

VISUALDSP++™ 3.0

Linker and Utilities Manual

for SHARC® DSPs

Revision 4.0, January 2003

Part Number
82-001965-01

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



Copyright Information

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

Printed in the USA.

Disclaimer

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

Trademark and Service Mark Notice

The Analog Devices logo, VisualDSP, the VisualDSP logo, 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	xix
DSP Product Information	xix
Related Documents	xx
Online Documentation	xxi
From VisualDSP++	xxii
From Windows	xxii
From the Web	xxii
Printed Manuals	xxiii
VisualDSP++ Documentation Set	xxiii
Hardware Manuals	xxiii
Data Sheets	xxiii

CONTENTS

Contacting DSP Publications	xxiv
Notation Conventions	xxv

LINKER AND UTILITIES

Software Development Flow	1-2
Compiling and Assembling	1-3
Inputs – C/C++ and Assembly Sources	1-3
Input Section Directives in Assembly Code	1-4
Input Section Directives in C/C++ Source Files	1-4
Linking	1-6
Loading and Splitting	1-7
Linking Overview	1-10
Linker Operation	1-10
Directing Linker Operation	1-12
Linking Process Rules	1-13
Linker Description File (.LDF)	1-14
Linking Environment	1-15
Project Builds	1-15
Expert Linker	1-18
Linker Warning and Error Messages	1-19
Link Target Description	1-20
Representing Memory Architecture	1-20
Specifying the Memory Map	1-21
Placing Code on the Target	1-25
Linker Command-Line Reference	1-28

Linker 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
@filename	1-36
-DprocessorID	1-36
-L path	1-37
-M	1-37
-MM	1-37
-Map file	1-37
-MDmacro[=def]	1-37
-S	1-38
-T filename	1-38
-e	1-38
-es sectionName	1-38
-ev	1-38
-h or -help	1-38
-i path	1-39
-keep symbolName	1-39
-o filename	1-39
-od filename	1-39

CONTENTS

-pp	1-39
-proc ProcessorID	1-39
-s	1-40
-sp	1-40
-t	1-40
-v	1-40
-version	1-40
-warnonce	1-40
-xref filename	1-40
Memory Management Using Overlays	1-41
Memory Overlays	1-42
Overlay Managers	1-43
Memory Overlay Support	1-44
Example – Managing Two Overlays	1-49
Reducing Overlay Manager Overhead	1-58
Using PLIT{} and Overlay Manager	1-63

LINKER DESCRIPTION FILE

LDF Guide	2-2
.LDF File Overview	2-3
Example – Basic .LDF File	2-4
Notes on Basic .LDF File Example	2-5
LDF Structure	2-8
Command Scoping	2-9
LDF Expressions	2-10

LDF Keywords, Commands, and Operators	2-11
Miscellaneous LDF Keywords	2-12
LDF Operators	2-13
ABSOLUTE() Operator	2-13
ADDR() Operator	2-14
DEFINED() Operator	2-15
MEMORY_SIZEOF() Operator	2-15
SIZEOF() Operator	2-16
Location Counter (.)	2-16
LDF Macros	2-17
Built-In LDF Macros	2-18
User-Declared Macros	2-19
LDF Macros and Command-Line Interaction	2-20
LDF Commands	2-20
ALIGN()	2-21
ARCHITECTURE()	2-22
ELIMINATE()	2-22
ELIMINATE_SECTIONS()	2-23
INCLUDE()	2-23
INPUT_SECTION_ALIGN()	2-23
KEEP()	2-25
LINK_AGAINST()	2-25
MAP()	2-26
MEMORY{}	2-26

CONTENTS

MPMEMORY{}	2-29
OVERLAY_GROUP{}	2-31
Ungrouped Overlay Execution	2-33
Grouped Overlay Execution	2-35
PACKING()	2-37
Packing in SHARC DSPs	2-39
Overlay Packing Formats	2-41
External Execution Packing	2-42
Default Packing – No Reordering	2-43
PLIT{}	2-44
PLIT Syntax	2-44
Allocating Space for PLITs	2-46
PLIT Examples	2-47
PLIT – Summary	2-48
PROCESSOR{}	2-50
RESOLVE()	2-52
SEARCH_DIR()	2-52
SECTIONS{}	2-53
SHARED_MEMORY{}	2-58
LDF Programming Examples	2-61
Example – Linking a Single-Processor SHARC System	2-62
Example – Linking Large Uninitialized Variables	2-64
Example – Linking for MP and Shared Memory	2-65
Reflective Semaphores	2-71

Example – Linking for Overlay Memory	2-72
--	------

EXPERT LINKER

Expert Linker Overview	3-2
Launching the Create LDF Wizard	3-4
Step 1: Specifying Project Information	3-5
Step 2: Specifying System Information	3-6
Step 3: Completing the LDF Wizard	3-8
Expert Linker Window Overview	3-9
Using the Input Sections Pane	3-10
Using the Input Sections Menu	3-10
Mapping an Input Section to an Output Section	3-12
Viewing Icons and Colors	3-13
Sorting Objects	3-15
Using the Memory Map Pane	3-16
Using the Context Menu	3-17
Tree View Memory Map Representation	3-21
Graphical View Memory Map Representation	3-22
Specifying Pre- and Post-Link Memory Map View	3-30
Zooming In and Out on the Memory Map	3-30
Inserting a Gap into a Memory Segment	3-34
Working With Overlays	3-35
Viewing Section Contents	3-37
Viewing Symbols	3-39
Managing Object Properties	3-42

CONTENTS

Managing Global Properties	3-43
Managing Processor Properties	3-44
Managing PLIT Properties for Overlays	3-46
Managing Elimination Properties	3-47
Managing Symbols Properties	3-49
Managing Memory Segment Properties	3-53
Managing Output Section Properties	3-55
Managing Packing Properties	3-57
Managing Alignment and Fill Properties	3-58
Managing Overlay Properties	3-60
Managing Stack and Heap in DSP Memory	3-62

ARCHIVER

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

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

ADSP-2106X/ADSP-21160 LOADER

Loader Guide	5-2
Booting	5-2
General Power-Up Booting Process	5-4
Boot Mode Selection	5-5
Boot-Loading Process	5-8
Boot-Kernel Changes and Software Issues	5-10
Boot-Kernels	5-13
Interrupt Vector Table Location	5-14
Boot Types	5-15
EPROM Booting	5-15
Host Booting	5-19
Link Booting	5-23
No Boot Mode	5-24
Multiprocessor Booting	5-24
Multiprocessor EPROM Booting	5-24
Running the Loader	5-26
Load Page	5-26
Loader Command Line	5-26
File Names	5-29
File Extensions	5-29
Loader Command-Line Switches	5-30
Processor ID Numbers	5-33

ADSP-21161N LOADER

Loader Guide	6-2
Booting	6-3
Power-Up Booting Process	6-4
Boot Mode Selection	6-4
Boot-Kernels	6-5
Boot-Kernel Modification and Loader Issues	6-6
Rebuilding a Boot-Kernel File	6-6
Rebuilding a Boot-Kernel Using Command Lines	6-7
Loader File Issues	6-8
Loader File Header Words	6-9
Interrupt Vector Table Location	6-10
Include-Format Files for Booting	6-11
Boot Types	6-12
EPROM Booting	6-12
Host Booting	6-17
Link Port Booting	6-20
SPI Port Booting	6-22
No Boot Mode	6-24
Multiprocessor Booting	6-24
Multiprocessor EPROM Booting	6-25
Bootting From a Single EPROM	6-25
Sequential EPROM Booting	6-26
Running the Loader	6-27

Load Page	6-27
Loader Command Line	6-27
File Names	6-30
File Extensions	6-31
ADSP-21161 Loader Command-Line Switches	6-31
Processor ID Numbers	6-34

SPLITTER

Splitter Guide	7-2
Split Page	7-2
Splitter Command-Line Reference	7-4
elfspl21k Command-Line Syntax	7-4
File Searches	7-6
Output File Extensions	7-7
Command-Line Items	7-8

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-5
Linker Description Files (.LDF)	A-5
Linker Command-Line Files (.TXT)	A-6
Build Files	A-6

CONTENTS

Assembler Object Files (.DOJ)	A-7
Library Files (.DLB)	A-7
Linker Output Files (.DXE, .SM, and .OVL)	A-7
Memory Map Files (.MAP)	A-8
Loader Output Files in Intel Hex-32 Format (.LDR)	A-8
Loader Output Files in ASCII Format (.LDR)	A-10
Loader Output Files in Include Format (.LDR)	A-10
Loader Output Files in Binary Format (.LDR)	A-11
Splitter Output Files in Motorola S-Record Format (.S_#)	A-11
Splitter Output Files in Intel Hex-32 Format (.H_#)	A-13
Splitter Output Files in Byte-Stacked Format (.STK)	A-13
Debugger Files	A-15
Format References	A-16

UTILITIES

elfdump – ELF File Dumper	B-1
Disassembling a Library Member	B-4
Dumping Overlay Library Files	B-5
mem21k – Memory Initializer	B-6

LINKER LEGACY SUPPORT

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 SHARC DSPs* contains information about the linker and utilities (loader, splitter, archiver) for SHARC 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 also describes the Expert Linker, 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. The manual leads you through using the linker and other utilities necessary to produce DSP programs.

