROM and may replace the default application at runtime.

Table 4-5. ADSP-218x BDMA elfspl21.exe Command-Line Switches

Switch	Description
<code>-uload <i>file</i></code>	Reads the BDMA preloader from < <i>file.doj</i> >. The preloader must consist of exactly 32 instructions. Preloaders generated by the elfspl21 utility automatically determine the BWCOUNT value, which mirrors the numbers of instructions required by the page loaders. Customized preloaders do not have access to this length information and should always set BWCOUNT by assuming the maximal possible page loader length. In this software version, the maximal number of instructions required for the page loader is 658. This may change in future releases.
<code>offsetaddr #</code>	Provides byte memory address offset (valid only with <code>-2181 -noloader</code> or with <code>-218x -noloader</code>)
<code>offsetpage #</code>	Provides byte memory page offset (valid only with <code>-2181 -noloader</code> or with <code>-218x -noloader</code>)

¹ Switches are optional. Items in *italics* are user-definable.

IDMA - Host Booting

When booted through the IDMA interface, an ADSP-218x DSP behaves like a slave. After reset, it does not start program execution until internal PM location 0x0000 is overwritten by an IDMA write access. The host processor initializes all on-chip memories of the DSP, but writes to PM address 0x0000 lastly. Then the DSP starts program execution.

The host is responsible for handling the IDMA traffic properly. Typically, the host boots the DSP page-by-page or more generally, segment-by-segment. To boot a segment, the host first performs an address latch cycle to program the IDMA Control register and the IDMA Overlay register (ADSP-2184 through ADSP-2189 only). Afterwards, the host writes 16- or 24-bit words to the IDMA port. The DSP auto-increments its address counter. “[Host Processor Algorithm](#)” on [page 4-14](#) illustrates the algorithm the host processor must compute to boot the DSP successfully.

The loader utility generates an ASCII file (.IDM). Every segment data is headed by the following information: word count, IDMA control value, and overlay page number. Since the IDMA interface is a 16-bit interface, the .IDM file is organized in 16-bit portions. For details, refer to “[IDMA Host Boot File \(.IDM\)](#)” on [page A-10](#).

Typically, embedded processors cannot directly process ASCII files in .IDM format. Since an individual type of host processor may require many different formats, VisualDSP++ outputs this easy-to-handle format only. It is up to the user to post-process this file in a customized way.



Due to hardware restrictions, IDMA booting of off-chip memories is not possible. Refer to the description of IDMA capabilities in the *ADSP-218x DSP Hardware Reference*.

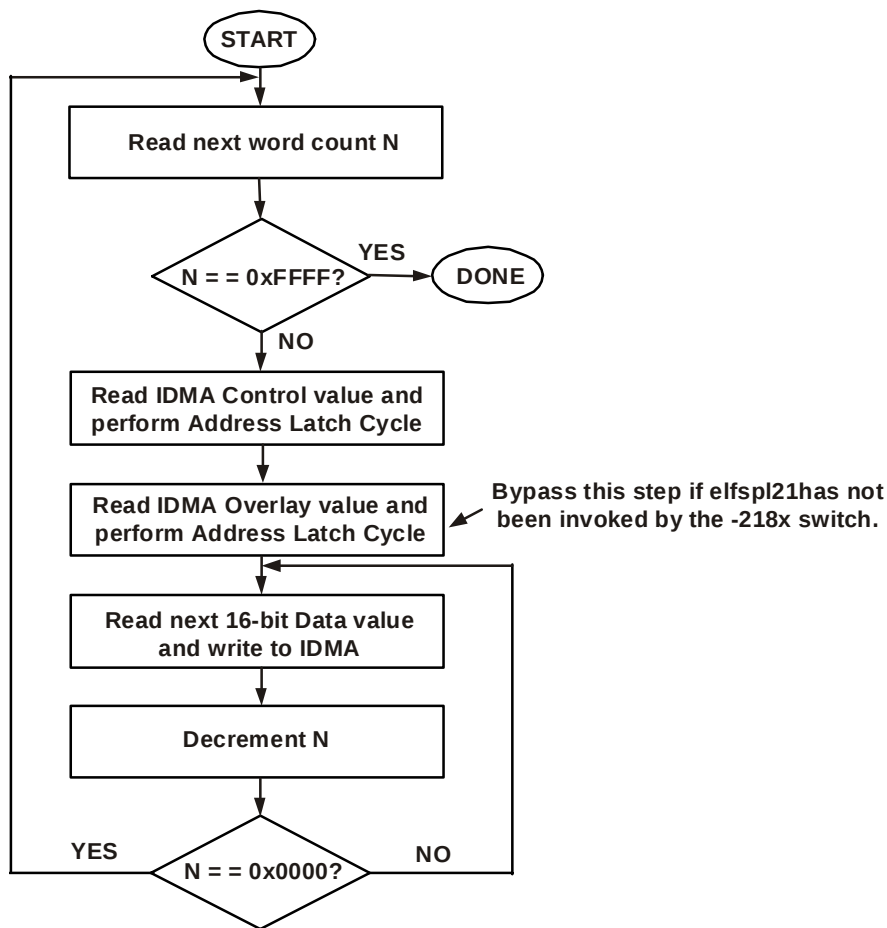


Figure 4-1. Host Processor Algorithm

IDMA Loader Command-Line Reference

To create an IDMA boot file, use the following syntax for the ADSP-218x loader command line.

```
elfspl21 sourcefile outputfile [-2181 | -218x] -idma
```

Table 4-6. ADSP-218x IDMA elfspl21.exe Command-Line Switches

Switch	Description
sourcefile	Identifies the executable file (.DXE) to be processed for a single-processor boot-loadable file. A file name can include the drive, directory, file name, and file extension. Enclose long file names within straight-quotes, "long file name".
outputfile	Specifies the output file. A file name can include the drive, directory, file name, and file extension (.IDM).
-218x	Provides support for IDMA Overlay register
-2181	Use this switch unless you are using the -218x switch
-idma	Forces the loader to create an IDMA boot file. It overwrites most of the BDMA boot-specific options.

Non-Bootable EPROM Images

In very rare cases, applications require that the DSP not be booted after reset, or that the DSP is booted with a ROM connected to the DSP's off-chip DM/PM address space.

Preparing a non-bootable image is called splitting. In most cases, developers working with ADSP-218x DSPs use the loader instead of the splitter.

For ADSP-218x DSPs, splitter and loader features are handled by the elfspl21.exe utility. The tools must be invoked by a completely different set of command-line switches. Refer to [“ADSP218x Splitter” on page 4-16](#).

ADSP218x Splitter

Use the splitter (`elfspl21.exe`) utility to process an executable file to produce a non-bootable, PROM-image file.

In most cases, developers working with ADSP-21xx DSPs should use the loader instead of the splitter.

This section contains the following information on the splitter:

- “[Using the Splitter](#)” on page 4-16 describes how to use the splitter from VisualDSP++
- “[Splitter Command-Line Reference](#)” on page 4-17 describes splitter commands and operation

Using the Splitter

 You must run the splitter (PROM splitter) via a command window. You cannot generate a non-bootable PROM file from within the VisualDSP++ environment. To automate the process, specify the splitter command-line within VisualDSP++ from the **Post Build** page of the **Project Options** dialog box.

The ADSP-218x splitter generates images for external PMOVLAY (1 and 2) and DMOVLAY (1 and 2) memory pages. If you use the splitter to produce ROM images (for example, ADSP-2181 program memory pages 1 and 2), the generated code must target ROM. Define the appropriate ROM segments in the `.LDF` file.

Splitter options control how the splitter processes your executable files, letting you select features such as memory type and file format.

Splitter Command-Line Reference

The splitter (`elfspl21.exe`) generates non-bootable, PROM-image files for ADSP-218x DSPs by processing executable files (`.DXE`). The splitter's output is a PROM file with the file extension `.BNL`, `.BNU`, or `.BNM`.

This section provides reference information about the splitter command line and PROM splitting. A list of all switches and their descriptions appears in [“Splitter Command-Line Switches” on page 4-20](#).

Syntax



The splitter command line is case sensitive.

The splitter command line takes the following syntax:

```
elfspl21 sourcefile outputfile -pm &| -dm [-switch ...]
```

where:

- `[-switch ...]` — One or more switches to be processed. Switches select the operations and modes for the splitter.
- `&|` — Indicates AND-OR.
- `sourcefile` — Name of the executable file (`.DXE`) to be processed for a single-processor boot-loadable file. A file name can include the drive, directory, file name and file extension. Enclose long file names within straight-quotes, “long file name”.

Example

The following two command lines, for example, run the splitter twice, first producing PROM files for program memory and then producing PROM files for data memory.

```
elfspl21 my_proj.dxe pm_stuff -pm
elfspl21 my_proj.dxe dm_stuff -dm
```

Splitter Command-Line Reference

The switches on these command lines are as follows.

`-pm -dm`

Select program memory or data memory as the source in the executable for extraction and placement into the image. Because these are the only switches used to identify the memory source, the source are any PM or DM ROM memory segments. Because no other contents switches appear on these command lines, the format for the output defaults to Motorola S3 format, and the output PROM width defaults to 8 bits for all PROMs.

`pm_stuff dm_stuff`

Specify names for the output files without file extension. Use different names so the output of the second run does not overwrite the output of the first run. The output names are `pm_stuff.s_#` and `dm_stuff.s_#`.

`my_proj.dxe`

Specifies an executable file to process into a non-bootable PROM-image file.

File Names

Many splitter switches take a file name as an optional parameter. [“Splitter Command-Line Switches” on page 4-20](#) lists the type of files, names, and extensions that the splitter expects on files. File searches are important in the splitter’s process. The splitter supports relative and absolute directory names, default directories, and user-selected directories for file search paths.

When you provide an input or output file name as a command-line parameter, use the following guidelines:

- Enclose long file names within straight-quotes, “long file name”.
- Append the appropriate file extension to each file.

File searches occur as follows:

- Specified path — If you include relative or absolute path information in a file name, the splitter searches for the file in that location only.
- Default directory — If you do not include path information in the file name, the splitter searches for the file in the current working

File Extensions

The splitter follows the conventions for file names in [Table 4-7](#).

Table 4-7. File Name Extensions

Extension	File Description
.DXE	Executable files and boot-kernel files
.BNM	Splitter binary output file — middle
.BNU	Splitter binary output file — upper
.BNL	Splitter binary output file — lower

Splitter Switches

[Table 4-8 on page 4-20](#) lists the available switches.

Splitter Command-Line Reference

Table 4-8. Splitter Command-Line Switches

Switch	Description
sourcefile	Specifies the source for the splitter operation.
outputfile	Specifies the splitter's output file. Uses the image name for the splitter's output file(s). If not specified, the default name is executable.ext where ext depends on the output format.
-dm	Extracts data memory. Extracts segments from the executable declared as data memory. The splitter generates two one-byte files. One file (.BNM) contains the upper bytes of the 16-bit data words. A second file (.BNL) contains the lower bytes.
-pm	Extracts program memory. Extracts segments from the executable declared as program memory. The splitter generates three one-byte files. One file (.BNU) contains the upper bytes of the 24-bit words. A second file (.BNM) contains the middle bytes. A third file (.BNL) contains the lowest bytes.
-readall	Includes RAM and ROM in PROM. Extracts both RAM and ROM segments from the input file(s). By default, only ROM segments are extracted.
-i	Produces Intel hex format
-us -us2 -ui	Produces a byte-stacked format file for 8-bit memory. -us yields Motorola S1 format -us2 yields Motorola S2 format -ui yields Intel format
-s	Produces s1 format
-byte	Produces byte-stream output format

5 ADSP-219X LOADER/SPLITTER

Use the ADSP-219x loader/splitter utility (`elfloader.exe`) to create boot stream, non-boot stream, or combinational output. This utility accepts one executable file (`.DXX`) as input and generates one output file (`.LDR`).

 In this chapter, ADSP-219x refers to any of the following DSPs: ADSP-2191, ADSP-2195, ADSP-2196, ADSP-21990, ADSP-21991, and ADSP-21992.

For information about ADSP-2192-12 DSPs, refer to [“ADSP-2192-12 Loader” on page 6-1](#).

Run the loader/splitter utility from a command window or directly from the VisualDSP++ IDDE. When working within the VisualDSP++, you specify options via the **Load** page of the **Project Options** dialog box.

This chapter contains the following information:

- [“ADSP-219x Booting” on page 5-2](#) contains loader procedures within the VisualDSP++ environment
- [“ADSP-219x Boot Loader Utility” on page 5-16](#) contains reference information on the loader commands and operations

For more information on VisualDSP++ IDDE, see the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs* or online Help.

ADSP-219x Booting

Upon power-up, the DSP can be booted via the EPROM, UART, SPI, or Host port. Booting can also be initiated in software after reset. Refer to *Application Note EE-131* and your DSP's *Hardware Reference* for details.

Before selecting loader/splitter options, become familiar with the loader/splitter features, how they apply to the boot-type you plan to use, and the specific loader/splitter features that support these boot-types.

This section contains the following information:

- [“ADSP-219x Boot Kernel” on page 5-2](#)
- [“Boot Modes” on page 5-3](#)
- [“Boot Stream Contents” on page 5-4](#)
- [“Parallel EPROM Booting” on page 5-4](#)
- [“Host Booting” on page 5-7](#)
- [“UART Booting” on page 5-7](#)
- [“Serial EPROM Booting” on page 5-8](#)
- [“No-Boot Mode” on page 5-9](#)

ADSP-219x Boot Kernel

Loader/boot kernel refers to the resident program in the BOOT ROM space responsible for booting the DSP. When the ADSP-219x comes out of a hardware /RESET, program control jumps to 0xFF0000, and execution of the boot ROM code begins.

In the ADSP-219x, the highest 16 locations in page 0 program memory and the highest 272 locations in page 0 data memory are reserved for use by the ROM boot routines (typically for setting up DMA data structures and for bookkeeping operations). Ensure that the boot sequence entry code or boot-loaded program are not allowed into this space.

Boot Modes

The DSP's `BMODE2-0` pins, which are sampled during hardware reset, control the boot mode that takes affect following a hardware reset.

The state of these three pins are captured and are placed in Reset Configuration register as `BMODE0`, `BMODE1`, and `OPMODE`, respectively. This register is also known as the System Configuration register (I/O address `0x00204`).

Table 5-1. Modes of Operation for ADSP-219x DSPs

BMODE1 Pin	BMODE0 Pin	OPMODE Pin	Description
0	0	0	No boot. Run from external 16-bit memory at logical address <code>0x10000</code> . Bypass ROM.
0	1	0	Boot from EPROM
1	0	0	Boot from Host
1	1	0	Reserved
0	0	1	No Boot. Run from external 8-bit memory at logical address <code>0x10000</code> . Bypass ROM.
0	1	1	Boot from UART
1	0	1	Boot from SPI (up to 4K bits)

The `OPMODE` pin has a dual role (it acts as a boot-mode select during /RESET and determines whether the DSP's third SPORT functions as a SPORT or an SPI). It is possible for an application to require `OPMODE` to

ADSP-219x Booting

operate differently at runtime than at /RESET (that is, boot from an SPI but use SPORT2 during runtime). In this case, the boot kernel is responsible for setting `OPMODE` accordingly at the end of the booting process.

Thus, software can change `OPMODE` anytime during runtime, as long as the corresponding peripherals are disabled at that time.

Boot Stream Contents

In ADSP-219x DSPs, the on-chip boot kernel is stored in a 24-bit wide, 1K portion of ROM. The starting address of this boot ROM is memory address `0xFF0000` (the first location of page 256), in 1-wait-stated memory.

Use a command-line option in the loader utility to include a user-specified loader kernel as part of and the first component of the boot-stream. In this case, the boot ROM sets up an initial DMA to load in the loader kernel via the medium of booting and then transfers control to the loader kernel. It will have no further role in the booting process. This, in effect, is a two-stage booting process.

Parallel EPROM Booting

When booting from an external 8- or 16-bit EPROM, the boot-stream format consists of a “header” and “data blocks”.

The ADSP-219x ROM resident loader is designed to parse and load a specific boot-stream format. The boot stream contains a header field that specifies whether the stream is guarded by a checksum. The ADSP-219x on-chip ROM is located at addresses `0xFF 0000` to `0xFF 03FF`. A boot interrupt will vector to address `0xFF 0000`.