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 includes the following parts:

- 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.
- Chapter 4, “[Archiver](#)”
Describes how to combine object files into reusable library files to link routines referenced by other object files.
- Chapter 5, “[ADSP-2106x/ADSP-21160 Loader](#)”
Describes loading for most models of SHARC DSPs.
- Chapter 6, “[ADSP-21161N Loader](#)”
Describes loading for ADSP-21161N DSPs.
- Chapter 7, “[Splitter](#)”
Describes the splitter used to create EPROM 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](#)”
Provides an overview of legacy support issues for developers porting software built with older releases of the development tools.

What’s New in This Manual

In addition to documenting all existing linker, loader, 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 describes how linking and loading fit into the DSP development process.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

- Visit the DSP Development Tools Web site at www.analog.com/technology/dsp/developmentTools/index.html
- E-mail questions to dsptools.support@analog.com

Supported Processors

- 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 “*SHARC*” refers to a family of DSPs. VisualDSP++ for SHARC DSPs currently supports the following processors:

- ADSP-21020
- ADSP-21060/60L
- ADSP-21061/61L
- ADSP-21062/62L
- ADSP-21065L
- ADSP-21160
- ADSP-21161N

Product Information

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

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

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly e-mail notification containing updates to the Web pages that meet your interests.

MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:

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

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

DSP Product Information

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

Product Information

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

- E-mail questions or requests for information to
`dsp.support@analog.com`
- Fax questions or requests for information to
1-781-461-3010 (North America)
+49 (0) 089 76 903 157 (Europe)
- Access the Digital Signal Processor Division's FTP Web site at
`ftp ftp.analog.com` or **ftp 137.71.23.21**
`ftp://ftp.analog.com`

Related Documents

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

VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs

VisualDSP++ 3.0 User's Guide for SHARC DSPs

VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs

VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs

VisualDSP++ 3.0 Product Bulletin for SHARC DSPs

VisualDSP++ 3.0 Kernel (VDK) User's Guide for SHARC DSPs

VisualDSP++ Component Software Engineering User's Guide for SHARC DSPs

Quick Installation Card

For hardware information, refer to documentation that describes the DSP target, such as a *Hardware Reference*, *Technical Reference*, *Instruction Set Reference*, and data sheet.

All documentation is available online. Most documentation is available in printed form.

Online Documentation

Online documentation comprises Microsoft HTML Help (.CHM), Adobe Portable Documentation Format (.PDF), and HTML (.HTM and .HTML) files. A description of each file type is as follows.

File	Description
.CHM	VisualDSP++ online Help system files and VisualDSP++ manuals are provided in Microsoft HTML Help format. Installing VisualDSP++ automatically copies these files to the <code>VisualDSP\Help</code> folder. Online Help is ideal for searching the entire tools manual set. Invoke Help from the VisualDSP++ Help menu or via the Windows Start button.
.PDF	Manuals and data sheets in Portable Documentation Format are located in the installation CD's <code>Docs</code> folder. Viewing and printing a .PDF file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running <code>setup.exe</code> on the installation CD provides easy access to these documents. You can also copy .PDF files from the installation CD onto another disk.
.HTM or .HTML	Dinkum Abridged C++ library and FlexLM network license manager software documentation is located on the installation CD in the <code>Docs\Reference</code> folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.

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

Product Information

From VisualDSP++

VisualDSP++ provides access to online Help. It does not provide access to .PDF files or the supplemental reference documentation (Dinkum Abridged C++ library and FlexLM network licence). Access Help by:

- Choosing **Contents**, **Search**, or **Index** from the VisualDSP++ **Help** menu
- Invoking context-sensitive Help on a user interface item (toolbar button, menu command, or window)

From Windows

In addition to shortcuts you may construct, Windows provides many ways to open VisualDSP++ online Help or the supplementary documentation.

Help system files (.CHM) are located in the `VisualDSP\Help` folder. Manuals and data sheets in PDF format are located in the `Docs` folder of the installation CD. The installation CD also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation in the `\Reference` folder.

Using Windows Explorer

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

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

Printed copies of 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

Printed copies of hardware 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 title or by product number (located on the back cover of each manual).

Data Sheets

All data sheets can be downloaded from the Analog Devices Web site. As a general rule, printed copies of data sheets with a letter suffix (L, M, N, and S) can be obtained from the Literature Center at **1-800-ANALOGD (1-800-262-5643)** or downloaded from the Web site. Data sheets without

Product Information

the suffix can be downloaded from the Web site only—no hard copies are available. You can ask for the data sheet by part name or by product number.

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

Contacting DSP Publications

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

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

Notation Conventions

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

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

Example	Description
Close command (File menu) or OK	Text in bold style indicates the location of an item within the VisualDSP++ environment's menu system and user interface items.
{this that}	Alternative required items in syntax descriptions appear within curly brackets 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, code examples, 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.

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

Notation Conventions

1 LINKER AND UTILITIES

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

This chapter includes:

- [“Software Development Flow” on page 1-2](#) – Shows how linking, loading, and splitting fit into the DSP project development process.
- [“Linking Overview” on page 1-10](#) – Describes the link target, linking environment, and other files used while linking.
- [“Linker Command-Line Reference” on page 1-28](#) – Describes linker command-line switches.
- [“Memory Management Using Overlays” on page 1-41](#) – Describes memory overlays as well as advanced LDF commands.

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

Software Development Flow

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

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

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

Finally, you process the debugged executable file(s) through the loader or splitter to create output for use on the actual DSP. The output file may reside on another processor (host) or may be burned into a PROM. Chapters 5, 6, and 7 describe options in generating these output files.

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 VisualDSP++ **Project Options** dialog box settings, the linker utility consumes object files (.DOJ) to yield an executable (.DXX) file. If specified, shared memory (.SM) and overlay (.OVL) files are also produced.
3. Loading or splitting – The executable (.DXX), as well as shared memory (.SM) and overlay (.OVL) files, are processed to yield output file(s). For SHARC DSPs, these are bootloadable (.LDR) files or non-bootable PROM image files, which execute from DSP external memory.

Compiling and Assembling

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

Inputs – C/C++ and Assembly Sources

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

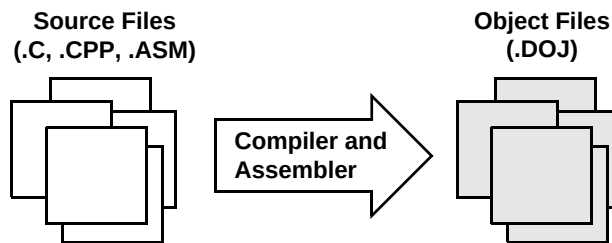


Figure 1-1. Compiling and Assembling

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

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

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

Input Section Directives in Assembly Code

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

Example

```
.section /dm asmdata;      // Declares section asmdata
.var input[3];            // Declares data buffer in asmdata

.section /pm asmcode;      // Declares section asmcode
R0 = 0x1234;              // Three lines of code in asmcode
R1 = 0x4567;
R3 = R1 + R2;
```

In the above example, the `asmdata` input section contains the array `input`, and the `asmcode` input section contains the three lines of code.

Input Section Directives in C/C++ Source Files

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

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

Example

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

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

Example

In the following example, `section ("extern")` is optional. Note the new function (`func2`) does not require `section ("extern")`. For more information on LDF sections, refer to [“Specifying the Memory Map” on page 1-21](#).

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

For information on compiler default section names, refer to the *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs* and column 1 of [Table 1-1 on page 1-22](#).

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

Linking

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

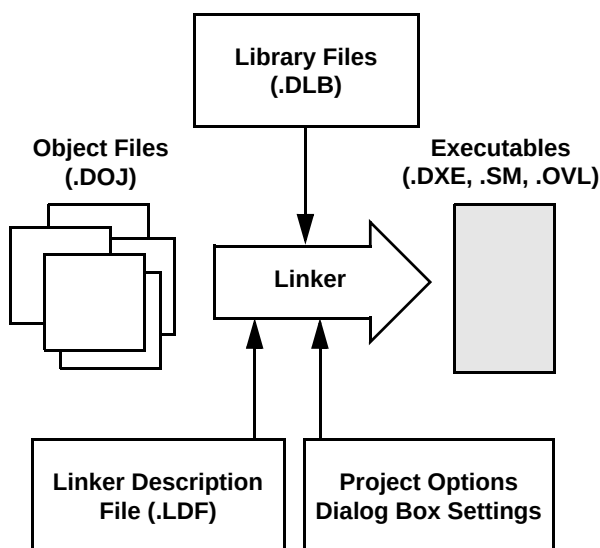


Figure 1-2. Linking

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

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

Loading and Splitting

After debugging the `.DXE` file, you process it through a loader or splitter to create output files used by the actual DSP. This file may reside on another processor (host) or may be burned into a PROM. Chapters 5 and 6 describe options for generating bootloadable `.LDR` (loader) files.