The first word (header) in the boot stream is a common word that applies to all booting formats except UART booting. Individual bits within this word are set or cleared, based on the booting method and specific command line options specified by the user and loader utility (“[elfloader.exe](#)

[Command-Line Items” on page 5-18](#)). This 16-bit field contains information on the number of wait-states and the physical width (8-bit or 16-bit) of the EPROM.

This first header is followed by the regular boot stream, that is, a series of “headers” and “data blocks”, with each header optionally followed by a corresponding block of data. Most headers are followed by data blocks, but some headers indicate regions of memory that need to be “zero-filled” and are not followed by data blocks.

Each header consists of four or six 16-bit words:

- The first word of a header consists of a flag that indicates whether the following block of data is a 24-bit or 16-bit payload or zero-initialized data. The flag uniquely identifies the last block that needs to be transferred.
- The second word contains the lower 16 bits of the 24-bit start address for data loading (destination). The first octet is the 8 LSBs, followed by the next most significant bits (8-15), and so on.
- The third word contains the upper-most 8 bits of the 24-bit destination address, padded (suffixed) with one byte of zeros.
- The fourth word contains the payload’s word count. Similar to the address, the first octet is the 8 LSBs, and the second octet is the 8 MSBs.

An extra word appears when a checksum is used to verify booting accuracy. This fifth word (also a 16-bit field) is the CRC-16 checksum for the header, and the data block immediately follows it. The header is buffered with a dummy word of zeros to ease the EMI addressing issue. The CRC checksum word is optional. Activate the checksum by running the elf-loader utility with the `-checksum` switch.

EPROM booting can be performed in an 8-bit or a 16-bit scenario. This information is controlled by the `-width` command-line switch and is finally embedded in the boot stream. Unlike in non-boot mode, bus width is not controlled by the mode pins. The width information configures the EMI interface and remains valid during the entire boot process. If you want to boot off-chip memories, be aware that the width of the memory you want to boot must be identical to the width of the interface from which you are booting.

The header is followed by the payload data. 16-bit data is sent in a 16-bit field, and 24-bit data is sent in a 32-bit field.

 24-bit data is represented differently in the boot stream from 24-bit addresses. 32-bit data is transmitted the following way — a byte of zeros (inserted by elfloader), bits 0-7, followed by bits 8-15, and finally bits 16-24.

When booting from an 8- or 16-bit EPROM, direct DSP core accesses and Memory DMA (under the control of an EPROM boot routine located in the ROM space) are used to load a boot-stream formatted program located in the boot space. Appropriate packing modes are selected, based on the requirements of the boot stream.

Each page of boot space is 64K words long, and 16-bits can address the EPROM per page. The upper 8 address bits specify the boot page. Upon hardware reset, booting is from address 0x0000 of logical page 0x80 which mirrors physical address 0x000000, because the upper address lines are not available off-chip.

The External Port Interface (EPI) uses its default configuration (divide-by-128 clock and 7 wait states) to access the EPROM. While booting via EPROM boot space, the highest 16 locations in page 0 program memory block (0x7FF0 to 0x7FFF) and the top 272 locations of page 0 data memory block (0xFE0 to 0xFFFF) are reserved for use by the ROM boot routines.

Host Booting

Host booting is performed in either an 8-bit or a 16-bit scenario. By default, little-endian format is used. If configured in host boot mode, the DSP does not support the boot process actively. It is the host's responsibility to initialize the DSP memories properly. The DSP passively waits until the host writes a "1" to the Semaphore A register (IO:0x1CFC). Then the DSP starts fetching and executing instructions at address 0x000000.

It is recommended that the host parses the boot stream and downloads segment by segment. The elfloader utility may store the boot stream as an Intel hex-32 file typically required by embedded host devices. PC-based hosts may favor the ASCII format, which stores one byte or one 16-bit word per line.

UART Booting

When booting via the UART port, a host downloads a boot stream formatted program to the DSP following an auto-baud handshake sequence. The auto-baud feature simplifies things during system design because you do not need to calculate boot clock rates as a function of crystal frequency, core clock divider, peripheral clock mode, and so on.

The booting host can select a baud rate (including a non-standard rate) anywhere within the UART clocking capabilities.

Following a hardware or software reset, the ADSP-219x monitors the UART transceiver channel and expects the predefined character (0xAA) to determine the bit rate. The DSP replies an "OK" string to acknowledge the bit rate.

Afterwards, the host may send the complete boot stream (8 data bits, no parity, 1 stop bit) without further handshake. The boot stream is decoded by the DSP and starts program execution automatically.

 Compared to other formats, the ADSP-2191/95/96 UART boot stream suppresses the very first byte. To force the loader utility to include this first byte, use the `-forcefirstbyte` switch.

This boot operation is controlled by a UART boot routine in the internal ROM space. While booting via UART, the highest 16 locations in page 0 program memory block (0x7FF0 to 0x7FFF) and the top 272 locations of page 0 data memory block (0xFE0 to 0xFFFF) are reserved for use by the ROM boot routine.

Serial EPROM Booting

The SPI0 port is used when booting from an SPI-compatible EPROM. The SPI port selects a single serial EPROM device using the `PF0` pin as a chip select, submits a read command and address 0x00, and begins to clock consecutive data into memory (internal memory or external memory) at a `SCK` clock frequency of `HCLK/60`. The DSP streams the complete boot image in, and processes it without further handshake with the SPI EPROM.

Two types of SPI EEPROM devices are supported: devices of 4K bytes and smaller (12-bit address range), and those larger than 4K bytes (16-bit address range). The SPI boot stream may not exceed 64 Kilobytes.

This boot operation is controlled by an SPI boot routine in internal ROM space. While booting via serial EPROM, the highest 16 locations in page 0 program memory block (0x7FF0 to 0x7FFF) and the top 272 locations of page 0 data memory block (0xFE0 to 0xFFFF) are reserved for use by the ROM boot routine. Refer to *Application Note EE-145* for SPI booting examples.

No-Boot Mode

When `BMODE2-0` is strapped to a `000` or `001`, the ADSP-219x DSP comes out of hardware reset and begins to execute code from page 1 memory space (`0x01 0000`). The specified packing mode depends on the state of the `BMODE0` pin (`0` = 8-bit ext/24-bit int , `1` = 16-bit ext/24-bit int). By default, the External Port Interface (EPI) is configured to operate with the divide-by-128 clock and a read wait-state count of 7.



No-boot mode does not use boot-stream format.

After reset, the DSP starts program execution from external address `0x010000`. When the no-boot option is selected, the DSP typically expects an 8- or a 16-bit EPROM or flash device connected to the memory strobe signal (`/MS0`). Splitter capabilities of the ADSP-219x linker and elfloader utilities support the generation of the required EPROM image files. Refer to the elfloader's `-romsplitter` switch.

Non-bootable memory segments are declared by the `TYPE(ROM)` command in the Linker Description File (`.LDF`). The `WIDTH()` command specifies the physical EPROM width (which equals to the EMI port setting). Every `.LDF` file that belongs to a no-boot project should define a proper memory segment, for example

```
MEMORY {
    . . .
    seg_ext_code {
        TYPE(PM ROM) START(0x010000) END(0x017FFF) WIDTH(16) }
    . . .
}
```

The `START()`, `END()`, and `LENGTH()` commands always expect logical addresses. Since the example segment stores 24-bit-wide instructions, the `TYPE(PM)` command defines the logical width of the segment to be 24-bits. The example assumes no-boot mode (`000`), run from external memory

ADSP-219x Booting

starting at address 0x010000. This is why the `WIDTH(16)` command sets the physical width to 16 bits. Due to ADSP-219x EMI packing rules, the first instruction is stored in physical 16-bit EPROM location 0x020000.

The 16-bit EPROM address locations between 0x010000 and 0x01FFFF can be used by an additional read-only data segment as shown below.

```
MEMORY {  
    . . .  
    seg_ext_code {  
        TYPE(PM ROM) START(0x010000) END(0x017FFF) WIDTH(16) }  
    seg_ext_data {  
        TYPE(DM ROM) START(0x010000) END(0x01FFFF) WIDTH(16) }  
    . . .  
}
```

The data segment `seg_ext_data` is defined by the `TYPE(DM)` command, which sets the logical width to 16 bits. Since the `WIDTH(16)` command also sets the physical width to 16 bit, no data packing and no address multiply is required. Logical addresses are equal to physical EPROM addresses in this special case.

The 16-bit EPROM image generated by the `elfloader` utility would look like [Table 5-2](#):

Table 5-2. EPROM Image

Address range	Purpose
0x000000 - 0x00FFFF	Not used
0x010000 - 0x01FFFF	<code>seg_ext_data</code>
0x020000 - 0x02FFFF	<code>seg_ext_code</code>



The DSP cannot access off-chip addresses lower than 0x010000. Typically this address space is accessed by taking advantage of address aliasing. The memory strobe (/MS0) covers an address range from 0x010000 to 0x400000. For example, if the EPROM size is less than 4M words and the EPROM is the only device connected to /MS0, the first 64K words can be accessed through addresses 0x200000 to 0x20FFFF.

If a project consists only of two segments (`seg_ext_data` and `seg_ext_code`), a 128K x 16-bit EPROM device would be sufficient to store all the required data and instructions. If the elfloader utility is invoked with the `-maskaddr 17` switch, all physical address bits greater or equal to A17 are masked out. The elfloader ANDs all physical addresses with 0x01FFFF. The resulting EPROM image maps segment `seg_ext_code` into the unused space below 0x010000 would look like [Table 5-3](#).

Table 5-3. EPROM Image - Two Segments Only

Address range	Purpose
0x000000 - 0x00FFFF	<code>seg_ext_data</code>
0x010000 - 0x01FFFF	<code>seg_ext_code</code>

This way, a 128K x 16-bit EPROM can be burned properly. At runtime, `seg_ext_code` aliases back to the addresses above 0x020000 when address lines A17 through A21 are not connected.

Typically, the elfloader utility generates an Intel hex-32 file, which is readable by most EPROM programmers. If, for any reason, the image must be post-processed, the elfloader may also generate user-friendly ASCII files.

DM Example

```
ext_data { TYPE(DM ROM) START(0x010000) END(0x010003) WIDTH(8) }
```

The above DM segment results in the following code.

```
00010000    // 32-bit logical address field
00000004    // 32-bit logical length field
00020201    // 32-bit control word: 2x address multiply
            // 02 bytes logical width, 01 byte physical width
00000000    // reserved
1234        // 1st data word, DM data is 16 bits
5678
9ABC
DEF0        // 4th (last) data word
CRC16       // optional, controlled by the -checksum switch
```

PM Example

```
ext_code { TYPE(PM ROM) START(0x040000) END(0x040007) WIDTH(16)}
```

The above PM segment results in the following code.

```
00040000    // 32-bit logical address field
00000008    // 32-bit logical length field
00020302    // 32-bit control word: 2x address multiply
            // 03 bytes logical width, 02 bytes physical width
00000000    // reserved
123456      // 1st data word, PM data is 16 bits
789ABC
DEF012
345678
9ABCDE
F01234
56789A
BCDEF0      // 8th (last) data word optional,
            // controlled by the -checksum switch
```

Enriching Boot EPROMs with No-Boot Data

The elfloader's splitter functionality (refer to [“No-Boot Mode” on page 5-9](#)) enables powerful memory utilization in combination with the parallel EPROM boot mode. The same EPROM used for booting can also be used at runtime for read-only data and overlay storage. Furthermore, the DSP can execute non-speed-critical parts of the program directly from the EPROM, whereas real-time algorithms have been booted into on-chip memory.

When invoked with the `-readall` switch, the elfloader utility processes all kinds of LDF segments. `TYPE(RAM)` segments are passed to the elfloader's boot stream generator, and `TYPE(ROM)` segments are passed to its splitter. Boot stream and splitter data can be combined within one single EPROM image.

Assuming a cost-sensitive application comprising an ADSP-2196 and a 64-Kilobyte EPROM, the boot stream probably does not exceed 40 Kilobytes (8K x 3 bytes + 8K x 2 bytes) of length. The rest of the EPROM can be used to store different sets of coefficients and the slow initialization and control code. A reasonable organization of the 8-bit EPROM could like [Table 5-4.](#):

Table 5-4. EPROM Image With No-Boot Data

Address range	Purpose
0x000000 - 0x009FFF	Boot stream
0x00A000 - 0x00AFFF	seg_ext_data (External read-only data)
0x00B000 - 0x00FFFF	seg_ext_code (External program)

Since the DSP cannot access off-chip memories with addresses lower than 0x010000, it needs to access segments `seg_ext_data` and `seg_ext_code` through alias windows. If only address lines A0 through A15 are connected, address 0x00A000 aliases to any 0x??A000 address. Segment

ADSP-219x Booting

`seg_ext_data` stores 16-bit data. Thus, its physical addresses must be divided by 2 to obtain the corresponding logical address. The first alias window of `seg_ext_data` that can be accessed properly is range 0x02A000 to 0x02AFFE; this results in the logical range 0x015000 to 0x0157FF.

Similarly, the addresses of segment `seg_ext_code` can be calculated by dividing the physical addresses by 4. For example, the alias window between 0x04B000 and 0x04FFFF can be used, resulting in the logical addresses 0x012C00 to 0x013FFF.

The corresponding LDF file would look like:

```
MEMORY {  
    . . .  
    seg_int_code {  
        TYPE(PM RAM) START(0x000000) END(0x000000) WIDTH(24) }  
    seg_int_data {  
        TYPE(DM RAM) START(0x008000) END(0x009FFF) WIDTH(16) }  
    seg_ext_code {  
        TYPE(PM ROM) START(0x012C00) END(0x013FFF) WIDTH(8) }  
    seg_ext_data {  
        TYPE(DM ROM) START(0x015000) END(0x0157FF) WIDTH(8) }  
    . . .  
}
```

By default, the `elfloader` utility emits true EPROM addresses by multiplying the logical addresses accordingly. The address aliasing in this example takes advantage of and can be corrected if the `elfloader` is invoked with the `-maskaddr 16` switch. Then all EPROM addresses are ANDed by 0xFFFF and the resulting EPROM image fits into a 64-Kilobyte EPROM.

The EPROM boot process assumes the boot device is connected to the DSP's /BMS strobe. During runtime, typically the /MSx strobes are used. To use one EPROM for both booting and run-time issues, set the proper

/BMS control bits in the E_STAT register. If several devices are connected to the individual /MSx strobes, an off-chip AND gate is recommended to OR the /BMS and the /MS0 strobe properly.

ADSP-219x Boot Loader Utility

The loader (`elfloader.exe`) produces a boot stream file by processing an executable file (`.DXX` file) generated by the linker (`linker.exe`) and outputs a loader file (`.LDR` file).

This section provides reference information about the `elfloader.exe` command line and boot-loading. A description for each switch appears in [“File Name Extensions” on page 5-18](#).

 When using the linker within VisualDSP++, settings on the **Link** page of the **Project Options** dialog box correspond to linker command-line switches. For more information, see the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSP* or online Help.

elfloader.exe Command-Line Reference

Use the following syntax for the `elfloader.exe` command line:

```
elfloader sourcefile -proc ADSP-21xx -switch [-switch ...]
```

where:

- `sourcefile` — Specifies the executable file or files (`.DXX`) to be processed. The file name can include the drive, directory, file name and extension. Enclose long file names within straight-quotes, "long file name".
- `-switch` — Specifies the switch to be processed. The loader provides many switches to select loader operations and modes.
- `-proc -21xx` — This mandatory switch identifies the processor

Example

```
elfloader p0.dxe -proc ADSP-2191
```

In the above example, the loader command line runs the loader with:

- `p0.dxe` — Specifies the executable file to be processed into a boot-loadable file. By default, the output file's name is `p0.ldr` because a name is not specified with the `-o <filename>` switch.
- `-proc ADSP-2191` — Specifies output for an ADSP-2191.

File Names

Many loader switches take a file name as an optional parameter. [“File Name Extensions” on page 5-18](#) lists the type of files, names, and extensions that the loader expects on files.

File searches are important in the loader's process. The loader supports relative and absolute directory names, and default directories. File searches occur as follows:

- *Specified path* — If you include relative or absolute path information in a file name, the loader searches only that location.
- *Default directory* — If you do not include path information in the file name, the loader searches for the file in the current working directory.

When you provide an input file name or an output file name as a command-line parameter, use the following guidelines:

- Enclose long file names within straight-quotes, `"long file name"`.
- Append the appropriate file name extension to each file.

The loader follows the file extension conventions in [“File Name Extensions” on page 5-18](#).