Chapter 7 describes splitting, which creates non-bootloadable files that execute from the DSP's external memory. Splitting uses the `elfspl21k` utility, which allows you to specify the contents of the memory image file(s), their width, and format.

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

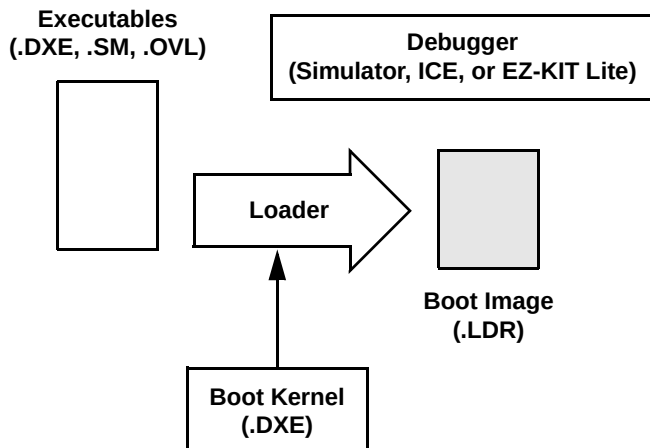


Figure 1-3. Loading

For SHARC DSPs, use the loader (`elfloader.exe`) to yield a bootloadable image (`.LDR`), which resides in memory external to the DSP (PROM or host processor). 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 1-4](#)) by the boot-kernel.

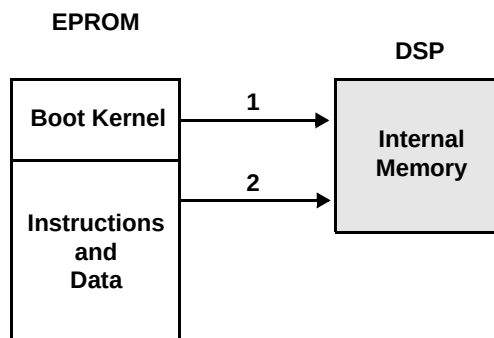


Figure 1-4. Booting from a Bootloadable (.LDR) File

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

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

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

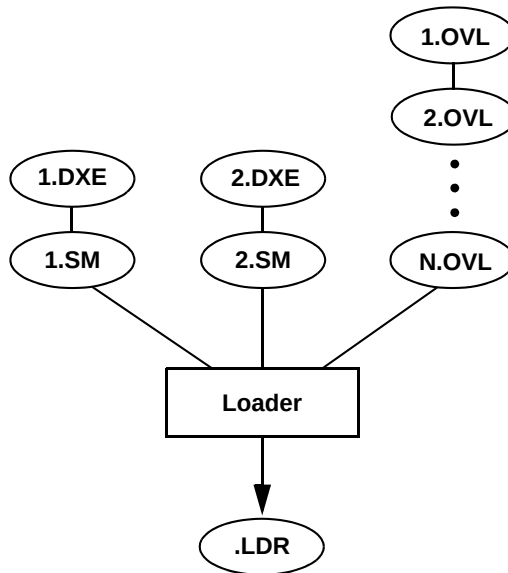


Figure 1-5. Input Files for a Multiprocessor System

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

Linking Overview

Linking assigns code and data to DSP memory. For a simple single-processor architecture, a single `.DXE` file is generated. Repeat the same linking process to create multiple executable files (`.DXE`) for multiprocessor (MP) architectures. Linking can also produce a shared memory (`.SM`) file for an MP system. A large executable can be split into a smaller executable and overlays (`.OVL` files), which contain code that is called in (swapped into internal DSP memory) as needed. [“Memory Management Using Overlays” on page 1-41](#) provides information about advanced topics.

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

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

Linker Operation

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



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

When a DSP architecture is for a multiprocessor system, a `.DXE` file for each processor is generated. For example, if your multiprocessor application includes eight SHARC DSPs, you must generate eight `.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, whose code is used by multiple processors.

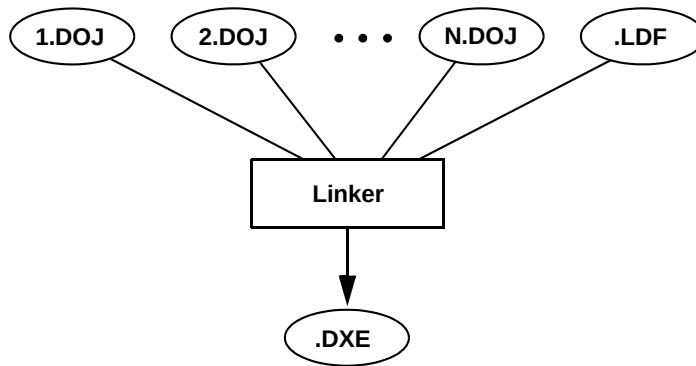


Figure 1-6. Linking Object Files to Produce an Executable File

Overlay files (.OVL), another linker output, support applications that require more program instructions and data than can fit in the processor's internal memory. Refer to [“Memory Management Using Overlays” on page 1-41](#) for more information.

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

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

The executable file(s) and auxiliary files (.SM and .OVL) are not loaded into the 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`) command-line switches (options)
- Settings (options) on the **Link** page of the **Project Options** dialog box
- LDF commands

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

LDF commands in a Linker Description File (`.LDF`) define the target memory map and the placement of program sections within 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/assembled object(s).
- The compiler and assembler produce object code with labels that direct various portion(s) to particular output sections.
- As directed by the `.LDF` file, the linker maps each input section in the object code to an output section in the `.DXE` file.
- As directed by the `.LDF` file, the linker maps each output section to a memory segment.
- Each input section may contain multiple code items, but a code item may appear in one input section only.
- More than one input section may be placed in an output section.
- Each memory segment must have a specified width.
- Contiguous addresses on different-width hardware must reside in different memory segments.
- More than one output section may map to a memory segment if the output sections fit completely within the memory segment.

Linker Description File (.LDF)

Whether you are linking C/C++ functions or assembly routines, the mechanism is the same. After converting the source files into object files, the linker uses directives in an .LDF file to combine the objects into an executable file (.DXX), 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 that specifies the linking process by defining the target memory and mapping the code and data into that memory. You can write your own .LDF file, or you can modify an existing file; modification is often the easier alternative when there are few changes in your system's hardware or software. VisualDSP++ provides an .LDF file that supports the default mapping of each DSP type.

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

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

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

Linking Environment

The linking environment refers to Windows Command Prompt windows and the VisualDSP++ IDDE. At a minimum, run DSP development tools (such as the linker) via a command line and view 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's **Type** option allows you to choose whether to build a library (.DLB) file, an executable (.EXE) file, or an image file. Error and warning messages display in the **Output** window.

Project Builds

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

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



Refer to the *VisualDSP++ 3.0 User's Guide for SHARC 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, select text in an IDDE editor window or **Output** window and press the keyboard's F1 key.