ADSP-219x Boot Loader Utility

Table 5-5. File Name Extensions

Extension	File Description
.DXE ¹	Executable files and boot-kernel files
.LDR	Loader output file for EPROMs

- 1 The loader recognizes overlay memory (.OVL) files, but does not expect these files on the command line. Place .OVL files in the same directory as the .DXE file that refers to them so the loader can find .OVL files when processing the .LDR file.

elfloader.exe Switches

Descriptions for each switch appear in “[elfloader.exe Command-Line Items](#)” on page 5-18.

Table 5-6. elfloader.exe Command-Line Items¹

Switch	Description
<i>filename</i>	Without a switch, specifies an executable file for single-processor boot-loadable files. For multiprocessor boot-loadable files, include the <code>-id#exe=file</code> switch.
<code>-b type</code>	Boot type. Prepares a boot-loadable file for the <i>type</i> booting-mode. Valid <i>type</i> selections are PROM (default), HOST, UART, SPI, and NOBOOT. The specified boot type must correspond to the boot-kernel selected with the <code>-l</code> switch and the file format selected with the <code>-f</code> switch.
<code>-blocksize #</code>	Specifies the size (decimal) in words, of each block of data in the boot-stream. Default is 6K words. Valid block sizes may vary up to 64K words.
<code>-checksum</code>	Calculates and generates checksums for each block of code/data. Currently, the only valid input is “CRC16”.
<code>-clkdivide #</code>	(EPROM and HOST boot modes only) Specifies the base clock divide factor. Valid values are 0 to 7 (inclusive). The default is 5.
<code>-proc ADSP-21xx</code> or <code>-dADSP-21xx</code>	This mandatory switch specifies the processor (for example, <code>-proc ADSP-2191</code> or <code>-proc ADSP-21990</code>). Note that <code>-proc ADSP-21xx</code> is the preferred switch and <code>-dADSP-21xx</code> is for legacy support.
<code>-endian</code>	Specifies big endian format. Without this switch, little endian format (default) is used.

Table 5-6. elfloader.exe Command-Line Items¹ (Cont'd)

Switch	Description
-f <i>format</i>	Specifies the format of the boot-loadable PROM file output. The available <i>format</i> selections are hex (Intel Hex-32), ASCII, and binary. Hex is the default. For PROM booting, Intel hex is the only valid entry. When -f ASCII and -romsplitter are selected, regardless of the -b file_type setting, an ASCII file is produced.
-forcefirstbyte	Forces the writing of the first byte of the UART boot stream. Use this option when the bit rate is high.
-h	Command-line Help. Outputs the list of command-line switches to standard output and exits.
-host3bytes	(HOST boot only, when width is 8) Uses three bytes to represent 24-bit data. The default is four bytes with one byte of zero padding.
-l <i>file</i>	Bypasses the BOOT ROM kernel. The first word of the boot stream contains a predetermined format. On seeing this, the BOOT ROM kernel proceeds to set up a DMA to transfer 256 words, corresponding to the loader specified in the command line, into the first 256 locations of internal memory (the location from which the DSP boots) and then performs a JUMP to location 0. At this point, it relinquishes control over the booting process.
-o <i>filename</i>	Specifies the name of the output file. If no file name is specified, the output file takes the name of the executable file. The extension is .LDR.
-opmode #	Specifies whether the boot kernel sets the DSP to 3-SPORT mode (0) or 2-SPORT/1-SPI mode (1) at the end of booting. Default is 0.
-p <i>address</i>	Specifies a hexadecimal integer as the PROM starting address
-v	Verbose loader messages. Outputs status information as the loader processes files.
-waits #	(EPROM and HOST boot modes only) Determines the number of wait states for external accesses. Valid inputs are 0 to 7 (inclusive). Default is 7.
-width #	External bus width. Specifies the bit width for EPROM/FLASH or HOST. Choices are 8 (default) or 16. Width must correspond to the EMICTL register's E_BWS bit.
-M	Shows dependencies only.

Table 5-6. elfloader.exe Command-Line Items¹ (Cont'd)

Switch	Description
-MM	Shows dependencies while processing the input files and producing an output file.
-Mo <i>filename</i>	Places dependency information in the output file specified by <i>filename</i> . This switch must be used in conjunction with -M or -MM.
-Mt <i>targetname</i>	Uses <i>targetname</i> in the dependency file. This switch must be used in conjunction with -M or -MM.
-readall	Creates a non-bootable image and non-boot stream image in the same output file (together with the boot-loadable image). Boot mode must be set to PROM (-b PROM) and format must be set to hexadecimal (-f HEX).
-romsplitter	Creates a non-bootable image only. This switch overwrites -b boot_type and any other switch bounded by boot types. An ASCII file is produced when -f ASCII and -romsplitter are specified, regardless of the -b file_type setting.
-maskaddr #	Masks all EPROM address bits above or equal to #. For example, -maskaddr 18 masks all the bits above and including A18 (ANDed by 0x3FFFF). This switch does not require -romsplitter and affects ROM the section address only.
-split # [#]	(valid only when width is 16) -split 8 8 generates two output files for the 16-bit wide EMI data bus; the .LDU file contains the upper eight data bits, and the .LDL file contains the lower eight data bits. -split 16 produces one 16-bit wide file.

- ¹ Switches are optional. Items shown in *italics* are user-definable and are described with each switch.

6 ADSP-2192-12 LOADER

Use the loader utility (`elfloader.exe`) to convert executable files (.EXE) into a boot-loadable file (.H) for an ADSP-2192-12 DSP. From the loader command line (or the **Load** tab of the **Project Options** dialog box within the VisualDSP++ environment), set options for the type of boot-loadable file that the loader produces.

 You cannot produce a non-bootable PROM image file (that is, splitting is not permitted) for an ADSP-2192-12 DSP.

This chapter contains the following information:

- “ADSP-2192-12 Boot Loader Utility and Run-Time Boot Loader” on page 6-2 contains reference information on the loader commands and operations
- “ADSP-2192 Boot Loader Utility Command-Line Reference” on page 6-10

ADSP-2192-12 Boot Loader Utility and Run-Time Boot Loader

An ADSP-2192-12 DSP can be boot-loaded through its PCI or USB interface. For PCI loading, the loadable image (.EXE) must reside in the PC host's memory space before it is loaded into the DSP.

VisualDSP++ includes a boot loader (`elfloader.exe`) utility that repackages .DXE files (and associated .OVL and .SM files) into an .H file for use with a run-time boot loader (RTBL). The RTBL is implemented by the user, but VisualDSP++ includes a reference RTBL that you can use to boot-load to an ADSP-2192-12 EZ-KIT Lite evaluation system on Windows 98 and Windows 2000 platforms.

This section contains the following information:

- [“Boot Loader Utility \(BLU\)” on page 6-3](#)
- [“Building the .DXE Files” on page 6-5](#)
- [“Running the Run-Time Boot Loader and RTBL” on page 6-6](#)
- [“Run-Time Boot Loader and Overlays Over PCI” on page 6-7](#)
- [“Using OvlPciAdrTbl and OvlMgrTbl Symbols” on page 6-8](#)

Refer to the *ADSP-2192 DSP Hardware Reference and Application Note EE-124* for further details.

Boot Loader Utility (BLU)

The VisualDSP++ linker produces files with `.DXE`, `.OVL`, and `.SM` extensions. These ELF-format files are usually used by the debugger.

Typically, these files are not shipped with your final end product. Instead, the linker's output is run through the boot loader utility (BLU) that repackages the output as an `.H` file, which is consumed by an RTBL. The RTBL output (`.EXE` file) is used by a bootable device (the PCI or USB interface) in the case of the ADSP-2192-12 DSP. Refer to [Figure 6-1 on page 6-3](#).

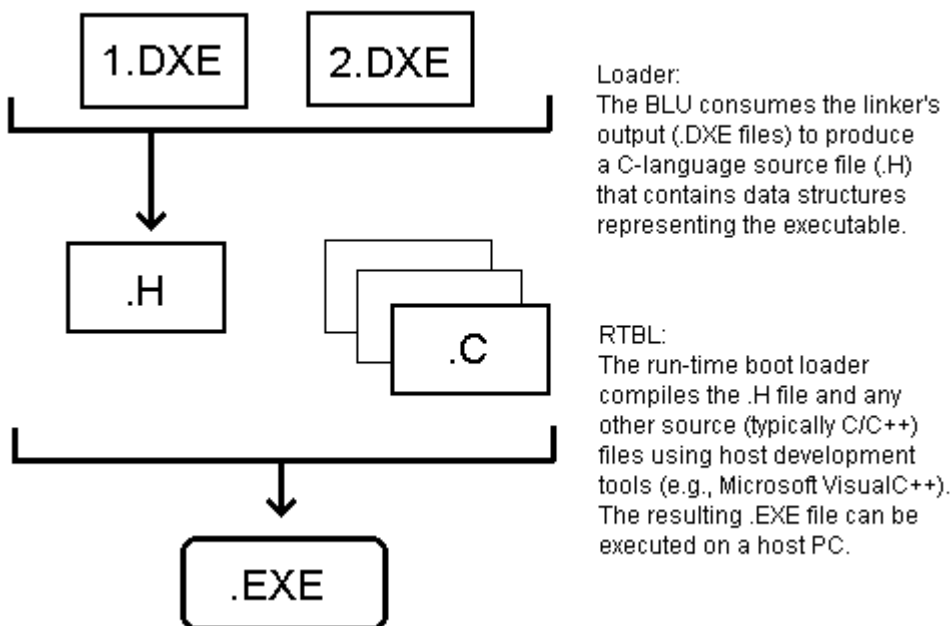


Figure 6-1. Boot Loader Utility Usage

Given that booting from PCI or USB involves code running on the PC host, you must create a program that executes on the host to initiate and conduct the actual PCI or USB transfer. Because of this, the BLU output (.H file) is the C language source code for inclusion and compilation into a host program (.EXE) using a C compiler (such as Microsoft Visual C++) to create the RTBL (see [“Running the Run-Time Boot Loader and RTBL” on page 6-6](#)).

The source code emitted by the BLU is essentially arrays and structures representing the executable's sections and their contents. Refer to the comments contained within the generated .H file for specific information about the data structures and their usage.

 When a DSP executable image (.DXE) changes, rerun the BLU (this creates a new .H file from the .DXE, .OVL, and .SM input files) and then run the RTBL (this builds an .EXE file from the .H file). Automate these tasks from the VisualDSP++ environment by specifying the target type as **Loader file** on the **Project** page of the **Project Options** dialog box and running the RTBL with a post-build command (**Post Build** page of the **Project Options** dialog box); this invokes a makefile that builds the RTBL.

Building the .DXE Files

You must build the two .DXE files which serve as input to the boot loader utility (BLU). Use the VisualDSP++ environment or a command line.

To build the .DXE files from VisualDSP++

1. Open the **Project** page of the **Project Options** dialog box.
2. Under **Processor**, select **ADSP-2192-12**.
3. Under **Type**, select **Loader file**.
4. Under **Name**, type a name for the DSP's core 0 .DXE file.
5. From **Link** page, configure linker options for the core 0 file.
6. Run the project. This generates the .DXE, .OVL, and .SM files.
7. From **Project** page, under **Name**, type a name for the DSP's core 1 .DXE file.
8. From **Link** page, configure linker options for the core 1 file.
9. Run the project. This generates the .DXE, .OVL, and .SM files.

For more information about the VisualDSP++ IDDE, see the *VisualDSP++ 3.0 User's Guide for ADSP-21xx DSPs* or online Help.



You first build the two .DXE files. Then you build the .H file. Lastly, you create the .EXE file. The steps in the above procedure and the following section suggest one method to build the .EXE file. You may choose to combine steps.

Running the Run-Time Boot Loader and RTBL

The boot loader utility (`elfloader.exe`) generates a single output file with an `.H` extension. As described above, a host compiler is used in conjunction with the BLU output (`.H`) and other user-written code (`.C` or `.CPP`) to create a host executable (`.EXE`).

Loader (BLU) options control how the loader processes executable files. Specify options via loader command-line switches. Settings on the **Load** page of the **Project Options** dialog box correspond to command-line switches.

The `.EXE`, which is executed on the end-user system, traverses the data structures (`.H`) created by the BLU (and subsequently compiled by the RTBL). The content of these data structures are downloaded to the ADSP-2192-12 DSP through a user-provided Windows driver.

 A program running in virtual memory space (as do all Windows applications) cannot access the PCI-mapped memory of the ADSP-2192-12 DSP directly. A driver is mandatory.

To create an `.EXE` file from VisualDSP++

1. From **Load** page of the **Project Options** dialog box, specify the input (`.DXE`) files under **Core 0** and **Core 1**.
2. From **Post Build** page, configure one or more command lines to execute the RTBL.
3. Run the project. This generates the `.H` file and creates the `.EXE` file.

Reference RTBL

Creating the RTBL and driver can be a complex task, especially the first time you undertake such an endeavor. To assist you, VisualDSP++ includes a reference RTBL in the form of a Microsoft VisualC++ 6.0

project named `reference_rtbl.dsp`. This project is located in the `...\VisualDSP++\219x\ldr` subdirectory. Refer to the files in this project for information on the operation of the RTBL and the provided driver interface.

The RTBL downloads the code via the PCI bus to the ADSP-2192-12 EZ-KIT Lite evaluation system through the EZ-KIT Lite's PCI driver. This reference can serve as an example for traversing and using the data structures emitted by the BLU. Then download those structures to the target through a driver.



The reference RTBL provided with the EZ-KIT Lite evaluation system does not support USB booting.



The EZ-KIT Lite evaluation system driver can be reused for targets other than Analog Devices EZ-KIT Lite evaluation systems, but this extremely simple driver may be inadequate for anything other than prototyping. Furthermore, the EZ-KIT Lite driver can be reused only on so-called “plug-and-play” Windows operating systems like Windows 98 and Windows 2000.

Run-Time Boot Loader and Overlays Over PCI

Using overlays in an executable can greatly complicate the use of the RTBL and PCI driver for two main reasons:

- The overlay's “live space” is on the PC host and not the DSP. As a result, the linker is not aware of these addresses at build time, so the PCI driver must patch executables as they are downloaded to the DSP to insert the correct addresses at runtime.
- The DSP cannot access the Windows virtual memory space. An overlay must be copied from virtual memory space to PCI memory space, which can only be allocated in limited quantities by the PCI driver.

However, the BLU output considers the need for run-time patches of the executable image. A portion of the data structure created by the BLU is for the express purpose of enabling user code in the RTBL to set up these addresses at runtime. Refer to the reference RTBL for an example of how this is done.

Due to the DSP's inability to access Windows virtual memory space, the RTBL and driver must be coordinated to make overlays available in PCI memory space.

Typically, information (overlay's image and size) is sent to the driver. The driver then `malloc()`'s a memory buffer equal in size to the overlay and then copies the overlay to this space, thus making the overlay available in the PCI memory space.

Refer to the reference RTBL for more information.

Using OvIPciAdrTbl and OvMgrTbl Symbols

The boot loader utility, when running, searches for these two symbols: `OvIPciAdrTbl` and `OvMgrTbl`. If the loader cannot resolve the two symbols, it sets both values to zero. You must declare one or both symbols in the assembly source file to make the run-time boot loader handle the overlay properly.

`OvIPciAdrTbl` holds the start address of the overlay live address table. The loader gets this symbol's value from the input `.DXX` file and places it in the "offset" of the first `relocation_type` array. The value of the "offset" of the second `relocation_type` array is then the value of the first `relocation_type` "offset" plus 2.

Consequently, each subsequent "offset" gets a value equal to the previous "offset" plus 2. The run-time boot loader determines the exact the overlay live address of each overlay according to the provided "offset" value.

Instead of defining `OvlPciAdrTbl`, you can instead define `OvlMgrTbl` in the assembly source code. This symbol should contain the start address of the overlay table, and the overlay table can be declared and defined in your assembly source code. The boot loader gets this symbol's value and makes the `#define` statement along with the other `#define` statement. In a simple case, they look like:

```
#define CORE0_OVL_MGR_TBL      0  
#define CORE0_OVL_COUNT      3
```

The first `#define` statement provides the start address of the overlay table. This is zero when the boot loader fails to find the symbol in the input `.DXE` file. Correct the value by manually changing the value, or by defining it in the assembly source code and re-building the boot loader file.

The second `#define` statement provides the number of overlay for core 0. The run-time boot loader later uses the provided overlay table start address to find the entry for the live start address each overlay and changes it before loading the overlay table into DSP memory.

ADSP-2192 Boot Loader Utility Command-Line Reference

The boot loader utility (`elfloader.exe`) generates a loader file for ADSP-2192-12 DSPs by processing data from one or two executables (`.DXE`) along with overlay (`.OVL`) and shared memory (`.SM`) files generated by the linker. The boot loader utility's output file is a C-language header file with an `.H` extension.

Before running the boot loader utility (BLU), ensure that all overlay (`.OVL`) and shared memory (`.SM`) files reside in the same working directory as the executables. The boot loader utility automatically opens the overlay (`.OVL`) and shared memory (`.SM`) files and processes the data from them while processing the executables (`.DXE`).

This section provides reference information on the boot loader utility command line. When you run the boot loader utility, switches may be placed in any order. A list of all switches and a description of each switch appears in [“ADSP-2192 Boot Loader Utility Command-Line Reference” on page 6-10](#).

Single Processor

Use the following syntax for the boot loader utility command line when there is only one executable (`.DXE`).

```
elfloader -core0 sourcefile -proc ADSP-2192 -switch [-switch...]
```

or

```
elfloader -core1 sourcefile -proc ADSP-2192 -switch [-switch...]
```

where:

- `-core0` — Specifies to the boot loader utility that the sourcefile is for core 0. The boot loader utility makes up the array and structure names according to this core number supplied in the output file.
- `-core1` — Specifies to the boot loader utility that the sourcefile is for core 1. The boot loader utility makes up the array and structure names according to this core number supplied in the output file.
- `sourcefile` — Identifies the input executable file (`.DXE`) to be processed for a single processor. A file name can include the drive, directory, file name and file name extension; enclose long file name within straight-quotes “long file name”.
- `-proc ADSP-2192` — This mandatory switch informs the boot loader to produce an output file for ADSP-2192-12 processor. By default, the output file is a C-language header file.
- `-switches` — Option(s) to be passed to the loader.

Two Processors

Use the following syntax for the boot loader utility command line when there are two input executables (`.DXE`):

```
elfloader -core0 sourcefile0 -core1 sourcefile1 -proc ADSP-2192  
-switch [-switch...]
```

ADSP-2192 Boot Loader Utility Command-Line Reference

where:

- `-core0 sourcefile0` — Informs the boot loader utility that `sourcefile0` is the input file to be processed for core 0. The boot loader utility creates the array and structure names according to the supplied core number.
- `-core1 sourcefile1` — Informs the boot loader utility that `sourcefile1` is the input file to be processed for core 1. The boot loader utility creates the array and structure names according to the supplied core number.
- `-proc ADSP-2192` — This mandatory switch produces an output file for ADSP-2192-12 processor. By default, the output file is a C-language header file.
- `-switches` — Option(s) to be passed to the loader.

Example

```
elfloader -proc ADSP-2192 -core0 p0.dxe -core1 p1.dxe
```

This command line runs the boot loader utility with:

- `-proc ADSP-2192` — Informs the boot loader utility to produce an output file for a ADSP-2192-12 processor.
- `-core0 p0.dxe` — Specifies the input file (`p0.dxe`) to be processed for core 0.
- `-core1 p1.dxe` — Specifies the input file (`p1.dxe`) to be processed for core 1.
- By default, since no output file is specified, the default output file is named `p0.h`.

File Searches

File searches are important in the boot loader utility process. The boot loader utility supports relative and absolute directory names, default directories, and user-selectable directories for search paths.

File searches occur as follows:

- *Specified path* — If you include relative or absolute path information in a file name, the loader searches only in that location.
- *Default directory* — If you do not include path information in the file name, the loader searches for the file in the current working directory.

When you provide an input or output file name as a command-line parameter, use the following guidelines:

- Enclose long file names within straight-quotes, “long file name”.
- Append the appropriate file name extension to each file.

File Names

The boot loader utility follows the conventions for file name extensions that appear in [“File Name Extensions” on page 6-13](#). Descriptions for each switch appear in [“ADSP-2192 elfloader.exe Command-Line Items” on page 6-14](#).

Table 6-1. File Name Extensions

Extension	File Description
.DxE ¹	Executable files and boot-kernel files
.H	Loader output file for a C header file

- ¹ The loader recognizes shared memory (.SM) and overlay memory (.OVL) files, but does not expect these files on the command line. Place .SM and .OVL files in the same directory as the .DxE that refers to them, so the boot loader utility can find them when processing the .DxE file.

Command Line

The switches in “[ADSP-2192 elfloader.exe Command-Line Items](#)” on [page 6-14](#) may be used in any order on the command line.

Table 6-2. ADSP-2192 elfloader.exe Command-Line Items¹

Switch	Description
<code>-core0 <i>file</i></code>	The name of the input executable file processed for the core 0.
<code>-core1 <i>file</i></code>	The name of the input executable file processed for the core 0
<code>-proc ADSP-2192</code> or <code>-dADSP2192</code>	Produces an output file for ADSP-2192-12 processors. Note that <code>-proc ADSP-2192</code> is the preferred switch.
<code>[-f <i>format</i>]</code>	Boot file format. Prepares an output file in the specified format. Currently, the boot loader utility supports the C style header file only. This is the default.
<code>-h</code>	Command line Help. Outputs the list of command-line switches to standard output and exits.
<code>-o <i>file</i></code>	Specifies the name of the output file.
<code>-v</code>	Shows process information while processing input files.
<code>-M</code>	Shows dependencies only
<code>-MM</code>	Shows dependencies while processing the input files and producing an output file.
<code>-Mo <i>filename</i></code>	Places dependency information in the output file specified by <i>filename</i> . This switch must be used in conjunction with <code>-M</code> or <code>-MM</code> .
<code>-Mt <i>targetname</i></code>	Uses <i>targetname</i> in the dependency file. This switch must be used in conjunction with <code>-M</code> and <code>-MM</code> .

- ¹ Switches may occur in any order on the command line. Items shown in *italics* are user-definable and are described with each switch.

7 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.

You can run the archiver from a command line or produce an archive file as the output of a VisualDSP++ project.

This chapter provides the following archiver information:

- “[Archiver Guide](#)” on [page 7-2](#) introduces the archiver’s functions
- “[Archiver Command-Line Reference](#)” on [page 7-7](#) reference information on archiver operations

Archiver Guide

`elfar.exe` combines and indexes object files (or any other files), producing a searchable library file. It performs the following operations, as directed by options on the `elfar` command line:

- Creates a library file from a list of object files
- Appends one or more object files to an existing library file
- Deletes file(s) from a library file
- Extracts file(s) from a library file
- Prints the contents of a specified object file of an existing library file to `stdout`
- Replaces file(s) in an existing library file
- Encrypt symbols in an existing library file

The archiver can run only one of these operations at a time. However, for commands that can take a list of file names as arguments, it can input a text file that contains the names of object files (separated by white space) which makes long lists easily manageable.

The archiver, which is sometimes called a librarian, is general-purpose. It can combine and extract arbitrary files. This manual refers to DSP object files (.DOJ) because they are relevant to DSP code development.

Creating a Library From VisualDSP++

Within the VisualDSP++ IDDE, 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, you must run the archiver from the elfar command line.

File Name Conventions

To maintain code consistency, use the conventions in [Table 7-1](#). Note that VisualDSP++ writes `<projectname>.DLB` when it creates a library.

Table 7-1. File Name Extensions

Extension	File Description
.DLB	Library file
.DOJ	Object file. Input to archiver.
.TXT	Text file used as input with the <code>-i</code> switch

Making Archived Functions Usable

In order to use the archiver effectively, you must know how to write archive files which make your DSP functions available to your code (via the linker), and how to write code that accesses these archives.

Archive usage consists of two tasks:

- Writing *archive routines*, functions that can be called from other programs
- Accessing archive routines from your code

Writing Archive Routines: Creating Entry Points

An archive routine is a routine (or function) in code that can be accessed by other programs. Each routine must have a globally visible start label (*entry point*). Code that accesses that routine must declare the entry point's name as an external variable in the calling code.

To create entry points:

1. Declare the start label of each routine as a global symbol with the assembler's `.GLOBAL` directive. This defines the entry point.

The following code fragment has two entry points, `dIriir` and `FAE`.

```
...
.global dIriir;
.var/dm/seg=seg_data1 FAE[2];
.init FAE[2]: 0x1234, 0x4321;
...
.global FAE;
dIriir: CNTR=N-2;
    I2 = ^FAE;
...
```

2. Assemble and archive the code containing the routines. You can do so in two ways.
 - Direct the VisualDSP++ IDDE to produce an library (see [“on page 7-3”](#)) When you build the project, the object code containing the entry points is packaged in `<project-name>.DLB`. You can extract an object file (`.DOJ`), for example, to incorporate it in another project.
 - If you create executable or unlinked object files from the IDDE, you can 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 you link the program, you 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 need 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 Linker Description File (for example, `ADSP-218x.ldf`), and creates the executable file (`main.dxe`).

```
linker -DADSP-218x main.doj sum.doj filter.dlb -o main.dxe
```

Assuming that one or more library routines from `filter.dlb` are called from one or more of the object files, the linker searches the library, extracts the required routines, and links the routines into the executable.

Archiver File Searches

File searches are important in the archiver's process. The archiver supports relative and absolute directory names, default directories, and user-specified directories for file search paths. File searches include:

- *Specified path* — If you include relative or absolute path information in a file name, the archiver searches only in that location for the file.
- *Default directory* — If you do not include path information in the file name, the archiver searches for the file in the current working directory.

Archiver Command-Line Reference

The archiver processes object files into a library file with a `.DLB` extension, which is the default extension for library files. It can also append, delete, extract, or replace member files in a library, as well as list them to `stdout`. This section provides reference information on the archiver command line and linking.

elfar Command Syntax

Use the following syntax to run `elfar` from the command line. [Table 7-2 on page 7-8](#) describes each switch.

```
elfar [-a|c|d|e|p|r] [-v] [-M] [-MM] [-i filename]
      lib_file obj_file [obj_file ...]
```

When using symbol encryption, use the following syntax. Refer to [“Archiver Symbol Encryption Reference” on page 7-10](#) for details.

```
elfar -s [-v] out-archive in-archive exclude-file type-letter
```

Example elfar Command

```
elfar -v -c my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

The above 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 7-1 on page 7-3](#) lists typical file types, file names, and extensions.

Archiver Command-Line Reference

Parameters and Switches

[Table 7-2](#) describes each archiver part of the command. Switches must appear before the name of the archive file.

Table 7-2. elfar.exe Command-Line Items

Item	Description
<i>lib_file</i>	Specifies the library that the archiver modifies. This parameter appears after the switch.
<i>obj_file</i>	Identifies one or more object files that the archiver uses when modifying the library. This parameter must appear after <i>lib_file</i> . Use the <i>-i</i> switch to input a list of object files.
<i>-a</i>	Appends one or more <i>object files</i> to the end of the specified library file.
<i>-c</i>	Creates a new <i>lib_file</i> containing the listed <i>obj_file(s)</i> .
<i>-d</i>	Removes the listed <i>obj_file(s)</i> from the specified <i>lib_file</i> .
<i>-e</i>	Extracts the specified file(s) from the library.
<i>-i filename</i>	Uses <i>filename</i> , a list of object files, as input. This file lists <i>obj_file(s)</i> to add or modify in the specified <i>lib_file</i> (.DLB).
<i>-M</i>	(available only with the <i>-c</i> switch) Prints dependencies.
<i>-MM</i>	(available only with the <i>-c</i> switch) Prints dependencies and creates the library.
<i>-p</i>	Prints a list of the <i>obj_file(s)</i> (.DOJ) in the selected <i>lib_file</i> (.DLB) to standard output.
<i>-r</i>	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.
<i>-s</i>	Specifies symbol name encryption. Refer to “Archiver Symbol Encryption Reference” on page 7-10 .
<i>-v</i>	Verbose. Outputs status information as the archiver processes files.

Constraints

This command is subject to the following constraints:

- You may select one action switch (a, c, d, e, p, r, s, M, or MM) only in a single command.
- The verbose operation switch, -v, must not be placed in a position where it can be mistaken for an object file. It may not follow the *lib_file* on an append or create operation.
- The file include switch, -i, must immediately precede the name of the file to be included.

The archiver's -i switch lets you input a list of members from a text file, instead of listing each member on the command line.

- Use the library file name first, following the switches. -i and -v are not operational switches, and can appear later.
- When you use 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.
- Wildcards are not permitted. To perform an archive operation on a list of member files, write the list in a text file and use the text file as input to the command line with the -i switch.
- *obj_file* — The name of an object file (.OBJ) to be added, removed, or replaced in the *lib_file*.
- The archiver's command line is *not* case-sensitive.

Archiver Symbol Encryption Reference

Symbol name encryption protects intellectual property contained in an archive library (.DLB) that might be revealed by the use of meaningful symbol names. Code and test a library with meaningful symbol names, and then use archive library encryption on the fully tested library to disguise the names.

 Source file names in the symbol tables of object files in the archive are not encrypted. The encryption algorithm is not reversible. Also, encryption does not guarantee a given symbol will be encrypted the same way when different libraries, or different builds of the same library, are encrypted.

Command Syntax

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