Linking Overview

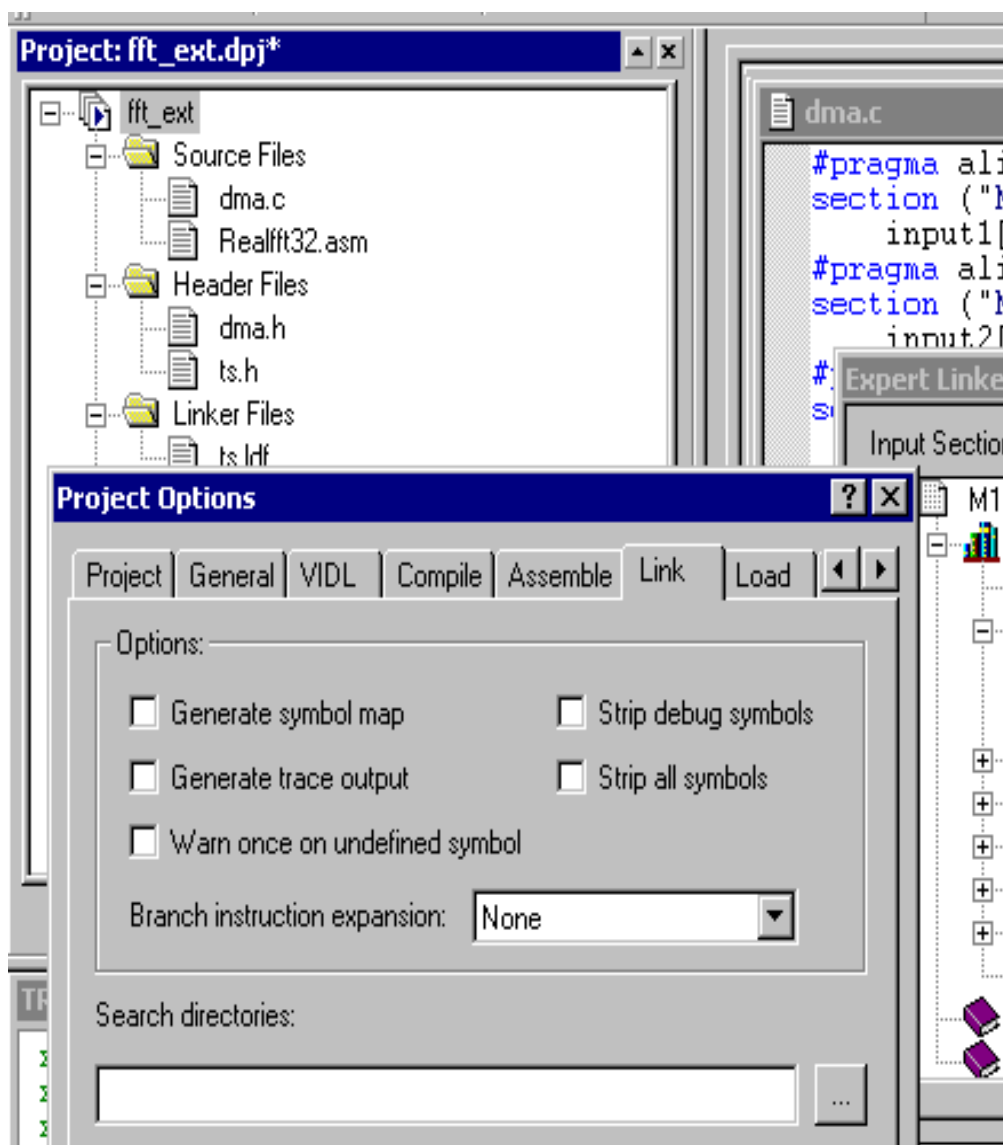


Figure 1-7. VisualDSP++ Environment

Within VisualDSP++, specify tool settings for project builds. Modify linker options via the **Link** page of the **Project Options** dialog box (Figure 1-8).

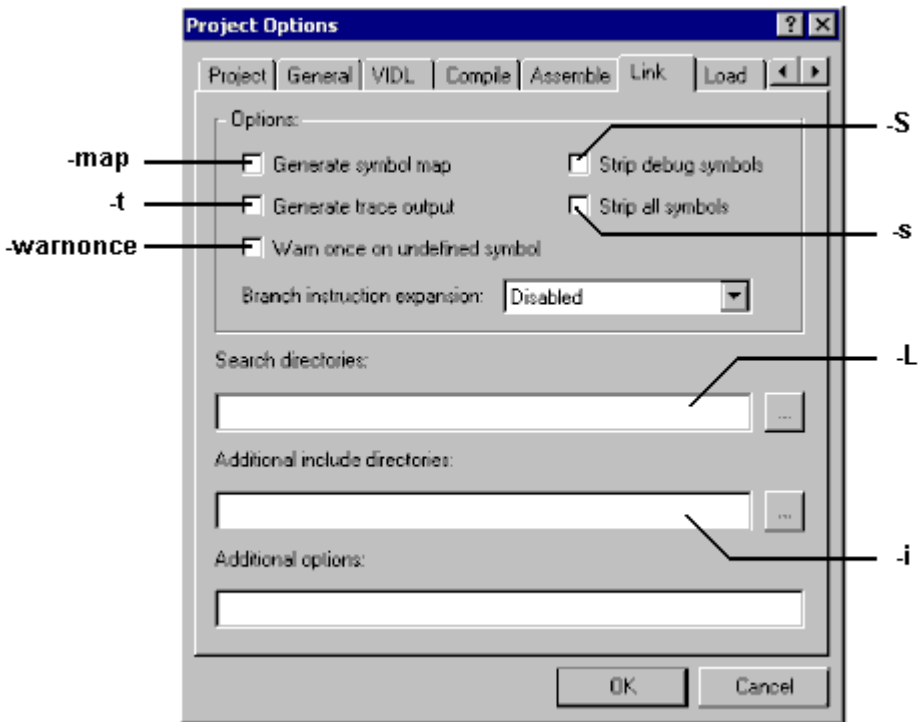


Figure 1-8. Link Page

The callouts in Figure 1-8 refer to corresponding linker command-line switches. In addition to selecting options on the **Link** page, use the page's **Additional options** field to specify switches and parameters that do not have corresponding **Link** page controls. Refer to “[Linker Command-Line Reference](#)” on page 1-28 for more information.

Linking Overview

Expert Linker

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

Figure 1-9 shows the Expert Linker window, which comprises two panes: **Input Sections** and **Memory Map** (output sections). Refer to “[Expert Linker](#)” on page 3-1 for additional information.

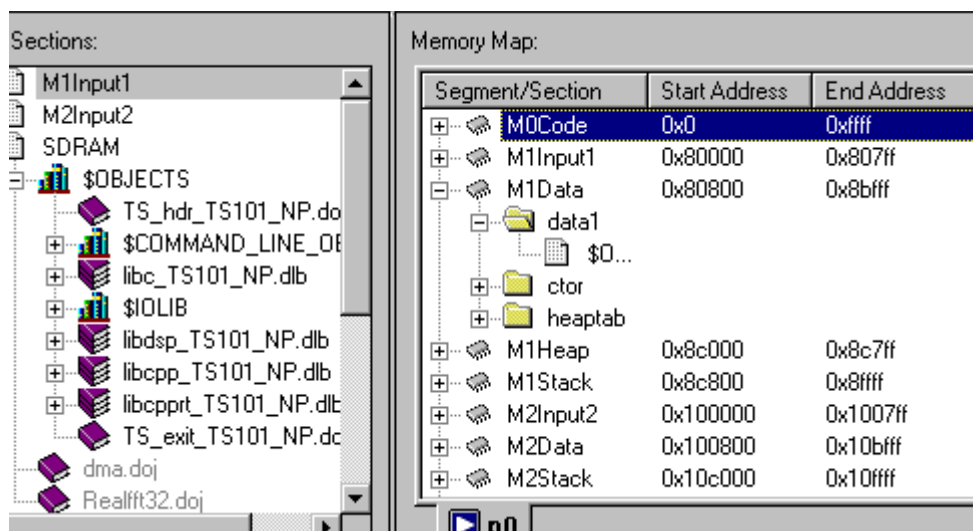


Figure 1-9. Expert Linker Window

Linker Warning and Error Messages

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

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

Example

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

Interpreting Linker Messages

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

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

Linking Overview

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

Link Target Description

Before defining the DSP system's memory and program placement with linker commands, analyze the target DSP system, so you can describe the target in terms the linker can process. Then, 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-21062 DSPs.

Be sure to understand the DSP's memory architecture, which is 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 this information to place the executable file into the system's memory.

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

- **Memory Usage.** List the ways your program uses memory in your system. Typical uses for memory segments include interrupt tables, initialization data, program code, data, heap space, and stack space. Refer to [“Specifying the Memory Map”](#).
- **Memory Characteristics.** List the types of memory in your DSP system and the address ranges and word width associated with each memory type. Memory type is defined as `RAM` or `ROM`.
- **Memory {} Command.** Construct a `MEMORY{ }` command to combine the information from the previous two lists and to declare your system’s memory segments. Use [“Overlay Declaration in an .LDF File” on page 1-45](#) as an example.

For complete information, refer to [“MEMORY{ }” on page 2-26](#).

Specifying the Memory Map

A DSP program must conform to the constraints imposed by the processor’s data path (bus) widths and addressing capabilities. The following steps show an `.LDF` file for a hypothetical project. This file specifies several memory segments that support the `SECTIONS{ }` command. (See [“SECTIONS{ }” on page 2-53](#).)

The three steps involved in allocating memory are demonstrated below.

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

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

Linking Overview

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

[Table 1-1](#) shows how input sections, output sections, and memory segments correspond in the ADSP-21062 DSP's default `.LDF` file. Refer to your DSP's default `.LDF` file and to the DSP's *Hardware Reference* manual for details.