```
elfar -s [-v] out-archive in-archive exclude-file type-letter
```

where:

-s — Selects the encryption operation

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

out-archive — Specifies the name of the library file (.DLB) to be produced by the encryption process

in-archive — Specifies the name of the archive file (.DLB) to be encrypted. This file is not altered by the encryption process, unless in-archive is the same as out-archive.

`exclude-file` — Specifies the name of a text file containing a list of symbols not to be encrypted. The symbols are listed one or more to a line, separated by white space.

`type-letter` — The initial letter of `type-letter` provides the initial letter of all encrypted symbols.

Constraints

All local symbols can be encrypted, unless they are correlated (see below) with a symbol having external binding that should not be encrypted. Symbols with external binding can be encrypted when they are used only within the library in which they are defined. Symbols with external binding that are not defined in the library (or are defined in the library and referred to outside of the library) should not be encrypted. Symbols that should not be encrypted must be placed in a text file, and the name of that file given as the `exclude-file` command line argument.

Some symbol names have a prefix or suffix that has special meaning. The debugger does not show a symbol starting with “.” (period), and a symbol starting with “.” and ending with “.end” is correlated with another symbol. For example, “.bar” will not be shown by the debugger, and “._foo.end” is correlated with the symbol “_foo” appearing in the same object file. The encryption process encrypts only the part of the symbol after any initial “.”, and before any final “.end”. This part is called the root of the symbol name. Since only the root is encrypted, a name with a prefix or suffix having special meaning retains that special meaning after encryption.

The encryption process ensures that a symbol with external binding will be encrypted the same way in all object files contained in the library. It also ensures that correlated symbols (see explanation above) within an object file will be encrypted the same way, so they remain correlated.

The names listed in the `exclude-file` are interpreted as root names. Thus, “_foo” in the `exclude-file` prevents the encryption of the symbol names “_foo”, “._foo”, “_foo.end”, and “._foo.end”.

Archiver Command-Line Reference

The `type-letter` argument, which provides the first letter of the encrypted part of a symbol name, ensures that the encrypted names in different archive libraries can be made distinct. If different libraries are encrypted with the same `type-letter` argument, there is a possibility that unrelated external symbols of the same length will be encrypted identically.

A FILE FORMATS

The development tools support many file formats, in some cases several for each development tool. This appendix describes file formats that you prepare as input for the tools and points out the features of files produced by the tools.

This appendix describes these three types of file formats:

- [“Source Files” on page A-2](#)
- [“Build Files” on page A-6](#)
- [“Debugger Files” on page A-13](#)

Most of the development tools use industry standard file formats. Sources that describe these formats appear in [“Format References” on page A-14](#).

Source Files

This section describes the following types of 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 such extensions as .C, .CPP, .CXX, and so on) containing C/C++ code, compiler directives, possibly a mixture of assembler code and directives, and (typically) preprocessor commands.

Several “dialects” of C code are supported: pure (portable) ANSI C, and at least two subtypes¹ of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives, used by the linker to structure and place executable files.

For information on using the C/C++ compiler and associated tools, as well as a definition of ADI extensions to ANSI C, see the *VisualDSP++ 3.0 C/C++ Compiler & Library Manual for Blackfin DSPs*.

¹ With and without built-in function support; a minimal differentiator. There are others.

For information on specifying the C dialect and general C code handling within VisualDSP++, see the *VisualDSP++ 3.0 User's Guide for Blackfin DSPs* and VisualDSP++ online Help.

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 the *Instruction Set Reference* manual for the corresponding DSP.

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++ Assembler and Preprocessor Manual for Blackfin DSPs*.

Assembly Initialization Data Files (.DAT)

These are text files containing fixed-point data. These files can provide the initialization data for an assembler `.VAR` directive or serve in other tool operations. When a `.VAR` directive uses a `.DAT` file for initialization data, 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 merely explanatory or mnemonic. A directive to `#include <file>` can of course 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](#) for identifying a fixed-point value's numeric base.

Table A-1. Fixed-Point Values in Data Files

Convention	Description
<i>0xnumber</i> , <i>Hnumber</i> <i>hnumber</i>	An “0x”, “H#” or “h#” prefix indicates a hexadecimal <i>number</i>
<i>number</i> , <i>Dnumber</i> <i>dnumber</i>	A “#D”, “#d”, or no prefix indicates a decimal number
<i>Onumber</i>	An “#O” or “#o” prefix indicates an octal number
<i>Bnumber</i> <i>bnumber</i>	A “B#” or “b#” prefix indicates a binary <i>number</i>

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 into the 8-bit memory location and zero-fills the lower eight bits.

Header Files (.H)

Header files are text files that contain macros or other preprocessor commands that the preprocessor substitutes into source files. For information on macros or other preprocessor commands, see the VisualDSP++ 3.0 *Assembler and Preprocessor Manual for Blackfin DSP*.

Linker Description Files (.LDF)

The linker’s .LDF files are ASCII text files that contain commands for the linker in the linker’s scripting language. For information on this scripting language see [“LDF Guide” on page 2-2](#).

Linker Command-Line Files (.TXT)

The linker's command-line files are ASCII text files that contain command-line input for the linker. For more information on the linker's command line, see [“Linker Command-Line Reference” on page 1-28](#).

Build Files

Build files are files that the development tools produce when building your VisualDSP++ project. This section describes the following types of build file formats:

- [“Assembler Object Files \(.DOJ\)” on page A-6](#)
- [“Library Files \(.DLB\)” on page A-6](#)
- [“Linker Executable Files \(.DXE, .SM, and .OVL\)” on page A-7](#)
- [“Linker Memory Map Files \(.MAP\)” on page A-7](#)

Assembler Object Files (.DOJ)

The assembler’s output object files are binary, Executable-Linkable-File (ELF) format. Object files contain relocatable code and debugging information for your program’s segments. The linker processes your object files into an executable file. For information on the ELF format used for object files, see the [“Format References” on page A-14](#).

Library Files (.DLB)

Library files, the archiver’s output, are in binary, Executable-Linkable-File (ELF) format. Library files (called archive files in previous software releases) contain one or more object files, called archive elements.

The linker searches through library files for library members used by your code. For information on the ELF format used for executable files, see the [“Format References” on page A-14](#).

Linker Executable Files (.DXE, .SM, and .OVL)

The linker's output executable files are binary, Executable-Linkable-File (ELF) format. Executable files contain your program's code and debugging information. The linker may fully or partially resolve addresses in executable files, depending on the options given on the linker's command line and in your linker description file. For information on the ELF format used for executable files, see the TIS Committee texts cited in ["Format References" on page A-14](#).

Linker Memory Map Files (.MAP)

The linker's memory map files are ASCII text files that contain memory and symbol information for your executable file(s). The map contains a summary of memory that you define with `MEMORY{ }` commands in your linker description file, and provides a listing of the absolute addresses of all symbols.

Loader Hex Format Files (.LDR, .BNM)

The loader's output Hex-format files are in ASCII, Intel Hex-32 format. Hex files from the loader support 8-bit wide PROMs. The files are used with an industry-standard PROM programmer to program memory devices for your 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 either a data record, an extended linear address record or the end of file record:

:020000040010E9	extended linear address record
:0A0004003C40343434261422260850	data record
:00000401FB	end of file record

Build Files

The extended linear address record is used because a data record has only a 4-character (16-bit) address field, but ADSP-21xx processors require up to 24 bits of word-address address space, resulting in 26 bits of byte-address space. The extended linear address record specifies address bits 16-31 for the data records that follow it. Data records are organized into the following fields:

:0A0004003C40343434261422260850	example record
:	start character
0A	byte count of this record
0004	address of first data byte
00	record type
3C	first data byte
08	last data byte
50	checksum

Extended linear address records have the following fields:

:02000000340850	
:	start character
02	byte count (always 02)
0000	address (always 0000)
00	record type
3408	offset address
50	checksum

The end of file record looks like this:

:00000001FF

:	start character
00	byte count (zero for this record)
0000	address of first byte
01	record type
FF	checksum

IDMA Host Boot File (.IDM)

The IDMA boot file is an ASCII file that contains one 16-bit hex value per line. Data is organized segment-by-segment, and every segment is headed by the 16-bit word count and the IDMA control value to be latched into the IDMA Control register. If the `elfspl21.exe` utility is invoked with the `-218x` switch, a third word specifies the overlay page.

 Unlike former releases, VisualDSP++ 3.0 pre-formats this word to be ready for the IDMA Overlay register.

Bit 15 is always set, and DM overlay information is shifted accordingly. 24-bit PM data is split into two 16-bit words as required by the 16-bit IDMA port. The stream ends with a `0xFFFF` word.

Example

This example illustrates an `.IDM` file consisting of a single segment containing one 24-bit word (`0x123456`) to be mapped to PM overlay page 4 at address `0x2000`.

```
0002 // segments contains 2 16-bit values
2000 // IDMA Control, IDMA0=0, IDMA1=0x2000
8004 // IDMA Overlay, PM Page 4
1234 // MSB of 24-bit data
0056 // LSB of 24-bit data, right aligned
FFFF // end of boot stream marker
```

IDMA Unit Operation

The `-idma` switch causes `elfspl21` to produce a series of records suitable for programming the ADSP-218x DSP's IDMA unit.

 The `-idma` switch implies the `-2181` switch and overrides the `-loader` switch.

The .IDM file, an ASCII text file, consists of:

- <count> (number of 16-bit words in the record; same as the number of DM words, twice the number of PM words)
 - <IDMA Control Register value>
 - <overlay page> (optional, only when invoked with -218x instead of -2181)
- <16 bit words> (PM have 16 MSBs first, then 8 LSBs, right-justified).
- The final record programs the restart vector and is followed by the word “ffff”, which serves as an end marker since it occurs in the position of a count word, but the maximum count for an ADSP-21xx part would be 4000 (hex). For example,

```
2002
4000
0000
FABC
:
FFFF
```

ASCII Byte-Stream Format

Byte-stream files may be created for program and data memory, for vertical rather than horizontal word organization. This format applies to .BNM files produced when using a -us, -us2, and -ui command-line switch to the elfsp121.exe utility for ADSP-218x DSPs.

In a byte-stream output file, the most significant byte of each word precedes the less significant byte(s), from lower address to higher address (that is, the high-order byte of each word is located at the lowest address). The 24-bit words of program memory require sequences of three bytes; the 16-bit words of data memory require sequences of two bytes.

Build Files

If you have assigned absolute addresses to memory objects, the information is lost in byte-stream format.

You cannot generate byte-stream files from input which contains discontinuous blocks of memory (non-relocatable segments with unused blocks of memory in between). Program segments must be relocatable or you must place them in contiguous blocks of memory.

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, .DLB). To simulate I/O, the debugger also supports the data file formats (.DAT) from the assembler and the loadable file formats from the loader (.LDR).

The standard hexadecimal format for a SPORT data file is a single integer value per line. Hex numbers do not require the 0x prefix to indicate hexadecimal. A value can have any number of digits, but are read into the SPORT register, as follows:

- The hexadecimal number is converted to binary
- The number of binary bits read in matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zeros values that are shorter than the word size or, conversely, truncates any bits beyond the word size on the MSB end.

Example: a SPORT register is set for 20-bit words and the data file contains hex numbers. The simulator converts these hex numbers to binary, then fills/truncates to match the SPORT word size. In the following table, the number A5A5 is filled and the number 123456 is truncated

Table A-2. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
A5A5A	1010 0101 1010 0101 1010	1010 0101 1010 0101 1010
FFFF1	1111 1111 1111 1111 0001	1111 1111 1111 1111 0001
A5A5	1010 0101 1010 0101	0000 1010 0101 1010 0101
5A5A5	0101 1010 0101 1010 0101	0101 1010 0101 1010 0101

Format References

Table A-2. SPORT Data File Example

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++:

- (1993) Executable and Linkable Format (ELF) V1.1 from the Portable Formats Specification V1.1, Tools Interface Standards Committee {available from <ftp://ftp.intel.com/pub/tis>}
- (1993) Debugging Information Format (DWARF) V1.1 from the Portable Formats Specification V1.1, UNIX International, Inc. {available from <ftp://ftp.intel.com/pub/tis>}

B UTILITIES

The VisualDSP++ development software includes several file conversion utilities which run from a command line only. Some utilities provide support for legacy code, and others are intended for users who prefer to use the command-line version of the tools instead of running the utilities from the IDDE (VisualDSP++ environment).

This appendix describes the following utilities:

- [“ELF File Dumper” on page B-1](#)
- [“AEXE2ELF and ELF2AEXE Converters” on page B-5](#)

ELF File Dumper

The ELF file dumper (`elfdump.exe`) extracts data from ELF format executable files (`.DXE`) and provides a text output file that describes the ELF file’s contents.

The ELF file dumper runs from the following command-line entry:

```
C:\Program Files\Analog Devices\VisualDSP> elfdump
```

Usage: `elfdump {option} {objectfile}`

Table B-1. ELF File Dumper Command-Line Switches

Switch	Description
-fh	Prints the file header
-arsym	Prints the library symbol table
-arall	Prints every library member
-ph	Prints the program header table
-sh	Prints the section header table. This is the default when no options are specified
-notes	Prints note segment(s)
-n <i>name</i>	Prints contents of the named section(s). The <i>name</i> 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 the sections numbered x0 through x1, where x0 and x1 are decimal integers, and x1 defaults to x0 if omitted. Formatting rules as are for -n.
-all	Prints everything. Same as -fh -ph -sh -notes -n '*'
-ost	Omits string table sections
<i>objectfile</i>	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, forcing section contents to be formatted as follows: - a Dumps contents in hex and ASCII, 16 bytes per line. - x Dumps contents in hex, 32 bytes per line. - xN Dumps contents in hex, N bytes per group (default is N=4). - t Dumps contents in hex, N bytes per line, where N is the section's table entry size. If N is not in the range 1-32, 32 is used. - i Prints contents as list of disassembled machine instructions.

Using the Archiver and Dumper for Disassembly

The file utilities become more effective when you combine their capabilities. One application of these utilities is to disassemble a library member, 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.

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 -e libc.dlb > libc.txt
```



This command assumes the current directory is:

```
C:\Program Files\Analog Devices\VisualDSP++\21xx\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.

Using disassembly, you get 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 symbols were stripped during linking, there are no symbols in the dumped file.

 Using disassembly on 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 `elfar` and `elfdump` to extract and view the contents of an overlay library file (`.OVL`).

For example, the following command lists (prints) the contents (library members) of the `CLONE2.OVL` library file.

```
elfar -p CLONE2.OVL
```

Suppose that you want to view one of the library members, `CLONE2.ELF`. You then use the `elfdump` utility to view `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.

AEXE2ELF and ELF2AEXE Converters

The linker supports legacy object files and libraries created by the VisualDSP development tools. The linker recognizes AEXE-formatted object files (.OBJ) and libraries (.O or .LIB), and translates them automatically into ELF format before linking.

If you use multiple object or library files in your project builds or rebuild frequently, consider converting these files first. You can do this using the `aexe2elf` utility. Conversion can potentially save time in your project development cycle.

Converting From AEXE to ELF

Use the following command syntax to perform AEXE-to-ELF file conversion.

```
aexe2elf [-switches] input_filename [output_filename]
```

`input_filename` is the base name of the file to translate, and `output_filename` is the output base file name. If `output_filename` is not specified, the default is the input base file name plus an .ELF extension.

For example, the following command results in the file `App70.dxe`.

```
aexe2elf -x App61 App70
```

The following is the list of switches available for use with the `aexe2elf` command.

Figure B-1. AEXE2ELF Switches

Switch	Description
-o	Translates AEXE object files to ELF (default) object files
-x	Translates AEXE executable files to ELF (default) executable files

AEXE2ELF and ELF2AEXE Converters

Figure B-1. AEXE2ELF Switches (Cont'd)

Switch	Description
-a	Translates AEXE library files to ELF 'ar' library files
-s source_filename or -source source_filename	Specifies the original source file's name
-h or -help	Displays a Help screen showing the list of switches

Converting From ELF to AEXE

Use the following command syntax to perform ELF-to-AEXE file conversion.

```
elf2aexe [-switches] input_filename [output_filename]
```

The `input_filename` is the base name of the files to translate, and `output_filename` is the output base file name. If the `output_filename` is not specified, the default is the input base file name with an `.EXE` extension. There are two EXE output files: `input_base.sym` and `input_base.exe`.

Figure B-2. ELF2AEXE Switches

Switch	Description
-x	Translates ELF executable files to AEXE (default) executable files
-h or -help	Displays the Help screen showing the list of switches
-v or -version	Displays version information only

C LINKER LEGACY SUPPORT

The linker and utilities provide support for code developed with previous versions of the DSP development software tools. For more information on legacy support, see [“Utilities” on page B-1](#).

This appendix describes the differences between current and previous linker releases targeting the ADSP-218x DSPs, and code migration to ADSP-219x DSPs. It also provides migration information to help you make the necessary code changes to use the newer tools for ongoing ADSP-218x family DSP development.

This appendix describes:

- [“Converting from .SYS to .LDF” on page C-1](#)
- [“Legacy Assembly of Sources with Absolute Placement Directives” on page C-22](#)

For more information on updating assembly code, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual* for your target DSP. For more information on updating your C/C++ code, see the *VisualDSP++ 3.0 Compiler and Library Manual* for your target DSP. These manuals are packaged with your software distribution.

Converting from .SYS to .LDF

This section describes how to convert .SYS files to .LDF file syntax. The process is not complicated, but the syntax is noticeably different.

Converting from .SYS to .LDF

Commands in the .LDF file (LDF) define the target system and specify the order in which to process files. Within the LDF, you can specify libraries, search paths, and input files that define the system and resolve variables and labels. You can write procedures for placing input sections within the LDF, as shown in the sequence of vectors in the interrupt table (`sec_inttab`), with a location counter increment following each increment. You can also specify multiple output files (not shown).

The following sections provide examples of .LDF files for the ADSP-218x, both for C and assembly code.

- [“Converting C Code section\(“...” \) to LDF SECTIONS{” on page C-2](#)
- [“Converting Assembler .SEGMENT to LDF SECTIONS{” on page C-12](#)
- [“Converting SYS .SEGMENT to LDF Commands” on page C-20](#)

Converting C Code section(“...”) to LDF SECTIONS{

C code can contain optional input section directives of the form:

```
section(“wave_data”) <data object declarations>
```

Continuing from the above example, the C compiler attaches `.section wave data` to the assembly code produced from this directive.

The input section directives do not need to be specific. `section(pm)` directs the linker to eventually place the results of this code in a memory segment of type PM.

This section provides two code samples:

- [Listing C-1](#), Example ADSP-2181 LDF File for C Code
- [Listing C-3](#), Sample Hardware Overlay LDF File for C Code



If you write C code, allocate memory segments for the stack and heap in the LDF, and enforce stack and heap size limits. See the sample .LDF files packaged with your VisualDSP++ distribution for reference.

Listing C-1. Example ADSP-2181 LDF File for C Code

```
ARCHITECTURE(ADSP-218x)
SEARCH_DIR( $ADI_DSP\218x\lib )
// specific code and data
// $LIBRARIES = ;
// Libraries from the command line are included in
// COMMAND_LINE_OBJECTS.
$OBJECTS = 218x_int_tab.doj , 218x_hdr.doj ,
$COMMAND_LINE_OBJECTS , libio.dlb , libc.dlb , 218x_exit.doj ;
// 2181 has 16K 24-bit words of program RAM
// and 16K 16-bit words of data RAM
MEMORY
{
  seg_inttab { TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24) }
  // hardware interrupts
  seg_code   { TYPE(PM RAM) START(0x00030) END(0x037ff) WIDTH(24) }
  // compiled code
  seg_data2  { TYPE(PM RAM) START(0x03800) END(0x03fff) WIDTH(24) }
  // pm data
  seg_data1  { TYPE(DM RAM) START(0x00000) END(0x02fff) WIDTH(16) }
  // dm data
  seg_heap   { TYPE(DM RAM) START(0x03000) END(0x037ff) WIDTH(16) }
  // dynamic memory
}
```

Converting from .SYS to .LDF

```
seg_stack { TYPE(DM RAM) START(0x03800) END(0x03fdf) WIDTH(16) }
// stack space
}
PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS
    {
        sec_inttab
        {
            INPUT_SECTIONS( $OBJECTS( IVreset ))           // 0x0000
            INPUT_SECTIONS( $OBJECTS( IVirq2 ))             // 0x0004
            INPUT_SECTIONS( $OBJECTS( IVirq11 ))            // 0x0008
            INPUT_SECTIONS( $OBJECTS( IVirq10 ))            // 0x000c
            INPUT_SECTIONS( $OBJECTS( IVsport0xmit ))       // 0x0010
            INPUT_SECTIONS( $OBJECTS( IVsport0recv ))       // 0x0014
            INPUT_SECTIONS( $OBJECTS( IVirqe ))             // 0x0018
            INPUT_SECTIONS( $OBJECTS( IVbdma ))             // 0x001c
            INPUT_SECTIONS( $OBJECTS( IVirq1 ))             // 0x0020
            INPUT_SECTIONS( $OBJECTS( IVirq0 ))             // 0x0024
            INPUT_SECTIONS( $OBJECTS( IVtimer ))            // 0x0028
            INPUT_SECTIONS( $OBJECTS( IVpwrdown ))          // 0x002c
        } >seg_inttab
        sec_code
        {
            INPUT_SECTIONS( $OBJECTS(program) )
        } >seg_code
        sec_data1
        {
            INPUT_SECTIONS( $OBJECTS(data1) )
        } >seg_data1
        sec_data2
        {
```

```

        INPUT_SECTIONS( $OBJECTS(data2) )
    } >seg_data2
    // support for initialization, including C++
    sec_ctor
    {
        INPUT_SECTIONS( $OBJECTS(ctor) )
    } >seg_data1
    // 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(seg_stack) - 1;
    } >seg_stack
    sec_heap
    {
        .heap = .;
        .heap_size = MEMORY_SIZEOF(seg_heap);
        .heap_end = . + MEMORY_SIZEOF(seg_heap) - 1;
    } >seg_heap
}
}

```

Listing C-2. Sample Hardware Overlay LDF File for C Code

```

ARCHITECTURE(ADSP-2189)
SEARCH_DIR($ADI_DSP\218x\lib)
$OBJECTS = 2189_int_tab.doj, 218x_hdr.doj, $COMMAND_LINE_OBJECTS,
libio.dlb, libc.dlb, 218x_exit.doj;

MEMORY{
    // Internal PM Memory Segments

```

Converting from .SYS to .LDF

```
mem_vect_tbl { TYPE(PM RAM) START(0x00000) END(0x0002f)
WIDTH(24) }
    // 48 locations for vector table (12 * 4)
mem_pmcode   { TYPE(PM RAM) START(0x00030) END(0x01fff)
WIDTH(24) }   // 8k segment for internal PM

////////// Declare PM Overlay Segment here ("run" address)
mem_pm_ovly   { TYPE(PM RAM) START(0x02000) END(0x03fff)
WIDTH(24) }   // PM Overlay Segment Declaration for PLIT only

// Internal PM Overlay Segments ("live" addresses)
mem_pmpage0   { TYPE(PM RAM) START(0x02000) END(0x03fff)
WIDTH(24) }   // Internal 8k PM Overlay Segment 0
mem_pmpage4   { TYPE(PM RAM) START(0x42000) END(0x43fff)
WIDTH(24) }   // Internal 8k PM Overlay Segment 4
mem_pmpage5   { TYPE(PM RAM) START(0x52000) END(0x53fff)
WIDTH(24) }   // Internal 8k PM Overlay Segment 5

////////// Declare DM Overlay Segment here ("run" address)
mem_dm_ovly   { TYPE(DM RAM) START(0x00000) END(0x01fff)
WIDTH(16) }   // DM Overlay Segment Declaration for PLIT only

// Internal DM Overlay Segments ("live" addresses)
mem_dmpage0   { TYPE(DM RAM) START(0x00000) END(0x01fff)
WIDTH(16) }   // Internal 8k DM Overlay Segment 0
mem_dmpage4   { TYPE(DM RAM) START(0x40000) END(0x41fff)
WIDTH(16) }   // Internal 8k DM Overlay Segment 4
mem_dmpage5   { TYPE(DM RAM) START(0x50000) END(0x51fff)
WIDTH(16) }   // Internal 8k DM Overlay Segment 5
mem_dmpage6   { TYPE(DM RAM) START(0x60000) END(0x61fff)
WIDTH(16) }   // Internal 8k DM Overlay Segment 6
mem_dmpage7   { TYPE(DM RAM) START(0x70000) END(0x71fff)
WIDTH(16) }   // Internal 8k DM Overlay Segment 7
```

```
// Internal DM Memory Segment
    mem_int_dm      { TYPE(DM RAM) START(0x02000) END(0x02fff)
WIDTH(16) }        // internal (non-overlay) DM segment
    seg_heap       { TYPE(DM RAM) START(0x03000) END(0x037ff)
WIDTH(16) }        // default heap segment declaration (in
non-overlay DM)
    seg_stack      { TYPE(DM RAM) START(0x03800) END(0x03fdf)
WIDTH(16) }        // default stack memory declaration (in
non-overlay DM)

// External PM Overlay Segments ("live" addresses)
    mem_pmpage1    { TYPE(PM RAM) START(0x12000) END(0x13fff)
WIDTH(24) }        // External 8k PM Overlay Segment 1
    mem_pmpage2    { TYPE(PM RAM) START(0x22000) END(0x23fff)
WIDTH(24) }        // External 8k PM Overlay Segment 2

// External DM Overlay Segments ("live" addresses)
    mem_dmpage1    { TYPE(DM RAM) START(0x10000) END(0x11fff)
WIDTH(16) }        // Internal 8k DM Overlay Segment 1
    mem_dmpage2    { TYPE(DM RAM) START(0x20000) END(0x21fff)
WIDTH(16) }        // Internal 8k DM Overlay Segment 2

}

PROCESSOR p0{

    OUTPUT($COMMAND_LINE_OUTPUT_FILE)

    SECTIONS{
        dx_vect_tbl{ // C Runtime Interrupt Vector Declarations
            INPUT_SECTIONS( $OBJECTS( IVreset ))// 0x0000 Reset vector
            INPUT_SECTIONS( $OBJECTS( IVirq2 ))// 0x0004 IRQ2
        }
    }
    interrupt
```

Converting from .SYS to .LDF

```
        INPUT_SECTIONS( $OBJECTS( IVirq11 ))// 0x0008 IRQL1
interrupt
        INPUT_SECTIONS( $OBJECTS( IVirq10 ))// 0x000c IRQLO
interrupt
        INPUT_SECTIONS( $OBJECTS( IVsport0xmit ))// 0x0010 SPORT0
Tx
                                interrupt
        INPUT_SECTIONS( $OBJECTS( IVsport0recv ))// 0x0014 SPORT0
Rx
                                interrupt
        INPUT_SECTIONS( $OBJECTS( IVirqe ))// 0x0018 IRQE
interrupt
        INPUT_SECTIONS( $OBJECTS( IVbdma ))// 0x001c BDMA
interrupt
        INPUT_SECTIONS( $OBJECTS( IVirq1 ))// 0x0020 IRQ1
interrupt
        INPUT_SECTIONS( $OBJECTS( IVirq0 ))// 0x0024 IRQ0
interrupt
        INPUT_SECTIONS( $OBJECTS( IVtimer89 ))// 0x0028 Timer
                                interrupt (internal)
        INPUT_SECTIONS( $OBJECTS( IVpwrdown ))// 0x002c Powerdown
                                interrupt
>mem_vect_tbl

    dxcode{
INPUT_SECTIONS($OBJECTS(program))
>mem_pmcode

    dxint_dm{
INPUT_SECTIONS($OBJECTS(data1))
>mem_int_dm

    // support for initialization
    sec_ctor
```

```
{
    INPUT_SECTIONS( $OBJECTS(ctor) )
} >mem_int_dm

// 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(seg_stack) - 1;
} >seg_stack

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

// DM Overlay "Run" Segment Declarations
dx_e_dmpage // arbitrary label for linker for overlay segments
PAGE_INPUT // define what material goes into the page image
ALGORITHM(ALL_FIT) // "fit" all of the material into
                    this page
PAGE_OUTPUT(DM_PAGE0.OVL) // output file name of overlay
                           segment for linker
INPUT_SECTIONS(DMOVLY0.DOJ(dm_ovly_0)) // specify what
    objects are inputs to this specific overlay region
>mem_dmpage0 // end of declaration for DM page 0

PAGE_INPUT // define what material goes into the page image
```

Converting from .SYS to .LDF

```
ALGORITHM(ALL_FIT)    // "fit" all of the material into
                        this page
PAGE_OUTPUT(DM_PAGE1.OVL)  // output file name of overlay
                        segment for linker
INPUT_SECTIONS(DMOVLY1.DOJ(dm_ovly_1))  // specify what
                        objects are inputs to this specific overlay region
>mem_dmpage1           // end of declaration for DM page 1

PAGE_INPUT{    // define what material goes into the page image
ALGORITHM(ALL_FIT)    // "fit" all of the material into
                        this page
PAGE_OUTPUT(DM_PAGE2.OVL)  // output file name of overlay
                        segment for linker
INPUT_SECTIONS(DMOVLY2.DOJ(dm_ovly_2))  // specify what
                        objects are inputs to this specific overlay region
>mem_dmpage2           // end of declaration for DM page 2

PAGE_INPUT{    // define what material goes into the page image
ALGORITHM(ALL_FIT)    // "fit" all of the material into
                        this page
PAGE_OUTPUT(DM_PAGE4.OVL)  // output file name of overlay
                        segment for linker
INPUT_SECTIONS(DMOVLY4.DOJ(dm_ovly_4))  // specify what
                        objects are inputs to this specific overlay region
>mem_dmpage4           // end of declaration for DM page 4

PAGE_INPUT{    // define what material goes into the page image
ALGORITHM(ALL_FIT)    // "fit" all of the material into
                        this page
PAGE_OUTPUT(DM_PAGE5.OVL)  // output file name of overlay
                        segment for linker
INPUT_SECTIONS(DMOVLY5.DOJ(dm_ovly_5))  // specify what
                        objects are inputs to this specific overlay region
>mem_dmpage5           // end of declaration for DM page 5
```

```
}>mem_dm_ovly          // assign name of "live" address for overlay
region

                        here (DM0x0000-0x1fff)

// PM Overlay "Run" Segment Declarations
dxm_pmpage{
PAGE_INPUT{           // define what material goes into the page image
ALGORITHM(ALL_FIT)      // "fit" all of the material into
                        this page
PAGE_OUTPUT(PM_PAGE0.OVL) // output file name of overlay
                        segment for linker
INPUT_SECTIONS(PMOVLY0.DOJ(pm_ovly_0)) // specify what
                        objects are inputs to this specific overlay region
>mem_pmpage0          // end of declaration for PM page 0

PAGE_INPUT{           // define what material goes into the page image
ALGORITHM(ALL_FIT)      // "fit" all of the material into
                        this page
PAGE_OUTPUT(PM_PAGE4.OVL) // output file name of overlay
                        segment for linker
INPUT_SECTIONS(PMOVLY4.DOJ(pm_ovly_4)) // specify what
                        objects are inputs to this specific overlay region
>mem_pmpage4          // end of declaration for PM page 0

PAGE_INPUT{           // define what material goes into the page image
ALGORITHM(ALL_FIT)      // "fit" all of the material into
                        this page
PAGE_OUTPUT(PM_PAGE5.OVL) // output file name of overlay
                        segment for linker
INPUT_SECTIONS(PMOVLY5.DOJ(pm_ovly_5)) // specify what
                        objects are inputs to this specific overlay region
>mem_pmpage5          // end of declaration for PM page 0
}>mem_pm_ovly          // assign name of "live" address for overlay
                        region here (PM0x2000-0x3fff)
```

Converting from .SYS to .LDF

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

Converting Assembler .SEGMENT to LDF SECTIONS{}

The program placement part of .SEGMENT declarations in a .SYS file corresponded to the .SECTION directives in the assembly code. The assembler's .SEGMENT directive defined the memory type for each data item's placement with the memory type /dm or /pm qualifier.

This section provides two code samples:

- [Listing C-3 on page C-14](#), Sample ADSP-2181 LDF File for Assembly Code
- [Listing C-4 on page C-21](#), Legacy SYSTEM File Example

In VisualDSP++, the INPUT_SECTIONS() of the SECTIONS{} command in an .LDF file uses the .SECTION declarations in each assembly source file to place each data object into an input section. This .SECTION declaration names and defines the section of code that the linker uses to place each output section into memory. The mappings can be many-to-one.

- .SECTION labels in an assembly source (obligatory for each data item) are mapped into corresponding input sections in the assembled object code. Multiple assembly code sections with the same label accumulate in an input section in the order they are assembled during a build.
- The LDF maps input sections into output sections, in the INPUT_SECTIONS() portion of the SECTIONS{} command. This “intermediate” mapping is useful for assembling analogous objects into one output section, using the .LDF file's macro expansion capabilities. For example, to build the wave_data output section from a combination of source code and library files, you would write:

```
SECTIONS {  
  
    ...  
  
    all_wave_data  
  
    { INPUT_SECTIONS( $OBJECTS(wave_data)  
  
        $LIBRARIES(wave_data) ) }
```

This command fragment places all code objects declared as `.SECTION wave_data` and all the library files labeled `wave_data` (originally assembled with `.SECTION wave_data` labels) into output section `all_wave_data`. The output section name is declared at the beginning of the line(s) defining its constituents.

Converting from .SYS to .LDF

- The output sections are mapped into memory segments in the top-level directives of the LDF's `SECTIONS{}` command. Continuing the example above, add the following command fragment to the preceding code.