Table 1-1. Section Mapping in the Default ADSP-21062 LDF

Input Section	Output Section	Memory Segment
seg_pmco	dxe_pmco	mem_pmco
seg_dmda	dxe_dmda	mem_dmda
seg_pmda	dxe_pmda	mem_pmda
heap	dxe_heap	mem_heap
stackseg	dxe_stak	mem_stak
sec_rth	dxe_rth	mem_rth
seg_init	seg_init	mem_init
seg_pmco	dxe_pmco	mem_ovly

2. **Memory characteristics** – In the example system, the ADSP-21062 DSP has internal memory addresses from 0x0 to 0x17FFFF. Refer to [Table 1-2](#).

Table 1-2. Example – ADSP-21062 Memory Structure

Block	Range	Word Size
	0x00000 - 0x000FF	IOP registers
	0x00100 - 0x1FFFF	Reserved
Block 0	0x20000 - 0x27FFF	Normal words (32 or 48 bits)
Block 1	0x28000 - 0x2FFFF	Normal words (32 or 48 bits)
Block 1 alias	0x30000 - 0x37FFF	Normal words (32 or 48 bits)
Block 1 alias	0x38000 - 0x3FFFF	Normal words (32 or 48 bits)
Block 0	0x40000 - 0x4FFFF	Short words (16 bits)
Block 1	0x50000 - 0x5FFFF	Short words (16 bits)
Block 1 alias	0x60000 - 0x6FFFF	Short words (16 bits)
Block 1 alias	0x70000 - 0x7FFFF	Short words (16 bits)

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 as shown in [Listing 1-1](#).

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

```
MEMORY
{
    /* This MEMORY{} command declares: */
    /* o 256 words of run-time header in memory block 0 */
    /* o 256 words of initializatin code in memory block 0 */
    /* o 18K words of C code space in memory block 0 */
    /* o 1.5K words of C PM data space in memory block 0 */
    /* o 16K words of C PM data space in memory block 0 */
    /* o 8K words of C heap space in memory block 1 */
    /* o 8K words of C stack space in memory block 1 */

    mem_rth    {TYPE(PM RAM) START(0x20000) END(0x200FF) WIDTH(48)}
    mem_init   {TYPE(PM RAM) START(0x20100) END(0x201FF) WIDTH(48)}
    mem_pmco   {TYPE(PM RAM) START(0x20200) END(0x249FF) WIDTH(48)}
    mem_pmda   {TYPE(PM RAM) START(0x24A00) END(0x24FFF) WIDTH(40)}
    mem_dmda   {TYPE(DM RAM) START(0x2A000) END(0x2BFFF) WIDTH(32)}
    mem_heap   {TYPE(DM RAM) START(0x2C000) END(0x2DFFF) WIDTH(32)}
    mem_stak   {TYPE(DM RAM) START(0x2E000) END(0x2FFFF) WIDTH(32)}
}
```

Notes on the Above Example

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

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

Placing Code on the Target

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

To write a `SECTIONS{}` command:

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

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

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

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



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

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

```
SECTIONS {
/* Begin output sections */
/* Place code in M0 (internal memory) */
dxerth { // run-time header and interrupt table
    INPUT_SECTIONS( $OBJ$(seg_rth) $LIB$(seg_rth))
} >mem_rth
```

Linking Overview

```
dxo_init { // Initialization data
    INPUT_SECTIONS( $OBS(seg_init) $LIBS(seg_init))
} >mem_init
dxo_pmco { // PM code
    INPUT_SECTIONS( $OBS(seg_pmco) $LIBS(seg_pmco))
} >mem_pmco
dxo_pmda { // PM data
    INPUT_SECTIONS( $OBS(seg_pmda) $LIBS(seg_pmda))
} >mem_pmda
dxo_rth { // DM data
    INPUT_SECTIONS( $OBS(seg_dmda) $LIBS(seg_dmda))
} >mem_dmda

stackseg{
    // Declare the stack space and length.
    // The linker will generate the following symbols:
    // * ldf_stack_space
    // * ldf_stack_length
    // These symbols will be entered into the DXE's
    // symbol table. The symbol definitions are required
    // since there are references to them in seg_init.asm
    // (where the stack and heap variables are initialized).
    //
    ldf_stack_space = .;
    ldf_stack_length = 0x2000;
} > seg_stak
    // Declare the heap space length.
    //
    // The linker will generate the following symbols:
    // * ldf_heap_space
    // * ldf_heap_length
    // * ldf_heap_end
    //
    // These symbols will be entered into the DXE's
```

```
        // symbol table. The symbol definitions are required
        // since there are references to them in seg_init.asm
        // (where the stack and heap variables are initialized).
        //
ldf_heap_space = .;
ldf_heap_end = ldf_heap_space + 0x2000;
ldf_heap_length = ldf_heap_end - ldf_heap_space;
} > seg_heap;
}          // end sections
```

Linker Command-Line Reference

This section provides reference information, including:

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



When using the linker via the VisualDSP++ IDDE, the settings on the **Link** page of the **Project Options** dialog box correspond to linker command-line switches. VisualDSP++ calls the linker with these switches when linking your code. For more information, see the *VisualDSP++ 3.0 User's Manual for SHARC DSPs* and VisualDSP++ online Help.

Linker Command Syntax

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

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

The linker command requires `-proc processorID` or a `-T <ldf name>` for the link to proceed. If the command line 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.

Example

The following is an example linker command.

```
linker -proc ADSP-21062 p0.doj -T target.ldf -t -o program.dxe
```

-  Use `-proc processorID` instead of `-Darchitecture` on the command line to select the target processor. See [Table 1-4 on page 1-34](#) for more information.
-  The linker command line (except for file names) is case sensitive. For example, `linker -t` differs from `linker -T`.

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

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

Command Line Object Files

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

- Standard object files (`.DOJ`) produced by the assembler.
- One or more libraries (archives), each with a `.DLB` extension. Examples include the C run-time libraries and math libraries included with VisualDSP++. You may create libraries of common or specialized objects. Special libraries are available from DSP algorithm vendors. For more information, see [“Archiver” on page 4-1](#)
- An executable (`.DxE`) file to be linked against. Refer to `$COMMAND_LINE_LINK_AGAINST` in [“Built-In LDF Macros” on page 2-18](#).

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 1-3 on page 1-30](#) lists valid file extensions used by the linker.

Command-Line File Names

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

Table 1-3. File Extension Conventions

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

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

1. Specified path – If the command line includes relative or absolute path information on the command line, the linker searches that location for the file.
2. Specified directories – If you do not include path information on the command line and the file is not in the default directory, the linker searches for the file in the search directories specified with the `-L` (path) command-line switch, and then searches directories specified by `SEARCH_DIR` commands in the `.LDF` file. Directories are searched in order of appearance on the command line or in the LDF.
3. Default directory – If you do not include path information in the `.LDF` file named by the `-T` switch, the linker searches for the `.LDF` file in the current working directory. If you use a default `.LDF` file (by omitting LDF information in the command line and instead specify `-Darchitecture`), the linker searches in the processor-specific LDF directory; for example, `...\$ADI_DSP\21062\ldf`.

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

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

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

Objects

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

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

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

For more information on objects, see the `$COMMAND_LINE_OBJECTS` macro. For information on executables, see the `$COMMAND_LINE_LINK_AGAINST` macro. Both are described in [“Built-In LDF Macros” on page 2-18](#).

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

Linker Command-Line Switches

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

Switch Format

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

```
-switch [argument]
```

Rules

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



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

Example

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

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

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

Switch	Description	More Info
<code>@file</code>	Uses the specified file as input on the command line.	on page 1-36
<code>-DprocessorID</code>	Specifies the target processor ID. The use of <code>-proc processorID</code> is recommended.	on page 1-36
<code>-L path</code>	Adds the path name to search libraries for objects.	on page 1-37
<code>-M</code>	Produces dependencies.	on page 1-37
<code>-MM</code>	Builds and produces dependencies.	on page 1-37
<code>-Map file</code>	Outputs a map of link symbol information to a file.	on page 1-37

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

Switch	Description	More Info
<code>-MDmacro[=def]</code>	Defines and assigns value <i>def</i> to a preprocessor macro.	on page 1-37
<code>-S</code>	Omits debugging symbols from the output file.	on page 1-38
<code>-T filename</code>	Names the LDF.	on page 1-38
<code>-e</code>	Eliminates unused symbols from the executable.	on page 1-38
<code>-es secName</code>	Names input sections (<i>secName</i> list) to which elimination algorithm is being applied.	on page 1-38
<code>-ev</code>	Eliminates unused symbols verbosely.	on page 1-38
<code>-h</code> <code>-help</code>	Outputs the list of command-line switches and exits.	on page 1-38
<code>-i path</code>	Includes search directory for preprocessor include files.	on page 1-39
<code>-ip</code>	Not currently available. Fills fragmented memory with individual data objects that fit. This option requires that objects have been assembled with the assembler's <code>-ip</code> switch.	
<code>-keep symName</code>	Retains unused symbols.	on page 1-39
<code>-o filename</code>	Outputs the named executable file.	on page 1-39
<code>-od filename</code>	Specifies the output directory.	on page 1-39
<code>-pp</code>	Stops after preprocessing.	on page 1-39
<code>-proc processorID</code>	Selects a target processor. This is preferred over the older <code>-DprocessorID</code> switch.	on page 1-39
<code>-s</code>	Strips symbol information from the output file.	on page 1-40
<code>-sp</code>	Skips preprocessing.	on page 1-40
<code>-t</code>	Outputs the names of link objects.	on page 1-40

Linker Command-Line Reference

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

Switch	Description	More Info
-v -verbose	Verbose. Outputs status information.	on page 1-40
-version	Outputs version information and exits.	on page 1-40
-warnonce	Warns only once for each undefined symbol.	on page 1-40
-xref <i>filename</i>	Outputs a list of all cross-referenced symbols.	on page 1-40

@*filename*

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

-D*processorID*

Specifies target processor (architecture); for example, -DADSP-21062 or -DADSP-21160.



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

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

-L *path*

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

-M

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

-MM

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

-Map *file*

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

-MD*macro*[=*def*]

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

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

Linker Command-Line Reference

-S

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

-T filename

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

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

-e

Eliminates unused symbols from the executable.

-es sectionName

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

-ev

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

-h or -help

Displays a summary of command-line options and exits.

-i *path*

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

-keep *symbolName*

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

-o *filename*

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

-od *filename*

Specifies the value of the `$COMMAND_LINE_OUTPUT_DIRECTORY` LDF macro. This allows you to make a command-line change that propagates to many places without changing the `.LDF` file. Refer to [“Built-In LDF Macros” on page 2-18](#)

-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-21062`
or `-proc ADSP-21160`.

Linker Command-Line Reference

-s

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

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

-sp

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

-t

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

-v

Verbose. Outputs status information while linking.

-version

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

-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-108* and *EE-166*) 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 1-42](#)
- [“Overlay Managers” on page 1-43](#)
- [“Memory Overlay Support” on page 1-44](#)
- [“Example – Managing Two Overlays” on page 1-49](#)
- [“Reducing Overlay Manager Overhead” on page 1-58](#)
- [“Using PLIT{} and Overlay Manager” on page 1-63](#)

The following LDF commands facilitate advanced linker features.

- [“OVERLAY_GROUP{}” on page 2-31](#)
- [“PLIT{}” on page 2-44](#)
- [“MEMORY{}” on page 2-26](#)

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*.

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

Overlay functions are written to *overlay files* (.OVL), which may be specified as one type of linker executable output file. The loader can read .OVL files to generate an .LDR file.

Figure 1-10 demonstrates the concept of memory overlays. There are two memory spaces: internal and external. The external memory is partitioned into five overlays. The internal memory contains the main program, an overlay manager function, and two memory segments reserved for execution of overlay program instructions.

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

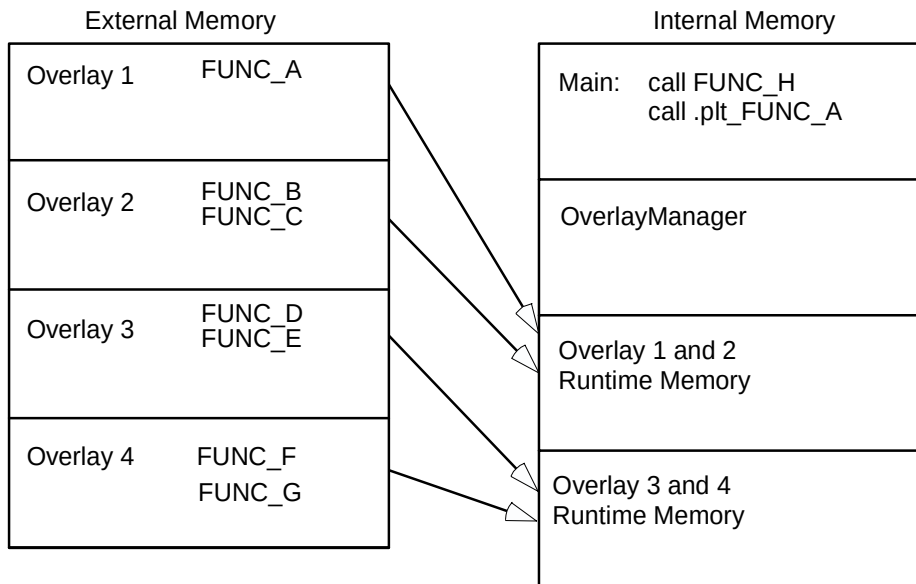


Figure 1-10. Memory Overlays

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

Overlay Managers

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

Memory Management Using Overlays

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

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

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

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

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

You may require an overlay manager to perform other specialized tasks to satisfy the special needs of a given application.

Memory Overlay Support

The overlay support provided by the DSP tools includes:

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

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

[Listing 1-3](#) shows the portion of an .LDF file that defines two overlays. This overlay declaration configures the two overlays to share a common run-time memory space. The syntax for the `OVERLAY_INPUT{}` command is described in [“OVERLAY_GROUP{ }” on page 2-31](#).

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

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

```
.dxm_pmco
{  OVERLAY_INPUT {
    OVERLAY_OUTPUT (OVLY_one.ovl)
    INPUT_SECTIONS (FUNC_A.doj(sec_pmco))
  } >ovl_pmco

  OVERLAY_INPUT
    OVERLAY_OUTPUT (OVLY_two.ovl)
    INPUT_SECTIONS (FUNC_B.doj(sec_pmco) FUNC_C.doj(pm_code))
  } >ovl_pmco
} >sec_pmco
```

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

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

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

Memory Management Using Overlays

`_ov_word_size_run_#`

`_ov_word_size_live_#`

`_ov_runtimestartaddress_#`

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

Overlay live and run word sizes differ when internal memory and external memory widths are different. For example, the instruction word width of the SHARC DSP is 48-bits. A system containing 32-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 32-bit words required to pack all of the 48-bit instructions.

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

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

The following is an example call instruction to an overlay function:

```
R0 = DM(I0,M3);
R1 = R0 * R2;
CALL FUNC_A;          /* Call to function in overlay */
DM(I3,M3) = R1;
```


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

```
R0 = DM(I0,M3);
R1 = R0 * R2;
CALL .plt_FUNC_A;      /* Call to PLIT entry */
DM(I3,M3) = R1;
```

`.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` file.

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

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

Listing 1-4. PLIT Definition in LDF

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

Memory Management Using Overlays

The linker expands the PLIT definition into individual entries in a table. An entry is created for each overlay symbol as shown in [Listing 1-5](#). 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).

Overlay 1		Overlay 2	
FUNC_A		FUNC_B	
		FUNC_C	

Internal Memory		
Main:	Plit_table	
call .plt_FUNC_A	.plt_FUNC_A	r0 = 0x00001;
.		r1 = 0x22000;
.		jump OverlayManager;
call .plt_FUNC_C	.plt_FUNC_B	r0 = 0x00002;
call .plt_FUNC_B		r1 = 0x22000;
.		jump OverlayManager;
.	.plt_FUNC_C	r0 = 0x00002;
.		r1 = 0x23000;
		jump OverlayManager;

Figure 1-11. Expanded PLIT Table

For example, the first PLIT entry in [Listing 1-5](#) is for the overlay symbol `FUNC_A`. The linker replaces the constant name `PLIT_SYMBOL_OVERLAYID` with the ID of the overlay containing `FUNC_A`. The linker also replaces the constant name `PLIT_SYMBOL_ADDRESS` with the run-time address of `FUNC_A`.

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

Example – Managing Two Overlays

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

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

The overlay manager:

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

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

Memory Management Using Overlays

Listing 1-5. FFT Overlay Example 1

```
OVERLAY_INPUT
{
    ALGORITHM (ALL_FIT)
    OVERLAY_OUTPUT (fft_one.ovl)
    INPUT_SECTIONS ( Fft_1st_last.doj(seg_pmco) )
        PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0
                B7 B8 B9 B10 B0)
    } > ovl_pmco    // Overlay to live in section ovl_code
OVERLAY_INPUT
{
    ALGORITHM (ALL_FIT)
    OVERLAY_OUTPUT (fft_two.ovl)
    INPUT_SECTIONS ( Fft_mid.doj(seg_pmco) )
        PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0
                B7 B8 B9 B10 B0)
    } > ovl_pmco    // Overlay to live in section ovl_c
```

The two defined overlays (fft_one.ovl and fft_two.ovl) live in memory segment ovl_pmco (defined by the MEMORY{} command), and run in section seg_pmco. 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 seg_pmco within the file Fft_mid.doj are part of overlay fft_two.ovl. The result is two functions within each overlay.