```
... > all_wave_info;
```

At this point, the line places the `all_wave_data` output section, built up as shown above, into the `all_wave_info` memory segment. You could add another output section, `all_wave_references`, to `all_wave_info` in another line, by declaring its name and components, and appending

```
... > all_wave_info;
```

Listing C-3. Sample ADSP-2181 LDF File for Assembly Code

```
ARCHITECTURE(ADSP-2181)

SEARCH_DIR( $ADI_DSP\218x\lib )

$OBJECTS = $COMMAND_LINE_OBJECTS;

// 2181 has 16K words (24-bit) of Program RAM and 16K words
// (16-bit) of Data RAM
MEMORY
{
    seg_inttab { TYPE(PM RAM) START(0x00000) END(0x0002f)
WIDTH(24) }
    seg_code   { TYPE(PM RAM) START(0x00030) END(0x03fff)
WIDTH(24) }

    seg_data1  { TYPE(DM RAM) START(0x00000) END(0x00fff)
WIDTH(16) }
```

```
    seg_data2 { TYPE(DM RAM) START(0x01000) END(0x01fff)
WIDTH(16) }
} // end MEMORY

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

    SECTIONS
    {
        sec_inttab
        {
            INPUT_SECTIONS( $OBJECTS(interrupts) )
RESOLVE(begin, 0x0000)
        } >seg_inttab

        sec_code
        {
            INPUT_SECTIONS( $OBJECTS(program) )
        } >seg_code

        sec_data1
        {
            INPUT_SECTIONS( $OBJECTS(data1) )
        } >seg_data1

        sec_data2
        {
            INPUT_SECTIONS( $OBJECTS(data2) )
        } >seg_data2

        // support for initialization, including C++
        sec_ctor
```

Converting from .SYS to .LDF

```
    {
        INPUT_SECTIONS( $OBJECTS(ctor) )
    } >seg_data1
} // end SECTIONS
} // end PROCESSOR p0
```

Sample Hardware Overlay LDF File for Assembly Code

```
ARCHITECTURE(ADSP-2189)
$OBJECTS = $COMMAND_LINE_OBJECTS;

// The ADSP-2189M has 32K words (24-bit) of Program RAM and 48K
words (16-bit) of Data RAM
MEMORY{
    // Internal PM Memory Segments (Non-overlay segments)
    mem_vect_tbl    { TYPE(PM RAM) START(0x00000) END(0x0002f)
WIDTH(24) } // 48 locations for vector table (12 * 4)
    mem_pmcode     { TYPE(PM RAM) START(0x00030) END(0x01fff)
WIDTH(24) } // 8k segment for internal PM

    /////////////// Declare PM Overlay Segment ("Live Address")
below
    mem_pm_ovly    { TYPE(PM RAM) START(0x02000) END(0x03fff)
WIDTH(24) } // PM Overlay Segment "Live Address" Declaration
(for PLIT)

    // Internal PM Overlay Segments ("Run Addresses") below
    mem_pmpage0    { TYPE(PM RAM) START(0x02000) END(0x03fff)
WIDTH(24) } // Internal 8k PM Overlay Segment 0
    mem_pmpage4    { TYPE(PM RAM) START(0x42000) END(0x43fff)
WIDTH(24) } // Internal 8k PM Overlay Segment 4
    mem_pmpage5    { TYPE(PM RAM) START(0x52000) END(0x53fff)
WIDTH(24) } // Internal 8k PM Overlay Segment 5
```

```
////////// Declare DM Overlay Segment ("Live Address")
below
    mem_dm_ovly    { TYPE(DM RAM) START(0x00000) END(0x01fff)
WIDTH(16) }      // DM Overlay Segment "Live Address" Declaration
(for PLIT)

    // Internal DM Overlay Segments ("Run Addresses") below
    mem_dmpage0    { TYPE(DM RAM) START(0x00000) END(0x01fff)
WIDTH(16) }      // Internal 8k DM Overlay Segment 0
    mem_dmpage4    { TYPE(DM RAM) START(0x40000) END(0x41fff)
WIDTH(16) }      // Internal 8k DM Overlay Segment 4
    mem_dmpage5    { TYPE(DM RAM) START(0x50000) END(0x51fff)
WIDTH(16) }      // Internal 8k DM Overlay Segment 5
    mem_dmpage6    { TYPE(DM RAM) START(0x60000) END(0x61fff)
WIDTH(16) }      // Internal 8k DM Overlay Segment 6
    mem_dmpage7    { TYPE(DM RAM) START(0x70000) END(0x71fff)
WIDTH(16) }      // Internal 8k DM Overlay Segment 7

    // Internal DM Memory Segment (Non-overlay segment)
    mem_int_dm     { TYPE(DM RAM) START(0x02000) END(0x03fdf)
WIDTH(16) }      // Internal 8k segment minus 32 locations for mem-
ory mapped control registers

    // External PM Overlay Segments ("Live Addresses") below
    mem_pmpage1    { TYPE(PM RAM) START(0x12000) END(0x13fff)
WIDTH(24) }      // External 8k PM Overlay Segment 1
    mem_pmpage2    { TYPE(PM RAM) START(0x22000) END(0x23fff)
WIDTH(24) }      // External 8k PM Overlay Segment 2

    // External DM Overlay Segments ("Live Addresses") below
    mem_dmpage1    { TYPE(DM RAM) START(0x10000) END(0x11fff)
WIDTH(16) }      // External 8k DM Overlay Segment 1
    mem_dmpage2    { TYPE(DM RAM) START(0x20000) END(0x21fff)
WIDTH(16) }      // External 8k DM Overlay Segment 2
```

Converting from .SYS to .LDF

```
// End "MEMORY" declaration section

PROCESSOR p0{ // single processor declaration for processor "p0"

    OUTPUT($COMMAND_LINE_OUTPUT_FILE)

    SECTIONS{
        dxv_vect_tbl{
INPUT_SECTIONS($OBJECTS(interrupts))
>mem_vect_tbl // places the file object in internal
                non-overlay PM segment "mem_vect_tbl"
(PM0x0000-0x002f)

        dxv_code{
INPUT_SECTIONS($OBJECTS(program))
>mem_pmcode // places the file object in internal
                non-overlay PM segment "mem_pmcode" (PM0x0030-0x1fff)

        dxv_int_dm{
INPUT_SECTIONS(DM0vly0.doj(int_dmda))
>mem_int_dm // places the file object "DM0vly.doj" in
                internal non-overlay DM segment "mem_int_dm"
(DM0x2000-0x3fdf)

        // DM Hardware Overlay Page Declarations (assign data modules
to
        their "run address" memory regions)
        dxv_dmpage{
PAGE_INPUT{
ALGORITHM(ALL_FIT)
PAGE_OUTPUT(dm_page1.ovl) // assembler overlay output
                            file for linker
INPUT_SECTIONS(DM0vly1.doj(dm_ovly_1)) // input object
                            file for this overlay region
```

```
>mem_dmpage1    // assign to external DM overlay region #1

PAGE_INPUT{
ALGORITHM(ALL_FIT)
PAGE_OUTPUT(dm_page2.ovl)    // assembler overlay output
                               file for linker
INPUT_SECTIONS(DM0vly2.doj(dm_ovly_2))    // input object file
                               for this overlay region
>mem_dmpage2    // assign to external DM overlay region #2

PAGE_INPUT{
ALGORITHM(ALL_FIT)
PAGE_OUTPUT(dm_page4.ovl)    // assembler overlay output file
                               for linker
INPUT_SECTIONS(DM0vly4.doj(dm_ovly_4))    // input object file
                               for this overlay region
>mem_dmpage4    // assign to internal DM overlay region #4

PAGE_INPUT{
ALGORITHM(ALL_FIT)
PAGE_OUTPUT(dm_page5.ovl)    // assembler overlay output file
                               for linker
INPUT_SECTIONS(DM0vly5.doj(dm_ovly_5))    // input object file
                               for this overlay region
>mem_dmpage5    // assign to internal DM overlay region #5
>mem_dm_ovly    // assign overlay objects to DM overlay "run
                  address" DM0x0000-0x1fff

    // PM Hardware Overlay Page Declarations (assign code modules
    // to their "run address" memory regions)
dxpmpage{
PAGE_INPUT{
ALGORITHM(ALL_FIT)
```

Converting from .SYS to .LDF

```
PAGE_OUTPUT(pm_page0.ovl)    // assembler overlay output file
                             for linker
INPUT_SECTIONS(PM0vly0.doj(pm_ovly_0)) // input object file
                             for this overlay region
>mem_pmpage0 // assign to internal PM overlay region #0

PAGE_INPUT{
ALGORITHM(ALL_FIT)
PAGE_OUTPUT(pm_page4.ovl)    // assembler overlay output file
                             for linker
INPUT_SECTIONS(PM0vly4.doj(pm_ovly_4)) // input object file
                             for this overlay region
>mem_pmpage4 // assign to internal PM overlay region #4

PAGE_INPUT{
ALGORITHM(ALL_FIT)
PAGE_OUTPUT(pm_page5.ovl)    // assembler overlay output file
                             for linker
    INPUT_SECTIONS(PM0vly5.doj(pm_ovly_5)) // input object file
                             for this overlay region
>mem_pmpage5 // assign to internal PM overlay region #5
>mem_pm_ovly // assign overlay objects to PM overlay "run
              address" PM0x2000-0x3fff

} // end "SECTIONS" declaration section of LDF
} // "PROCESSOR" declaration section of LDF
```

Converting SYS .SEGMENT to LDF Commands

Each **.SEGMENT** directive in the **.SYS** file is replaced by a pair of **MEMORY{}** and **SECTION{}** commands in the **.LDF**. The memory definition part of the **.SEGMENT** directive in the **.SYS** file corresponds to a **MEMORY{}** command in

the LDF, which specifies your system's physical memory. Refer to the following example, which shows how to a .SYS segment directive converts to its equivalent MEMORY{} command in the LDF.

Example

The following example shows the conversion of a .SYS .segment to equivalent code in an .LDF file. This is the portion of the .SYS file.

```
.segment /pm /ram /begin=A /end=B /width C NAME;
```

The resulting LDF code is:

```
MEMORY {NAME {TYPE (PM RAM) START(A) END(B) WIDTH(C) }}
```

Example

[“Legacy .SYS File Example” on page C-21](#) is a legacy .SYS file from the 6.1 tools release, which works for all of the ADSP-218x DSPs with 16K byte PM/DM (or greater) on-chip.

Listing C-4. Legacy .SYS File Example

```
.SYSTEM Example_2181_Architecture;
.ADSP2181;
.SEG/PM/RAM/ABS=0x0000/CODE/DATA      PM_INT[8192];
.SEG/PM/RAM/ABS=0x2000/CODE/DATA      PM_OVLY[8192];
.SEG/DM/RAM/ABS=0x0000/DATA           DM_OVLY[8192];
.SEG/DM/RAM/ABS=0x2000/DATA           DM_INT[8160];
.ENDSYS;
```

By contrast, the .SYS file syntax had a limited set of commands — a .processor declaration, followed by a series of .SEGMENT declarations — to identify memory, type, address range and word size. No commands were available to specify search paths, libraries, inputs, or multiple output files, nor was expression-handling syntax or macro expansion possible in .SYS files.

Legacy Assembly of Sources with Absolute Placement Directives

The `-Ao` (absolute resolve) switch directs the assembler to write `RESOLVE()` commands to the default or specified header Linker Description File (`.LDF`).

The assembler generates a `RESOLVE()` command for each `.VAR/ABS=address` directive found in a legacy source program (a program developed under Release 6.1). The assembler then outputs a `resolve.ldf` file, which you must manually include into your project's LDF via the `INCLUDE()` command.

You can use a default `resolve.ldf` file name, composed of the `'resolve_'` prefix and the `.LDF` extension applied to the source (or base) file name, or override it with the `filename` parameter.

The following code examples show the translation of assembly `.VAR/ABS=` directives into corresponding `RESOLVE()` commands.

Listing C-5. VAR/ABS Legacy Directives

```
/* Filename:  varAbs.dsp
Requires -legacy. Expect the creation of the LDF file.
*/
#define CMN_BASE0x010000
.module test;
nop;
.var/DM/ABS=CMN_BASE+0x20  eq_outp; // expect Resolve()
.var/DM/ABS=CMN_BASE+0x22  eq_outq; // expect Resolve()
.endmod;
```

When this code is assembled with

```
-easm218x -c -legacy varAbs.dsp -Ao resolve1.ldf
```

the assembler overrides `resolve_varAbs.ldf` with `resolve1.ldf`.



To successfully compile a `resolve1.ldf`, you must declare the ‘InByte’ symbol globally in the `RESOLVE()` linker command. This symbol, which has no scope beyond a single module, does not appear in the symbol table.

Listing C-6. Generated RESOLVE() Commands