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

To verify whether an overlay is in internal memory, place the overlay ID of this overlay into a register (for example, R0) and compare this value to the overlay ID of each overlay already loaded by loading these overlay values into a register (for example, R1).

```
                /* Is overlay already in internal memory? */
R0 = R0 - R1;
                /* If so, do not transfer it in. */
if EQ jump continue;
```

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 (db);
    call fft_middle_stages (db);
    call fft_next_to_last (db);
    call fft_last_stage (db);
wait:
    idle;
    jump wait;
```

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

```
PLIT
{
    R0 = PLIT_SYMBOL_OVERLAYID;
    j5 = PLIT_SYMBOL_ADDRESS;
    JUMP OverlayManager;
}
```

Memory Management Using Overlays

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

Linker-Generated Constants

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

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

The constants provide the following information to the overlay manager.

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

Overlay Word Sizes

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

These are the linker-generated constants.

```

_ov_startaddress_1      = 0x20000
_ov_startaddress_2      = 0x20000
_ov_endaddress_1        = 0x20017
_ov_endaddress_2        = 0x20017
_ov_word_size_run_1     = 0x00018
_ov_word_size_run_2     = 0x00018
_ov_word_size_live_1    = 0x00010
_ov_word_size_live_2    = 0x00010
_ov_runtimestartaddress_1 = 0x08800
_ov_runtimestartaddress_2 = 0x08800

```

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

```

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

```

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.

Memory Management Using Overlays

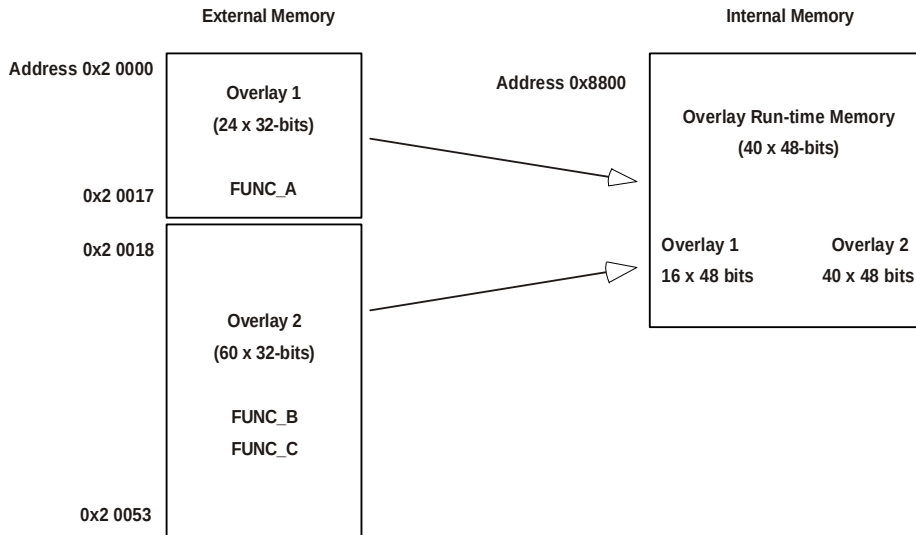


Figure 1-12. Example of Overlay Live and Run Memory Sizes

Figure 1-12 shows the difference between overlay “live” and “run” size.

- Overlays 1 and 2 are instruction overlays with a run word width of 48 bits.
- Because external memory is 32 bits, the live word size is 32 bits.
- Overlay 1 contains one function with 16 instructions. Overlay 2 contains two functions with a total of 40 instructions.
- The “live” word size for overlays 1 and 2 are 24 and 60 words, respectively.
- The “run” word size for overlay 1 and 2 are 16 and 40 words, respectively.

Storing Overlay ID

The overlay manager also stores the ID of an overlay that is currently in internal memory. When an overlay is transferred to internal memory, the overlay manager stores the overlay ID in internal memory in the buffer labeled `ov_id_loaded`. Before another overlay is transferred, the overlay manager compares the required overlay ID with that stored in the `ov_id_loaded` buffer. If they are equal, the required overlay is already in internal memory and a transfer is not required. The PC is sent to the proper location to execute the referenced function. If they are not equal, the value in `ov_id_loaded` is updated and the overlay is transferred into its internal run space via DMA.

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 R0, set in the PLIT of the LDF.
   Set up DMA transfer to internal memory through the
   external port. Store values of registers used
   by the overlay manager on the stack. */

dm(ov_stack)    = i8;
dm(ov_stack+1) = m8;
dm(ov_stack+2) = l8;
dm(ov_stack+3) = R2;

/* Use the overlay ID as an index (must subtract one)    */
R0 = R0 - 1;      /* Overlay ID -1                      */
m8 = R0;          /* Offset into the arrays containing
                  linker-defined overlay constants.    */

R2 = dm(ov_id_loaded);
R0 = R0 - R2;
if EQ jump continue;
[dm(ov_id_loaded) = m8;
```

Memory Management Using Overlays

```
R0 = i0;          dm(ov_stack+4) = R0;
R0 = m0;          dm(ov_stack+5) = R0;
R0 = l0;          dm(ov_stack+6) = R0;
```

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.

The index registers I8 and I7 point to the first location of the arrays. The overlay ID is stored in the modify registers M8 and M7. The index and modify registers together in DAG instructions read the appropriate elements from the arrays.

```
//  Get overlay "run" and "live" addresses from memory
//  and use them to set up the master mode DMA.
i8 = runAddresses;
i0 = liveAddresses;

r0 = 0;                // Disable DMA.
dm(DMAC0) = r0;

//  Set DMA external pointer to overlay live address
r0 = dm(m0,i0);
dm(EIEP0) = r0;

//  Set DMA internal pointer to overlay run address
r0 = pm(m8,i8);
dm(IIEP0) = r0;

i0 = runWordSize; // Number of words stored in internal memory
                // Most likely, the word size will be
                // 48 bits for instructions.

// Set DMA external modify to 1.
r0 = 1;
```

```
dm(EMEP0) = r0;

i8 = liveWordSize; // Number of words stored in external memory
                  // Most likely, the word size will be 32 or
                  // 16 bits for external storage.

dm(IMEP0) = r0;    // Set DMA internal modify to 1.
r0 = dm(m0,i0);    // Set DMA internal count to overlay run size.
dm(CEP0) = r0;

r0 = pm(m8,i8)     // Set DMA ext. count to overlay live size.
dm(ECEP0) = r0;

r0 = 0x2E1;        // DMA enabled, instruction word, Master,
dm(DMAC0) = r0;    // 48-32 packing
```

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

Summary

In summary, the overlay manager routine does the following.

- Maintains a run-time stack for registers being used by the overlay manager
- Compares the requested overlay's ID with that of the previously loaded overlay (stored in the `ov_id_loaded` buffer)

Memory Management Using Overlays

- Sets up the DMA transfer of the overlay (if it is not already in internal memory)
- Jumps the PC to the run-time location of the referenced function

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

Reducing Overlay Manager Overhead

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

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

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

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

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

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

Every overlay function call requires initializing the load/execute flag buffer. Here, the function calls are delayed branch calls. The two slots in the delayed branch contain instructions to initialize the flag buffer. Register R0 is set to the value that is placed in the flag buffer, and the value in R0 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.

Memory Management Using Overlays

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;
    R0 = 0;
    dm(prefetch) = R0;
call fft_middle_stages (db);
    R0 = 0;
    dm(prefetch) = R0;
call fft_next_to_last (db);
    R0 = 0;
    dm(prefetch) = R0;
call fft_last_stage (db);
    R0 = 0;
    dm(prefetch) = R0;
```

The next set of instructions represents a load function call.