```
// Filename: resolve1.ldf
// These are assembler generated LDF RESOLVE() commands for
// -legacy .var directives with /ABS qualifiers of the previous//
// assembly example.
// .var eq_outp in "varAbs.dsp", line 9:
RESOLVE( eq_outp, 0x10020 )
// .var eq_outq in "varAbs.dsp", line 10:
RESOLVE( eq_outq, 0x10022 )
```

For more information about the `RESOLVE()` command, see the corresponding section in [“PROCESSOR{}” on page 2-48](#). For more information about the `-Ao` assembler switch, refer to the *Visual++ 3.0 Assembler and Preprocessor Manual for ADSP-218x DSPs*.

Legacy Assembly of Sources with Absolute Placement Directives

I INDEX

Symbols

\$COMMAND_LINE_LINK_AGAI

 NST LDF macro [2-20](#)

\$COMMAND_LINE_OBJECTS

 LDF LDF macro [2-19](#)

\$COMMAND_LINE_OUTPUT_DI

 RECTORY LDF macro [2-20](#)

\$COMMAND_LINE_OUTPUT_FI

 LE LDF macro [2-7](#), [2-20](#)

\$macro LDF macro [2-20](#)

\$OBJECTS LDF macro [2-5](#)

.ASM files

 assembler [A-3](#)

.DAT files

 initialization data [A-3](#)

.DLB files [1-31](#)

 description of [A-6](#)

.DOJ files [1-31](#)

 description of [A-6](#)

.DXE files [1-8](#), [1-31](#)

 boot loader [6-10](#)

 linker [A-7](#)

.EXE files [6-4](#)

.H files [6-6](#)

 ADSP-2192-12 DSPs [6-13](#)

.IDM files [A-10](#)

.LDF files [1-31](#)

 commands in [1-16](#), [2-21](#)

 comments in [2-9](#)

 creating in Expert Linker [3-4](#)

 memory segments [1-17](#)

 operators [2-13](#)

 output sections [1-17](#)

 overview of [2-3](#)

 purpose [1-17](#)

.LDL files [5-20](#)

.LDU files [5-20](#)

.MAP files

 linker [A-7](#)

.OVL files [1-8](#), [1-32](#), [2-55](#)

 boot loader [6-10](#)

 dumping [B-4](#)

 linker [A-7](#)

.SECTION directives [1-4](#)

.SEGMENT

 converting to SECTIONS{} [C-12](#)

.SM files [1-8](#), [1-32](#)

 ADSP-2192-12 DSPs [6-10](#)

 linker [A-7](#)

.SYS files

 converting to .LDF files [C-1](#)

.TXT files

 linker [A-5](#)

@ file linker command-line switch [1-36](#)

INDEX

`_ov_endaddress_#` [1-47](#), [1-64](#)
`_ov_runtimestartaddress_#` [1-47](#), [1-64](#)
`_ov_size_#` [1-47](#), [1-64](#)
`_ov_startaddress_` [1-64](#)
`_ov_startaddress_#` [1-47](#)
`_ov_word_size_live_#` [1-47](#), [1-64](#)
`_ov_word_size_run_#` [1-47](#), [1-64](#)

A

absolute data placements [1-39](#)
ABSOLUTE() LDF operator [2-14](#)
adding
 input sections [3-10](#)
 LDF macros [3-10](#)
 library files [3-10](#)
 object files [3-10](#)
ADDR() LDF operator [2-15](#)
ADSP-218x DSPs
 overlays [2-81](#)
 splitter [4-16](#)
ADSP-2192-12 DSPs
 boot loader switches [6-14](#)
 boot loader utility [6-2](#), [6-10](#)
 booting [6-4](#)
 elfloader.exe [6-1](#)
ADSP-219x DSPs
 loader [5-1](#), [5-16](#)
 loader switches [5-18](#)
 splitting [5-1](#)
aexe2elf.exe [B-5](#)
ALGORITHM() LDF command [2-55](#)
ALIGN() LDF command [2-22](#)
alignment
 specifying properties [3-56](#)

ALL_FIT LDF identifier [2-55](#), [3-59](#)
ARCHITECTURE() LDF command
 [2-23](#)
archive files
 see library files [A-6](#)
archive members [A-6](#)
archive routines
 creating entry points [7-4](#)
archiver
 about [7-1](#)
 command-line reference [7-7](#)
 constraints [7-9](#)
 file searches [7-6](#)
 running [7-7](#)
 symbol name encryption [7-10](#)
 use in disassembly [B-3](#)
ASCII byte-stream format [A-11](#)
assembler
 conventions [A-3](#)
 initialization data files (.DAT) [A-3](#)
 legacy absolute resolve switch [C-22](#)
 legacy directives [C-22](#)
 non-keyword operators [A-3](#)
 object files (.DOJ) [A-6](#)
 source files (.ASM) [A-3](#)

B

B0 bytes [2-38](#)
BDMA [4-6](#)
 about [1-43](#)
BEST_FIT LDF identifier [2-55](#)
BMODE pins
 ADSP-219x DSPs [5-3](#)
boot loader utility

- ADSP-2192-12 DSPs [6-3](#)
- command line
 - ADSP-218x loader [4-8](#)
 - ADSP-218x splitter [4-19](#)
 - ADSP-2192-12 loader [6-10](#)
 - ADSP-219x loader [5-18](#)
- command lines
 - elfloader [5-16](#)
- commands
 - LDF [2-21](#)
 - linker [1-28](#)
- comments
 - .LDF file [2-9](#)
- conventions, manual [-xxiii](#)
- converting
 - .SEGMENT to SECTIONS{} [C-12](#)
 - .SYS files to .LDF files [C-1](#)
 - legacy files [B-5](#)
- Create LDF wizard [3-4](#)
- creating
 - run-time boot loader [6-6](#)
- customer support [-xvii](#)
- D**
- Darchitecture (target architecture)
 - linker command-line switch [1-36](#)
- data placement [1-38](#)
- debugger
 - files [A-13](#)
- DEFINED() LDF operator [2-15](#)
- dialog boxes
 - Legend [3-13](#)
- directories
 - supported by linker [1-30](#)
- disassembly
- C**
- cache
 - flushing [1-58](#)
- checksums [5-5](#)
- circular buffer with absolute placement
 - [C-22](#)
- color selection
 - Expert Linker [3-13](#)
- boot stream
 - contents [5-4](#)
- boot types
 - ADSP-218x DSPs [4-4](#)
 - ADSP-219x DSPs [5-2](#)
- booting
 - ADSP-218x DSPs [4-3](#)
 - ADSP-2192-12 DSPs [6-4](#)
 - ADSP-219x DSPs [5-2](#)
 - stream format on ADSP-219x [5-4](#)
 - verifying accuracy [5-5](#)
 - via UART on ADSP-219x [5-7](#)
- boot-stream format [5-4](#)
- branch expansion instruction [1-39](#)
- branch instruction [2-44](#)
- build files
 - description of [A-6](#)
- build options
 - ADSP-219x loader [5-1](#)
 - loader [6-5](#)
- byte_order_list byte [2-38](#)
- byte-stream format [A-11](#)

INDEX

- using archiver [B-3](#)
- using dumper [B-3](#)
- drivers
 - downloading for ADSP-2192-12 DSPs [6-6](#)
 - run-time boot loader [6-6](#)
- DSPs
 - development software [1-3](#)
- dumper
 - use in disassembly [B-3](#)
- DWARF
 - references [A-14](#)
- E**
- e (eliminate unused symbols) linker
 - command-line switch [1-38](#)
- ELF
 - references [A-14](#)
- ELF file dumper
 - command-line switches [B-1](#)
 - extracting data [B-1](#)
 - overlay library files [B-4](#)
- elf2aexe.exe [B-6](#)
 - switches [B-6](#)
- elfar.exe
 - about [7-1](#)
 - command-line reference [7-7](#)
- elfdump.exe
 - used by Expert Linker [3-35](#)
- elfloader.exe
 - ADSP-2192-12 DSPs [6-1](#), [6-10](#)
 - command line [5-16](#)
- ELIMINATE() LDF command [2-23](#)

- ELIMINATE_SECTIONS() LDF
 - command [2-24](#)
- elimination
 - specifying properties [3-45](#)
- empty bytes [2-38](#)
- encrypting
 - symbol names in libraries [7-10](#)
- END() LDF identifier [2-29](#)
- EPROM booting [4-6](#)
- errors
 - linker [1-23](#)
- es (eliminate listed sections) linker
 - command-line switch [1-38](#)
- ev (eliminate unused symbols, verbose) linker command-line switch [1-38](#)
- Expert Linker
 - about [3-1](#)
 - color selection [3-13](#)
 - launching [3-3](#)
 - mapping sections in [3-12](#)
 - memory map window [3-16](#)
 - object properties [3-40](#)
 - overview [1-22](#), [3-1](#)
 - resize cursor [3-25](#)
 - window [3-10](#)
- extracting data
 - ELF executable files [B-1](#)
- F**
- file extensions
 - ADSP-218x loader [4-8](#)
 - ADSP-2192-12 loader [6-11](#), [6-13](#)
 - ADSP-219x loader [5-17](#)

- files
 - .EXE [6-4](#)
 - .H [6-6](#)
 - .IDM [A-10](#)
 - .LDL [5-20](#)
 - .LDU [5-20](#)
 - boot-stream [5-4](#)
 - boot-stream format [5-4](#)
 - byte-stream format [A-11](#)
 - debugger [A-13](#)
 - linker command line [1-30](#), [1-32](#)
 - output [1-8](#)
- FILL() LDF command [2-24](#), [2-54](#)
- FIRST_FIT LDF identifier [2-55](#)
- fixed-point values
 - .DAT files [A-3](#)
- format
 - byte-stream [A-11](#)
- fragmented memory [1-38](#)
- G**
- gaps
 - inserting [3-32](#)
- global properties
 - setting [3-41](#)
- H**
- hard overlays [1-43](#)
- hardware overlays
 - setting up [3-20](#)
- hardware pages
 - overlays [3-20](#)
- heap
 - graphic representation [3-60](#)
 - managing in memory [3-60](#)
- help (command line help) linker
 - command-line switch [1-38](#)
- host booting [4-13](#)
- I**
- i (include search directory) linker
 - command-line switch [1-38](#)
- icons
 - Expert Linker [3-13](#)
 - unmapped icon [3-12](#)
- IDMA [4-13](#)
 - about [1-43](#)
 - host boot files [A-10](#)
- INCLUDE() LDF command [2-24](#)
- individual data placement option [1-39](#)
- input sections [1-26](#)
 - adding [3-10](#)
 - names [1-26](#)
 - source code [1-4](#)
- Input Sections pane [3-10](#)
 - menu selections [3-10](#)
- INPUT_SECTION_ALIGN() LDF
 - command [2-24](#)
- INPUT_SECTIONS() LDF identifier
 - [2-8](#), [2-53](#)
- inserting
 - gaps [3-32](#)
- inter-overlay calls [1-66](#)
- inter-processor calls [1-66](#)
- ip (individual placement) linker
 - command-line switch [1-38](#)

INDEX

J

-jcs21 (convert short calls) linker
command-line switch [1-39](#)

K

-keep (keep unused symbols) linker
command-line switch [1-39](#)

KEEP() LDF command [2-26](#)

L

-L path (libraries and objects) linker
command-line switch [1-36](#)

LDF

see .LDF files [1-8](#)

LDF commands [1-16](#), [2-21](#)

ALIGN() [2-22](#)

ARCHITECTURE() [2-23](#)

ELIMINATE() [2-23](#)

ELIMINATE_SECTIONS() [2-24](#)

FILL() [2-54](#)

INCLUDE() [2-24](#)

INPUT_SECTION_ALIGN() [2-24](#)

KEEP() [2-26](#)

LINK_AGAINST() [2-26](#)

MAP() [2-27](#)

MEMORY{} [2-27](#)

MPMEMORY{} [2-30](#)

OVERLAY_GROUP{} [2-32](#)

OVERLAY_INPUT() [2-54](#)

PACKING() [2-37](#)

PAGE_INPUT() [2-40](#)

PAGE_OUTPUT() [2-42](#)

PLIT{} [2-42](#), [2-54](#)

PROCESSOR{} [2-48](#)

RESOLVE() [2-50](#)

SEARCH_DIR() [2-50](#)

SECTIONS{} [2-51](#)

LDF macros

about [2-18](#)

adding [3-10](#)

list of [2-19](#)

LDF operators

ABSOLUTE() [2-14](#)

ADDR() [2-15](#)

DEFINED() [2-15](#)

MEMORY_SIZEOF() [2-16](#)

SIZEOF() [2-17](#)

legacy files

converting [B-5](#)

Legend dialog box [3-13](#)

legends

Expert Linker [3-11](#)

LENGTH() LDF identifier [2-29](#)

librarian

VisualDSP++ [7-1](#)

libraries

members [7-1](#)

symbol name encryption [7-10](#)

library files

adding [3-10](#)

creating [7-3](#)

description of [A-6](#)

searching [7-2](#)

library members

library routines

using [7-5](#)

Link page

options [1-20](#)

- LINK_AGAINST() LDF command
 - 2-26
- linker
 - about 1-13
 - command-line files (.TXT) A-5
 - command-line syntax 1-28
 - commands 1-28
 - describing the target 1-24
 - error messages 1-23
 - executable files A-7
 - file name conventions 1-31
 - memory map files (.MAP) A-7
 - objects 1-32
 - outputs 1-8
 - overlay constants generated by 1-47
 - selecting a target processor 1-40
 - switches 1-15
 - warning messages 1-23
- linker command-line switches
 - Darchitecture 1-36
 - e 1-38
 - es secName 1-38
 - ev 1-38
 - help 1-38
 - i path 1-38
 - ip 1-38
 - jcs21 1-39
 - keep symName 1-39
 - L path 1-36
 - M 1-36
 - Map file 1-37
 - MDmacro 1-37
 - MM 1-37
 - o filename 1-39
 - pp 1-40
 - proc processor 1-40
 - S 1-37
 - s 1-40
 - sp 1-40
 - t 1-40
 - T file 1-37
 - v 1-41
 - version 1-41
 - warnonce 1-41
 - xref filename 1-41
- Linker Description Files
 - see .LDF files A-4
- linking
 - controlling 1-15
 - file with large uninitialized variables
 - 2-63
 - multiprocessor system 2-77
 - overlay memory system 2-74
 - process 2-3
 - single-processor system 2-61
- loader
 - ADSP-218x switches 4-2
 - ADSP-2192-12 switches 6-11
 - ADSP-219x switches 5-1
 - build options 6-5
 - setting options 5-1
- loader kernel
 - inclusion 5-4
- location counter 2-17
- M
 - M (dependency check and output)
 - linker command-line switch 1-36

INDEX

- macros
 - LDF [2-18](#)
- Map (file) linker command-line switch [1-37](#)
- MAP() LDF command [2-27](#)
- mapping
 - input sections to output sections [3-12](#)
- MDmacro (macro value) linker
 - command-line switch [1-37](#)
- memory
 - allocation [1-25](#), [1-26](#)
 - architecture representation [1-24](#)
 - managing heap/stack [3-60](#)
 - overlays [1-42](#)
 - partitions [3-16](#)
 - segment declaration [1-25](#)
 - segments [3-16](#)
 - types [1-25](#)
- Memory Map pane [3-16](#), [3-17](#)
 - overlays [3-33](#)
- memory maps
 - graphical view [3-22](#)
 - highlighted objects in [3-25](#)
 - specifying [1-26](#)
 - tree view [3-21](#)
 - viewing [3-16](#)
- memory overlays [1-44](#)
- memory segments [1-17](#), [1-26](#), [3-16](#)
 - changing size of [3-24](#)
 - gap [3-32](#)
 - size [3-21](#)
 - specifying properties [3-51](#)
 - start address [3-21](#)
- memory spaces [1-44](#)
- MEMORY_SIZEOF() LDF operator [2-16](#)
- MEMORY{} LDF command [1-25](#), [2-7](#), [2-27](#)
- MM (dependency check, output and build) linker command-line switch [1-37](#)
- Mode D pin
 - ADSP-218x DSPs [4-5](#)
- MPMEMORY{} LDF command [2-30](#)
- multiple overlays [3-33](#)
- multiprocessor systems
 - linking [2-77](#)
- N
 - non-bootable EPROM images [4-15](#)
 - NOP instruction [3-57](#)
 - null bytes [2-38](#), [2-39](#)
- O
 - o (output file) linker command-line switch [1-39](#), [1-40](#)
 - object files
 - adding [3-10](#)
 - object properties
 - managing with Expert Linker [3-40](#)
 - objects
 - deleting [3-11](#)
 - sorting [3-15](#)
 - operators
 - LDF [2-13](#)
 - OPMODE pin
 - ADSP-219x DSPs [5-3](#)
 - output directory

- specifying 1-40
 - output sections 1-17, 1-26
 - specifying properties 3-53
 - OUTPUT() LDF command 2-7, 3-16
 - ov_id_loaded buffer 1-56
 - overlay
 - dumping library files B-4
 - overlay algorithm
 - ALL_FIT 3-59
 - overlay manager 1-42, 1-45, 1-49, 2-44
 - constants 1-53
 - major functions 1-45
 - placing constants 1-55
 - routine steps 1-59
 - overlay memory
 - linking for 2-74
 - OVERLAY_GROUP{} LDF
 - command 2-32
 - OVERLAY_ID LDF identifier 2-55
 - OVERLAY_INPUT LDF command 2-54
 - overlays
 - address 1-47
 - ADSP-218x DSPs 2-81
 - archive files B-4
 - constants 1-47, 1-53
 - grouped 2-32
 - grouping 2-32
 - hard 1-43
 - hardware page 3-20
 - in Memory Map pane 3-33
 - live address table 6-8
 - live space 3-33
 - loading instructions with PLIT 2-46
 - manager overhead (reducing) 1-59
 - managing properties 3-58
 - memory 1-42, 1-44
 - multiple 3-33
 - physical 1-43
 - run space 3-34
 - soft 1-43
 - start address 6-9
 - symbols 1-64
 - ungrouped 2-32
 - word size 1-47
 - OvlMgrTbl symbol 6-8
 - OvlPciAdrTbl symbol 6-8
- ## P
- packing
 - efficient 2-39
 - specifying properties 3-55
 - PACKING() LDF command 2-37
 - PAGE_INPUT() LDF command 2-40
 - PAGE_OUTPUT() LDF command 2-42
 - PCI drivers
 - run-time boot loader 6-7
 - physical overlays 1-43
 - pinning
 - to output section 3-19
 - PLIT
 - allocating space for 2-44
 - executing user-defined code 1-48
 - resolving inter-overlay calls 1-66
 - specifying properties 3-44
 - syntax 2-43
 - PLIT LDF commands

INDEX

PLIT_DATA_OVERLAY_IDS
2-43
PLIT_SYMBOL_ADDRESS 2-43
PLIT_SYMBOL_OVERLAYID
2-43
PLIT linker commands
PLIT_SYMBOL_ADDRESS 2-44
saving register contents 2-46
PLIT_DATA_OVERLAY_IDS 2-43
PLIT_SYMBOL_ADDRESS 2-43
PLIT_SYMBOL_OVERLAYID 2-43
PLIT{} LDF command 2-42, 2-54
-pp (end after preprocessing) linker
command-line switch 1-40
preprocessor
running from linker 1-40
-proc (processor)
linker command-line switch 1-40
procedure linkage table (PLIT) 1-48,
1-64, 2-42
processor
selection 1-36
PROCESSOR{} LDF command 2-48
processors
specifying properties 3-43
program counter 1-53
project builds
linker 1-19
Project Options dialog box
Link page 1-20
PROM splitter 4-16

R
resize cursor 3-25

RESOLVE() LDF command 2-27,
2-50
RESOLVE_LOCALLY() LDF
command 2-56
RTBL
see run-time boot loader 6-6
run-time boot loader
creating 6-6

S
-s (strip all symbols) linker
command-line switch 1-40
-S (strip debug symbols) linker
command-line switch 1-37
SEARCH_DIR() LDF command 2-50
SECTIONS{} LDF command 1-27,
2-7, 2-51
selecting
target processor 1-40
SHT_NOBITS
keyword 2-63
section qualifier 2-63
SIZE() LDF command 2-56
SIZEOF() LDF operator 2-17
soft overlays 1-43
software
development 1-3
sorting
objects 3-15
source code
in input sections 1-4
source files
assembly instructions A-3
C/C++ A-2

- fixed-point data [A-3](#)
- sp (skip preprocessing) linker
 - command-line switch [1-40](#)
- specifying
 - output directory [1-40](#)
- splitter
 - ADSP-218x DSPs [4-16](#)
 - ADSP-219x DSPs [5-1](#)
- SPORT data files [A-13](#)
- stack
 - graphic representation [3-60](#)
 - managing in memory [3-60](#)
- symbol declaration [2-5](#)
- symbols
 - adding [3-48](#)
 - deleting [3-50](#)
 - encryption of names [7-10](#)
 - managing properties [3-47](#)
 - removal [1-40](#)
 - viewing [3-38](#)
- sysstack
 - managing in memory [3-60](#)

T

- t (trace) linker command-line switch
 - [1-40](#)
- T file (executable program placement)
 - linker command-line switch [1-37](#)
- target processor
 - specifying [1-36](#)
- tree view
 - memory map [3-21](#)

U

- uninitialized variables [2-63](#)
- unmapped object icon [3-12](#)
- utilities
 - aexe2elf.exe [B-5](#)
 - conversion [B-1](#)
 - elf2aexe.exe [B-6](#)
 - elfdump.exe [B-1](#)
 - elfloader.exe (ADSP-2192-12) [6-1](#)
 - elfspl21.exe (ADSP-218x) [4-16](#)
 - file dumper [B-1](#)

V

- v (verbose) linker command-line
 - switch [1-41](#)
- VAR/ABS legacy directives [C-22](#)
- version (linker version) linker
 - command-line switch [1-41](#)
- VisualDSP++
 - archiver [7-1](#)
 - conversion utilities [B-1](#)
 - creating library files [7-3](#)
 - Expert Linker [3-2](#)
 - librarian [7-1](#)

W

- warnings
 - linker [1-23](#)
- warnonce (single symbol warning)
 - linker command-lineswitch [1-41](#)
- Windows drivers
 - downloading for ADSP-2192-12
 - loader [6-6](#)
- wizards

INDEX

Create LDF [3-4](#)

X

-xref (external reference file) linker command-line switch [1-41](#)