```
call fft_middle_stages (db);
    /* This function call pre-loads the function */
    /* into the overlay run memory. */
    R0 = 1;
    dm(prefetch) = R0;
    /* Set prefetch flag to 1 to indicate a load. */
```

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

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

```
OVERLAY_GROUP1 {
  OVERLAY_INPUT
  {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_ovl.doj (seg_pmco) )
    PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
  } >ovl_pmco // Overlay to live in section ovl_pmco
  OVERLAY_INPUT
  {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS( Fft_ovl.doj (seg_pmco1) )
    PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
  } >ovl_pmco // 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(seg_pmco3) )
    PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
  } >ovl_pmco // Overlay to live in section ovl_code
  OVERLAY_INPUT
  {
    ALGORITHM(ALL_FIT)
```

Memory Management Using Overlays

```
OVERLAY_OUTPUT(fft_last.ovl)
INPUT_SECTIONS( Fft_ovl.doj(seg_pmco2) )
PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
} >ovl_pmco // Overlay to live in section ovl_code
} > mem_code
```

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

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

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

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

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

Using PLIT{} and Overlay Manager

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

Refer to “[PLIT{}](#)” on page 2-44 for basic syntax information. Refer to “[Memory Management Using Overlays](#)” on page 1-41 for detailed information on overlays.

[Figure 1-13](#) shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker generates special symbols for overlays. These overlay symbols are:

- `_ov_startaddress_#`
- `_ov_endaddress_#`
- `_ov_size_#`
- `_ov_word_size_run_#`
- `_ov_word_size_live_#`
- `_ov_runtimestartaddress_#`

The `#` indicates the overlay number.



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

The two functions in [Figure 1-13](#) are on different overlays. By default, the linker generates PLIT code only when an unresolved function reference is resolved to a function definition in overlay memory.

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

Memory Management Using Overlays

Non-Overlay Memory

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

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

.plt_X: call OM
.plt_Y: call OM

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

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

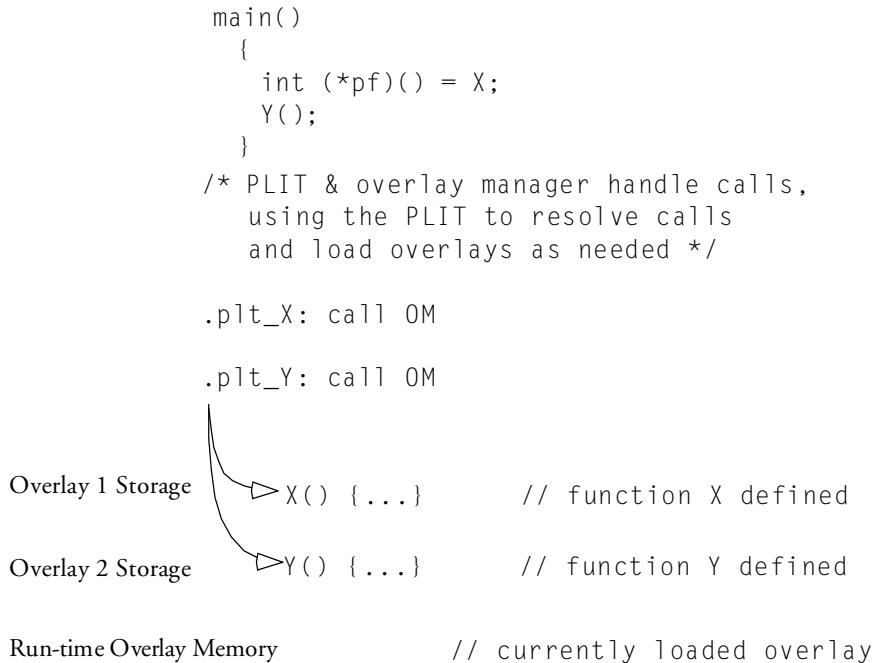


Figure 1-13. PLITs & Overlay Memory; main() Calls to Overlays

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

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

Inter-Overlay Calls

PLITs allow you to resolve inter-overlay calls, as shown in [Figure 1-14 on page 1-65](#). 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.

Code in Non-Overlay Memory

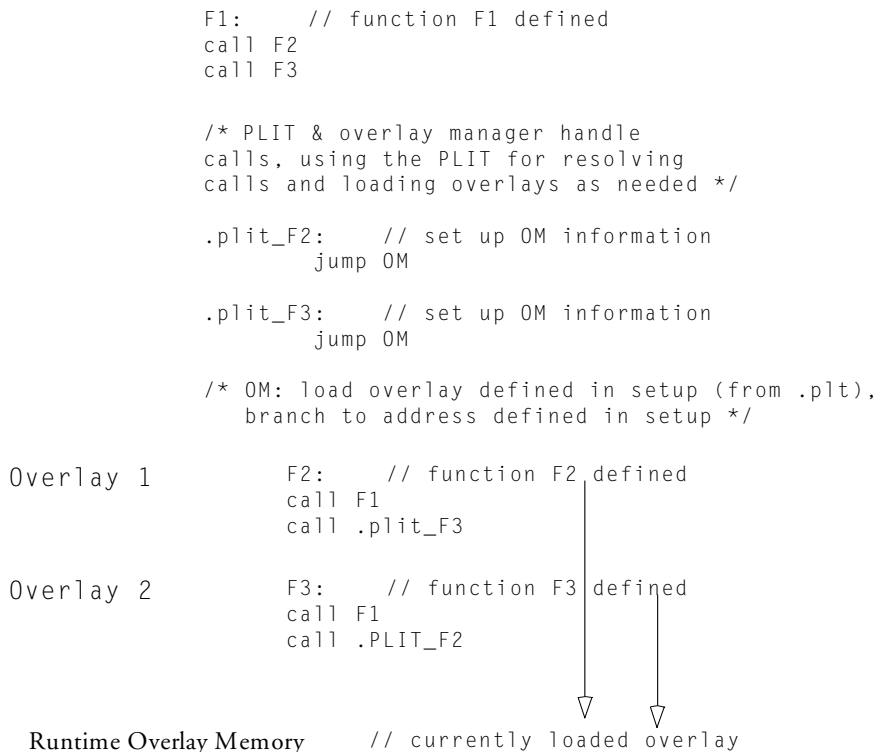


Figure 1-14. PLITs and Overlay Memory – Inter-Overlay Calls

Memory Management Using Overlays

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

Inter-Processor Calls

PLITs resolve inter-processor overlay calls, as shown in [Figure 1-15 on page 1-67](#), 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.

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

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

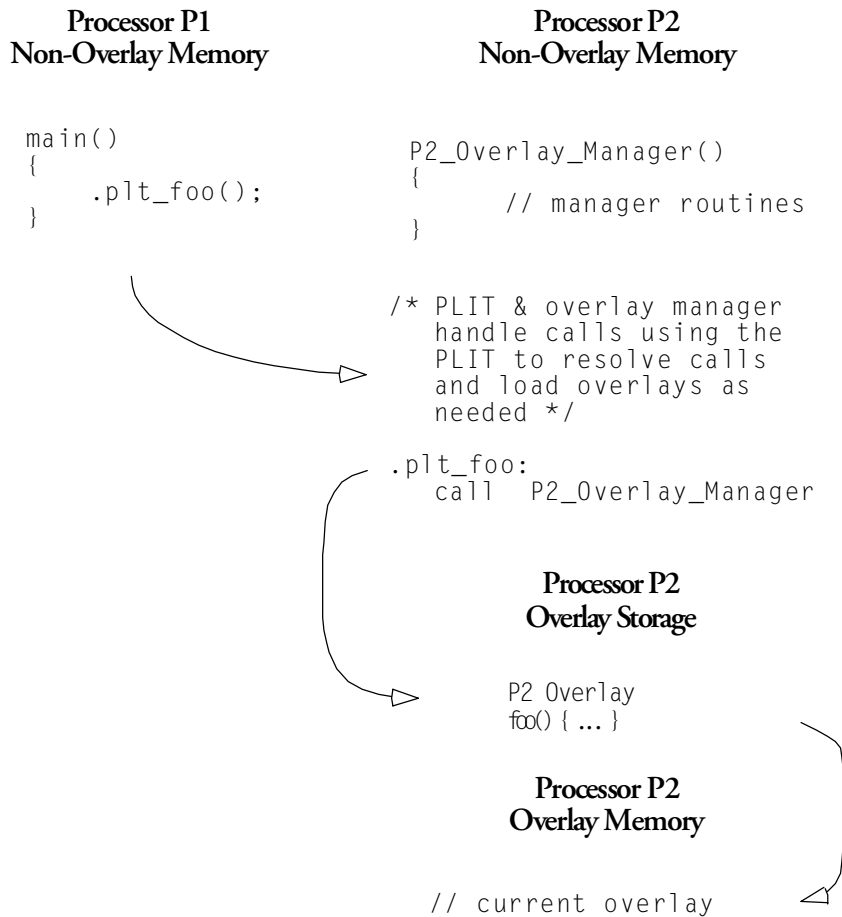


Figure 1-15. PLITs and Overlay Memory – Inter-Processor Calls

2 LINKER DESCRIPTION FILE

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



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

This chapter is an .LDF file reference and includes:

- [“LDF Guide” on page 2-2](#) – Describes Linker Description File syntax, commands, macros, operators, and programming techniques.
- [“LDF Programming Examples” on page 2-61](#) – Provides example .LDF files for various system architectures.

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

LDF Guide

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

This section contains:

- [“.LDF File Overview” on page 2-3](#)
- [“LDF Structure” on page 2-8](#)
- [“LDF Expressions” on page 2-10](#)
- [“LDF Keywords, Commands, and Operators” on page 2-11](#)
- [“LDF Operators” on page 2-13](#)
- [“LDF Macros” on page 2-17](#)
- [“LDF Commands” on page 2-20](#)



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 SHARC DSPs*.

Assembler section declarations in this document correspond to the SHARC DSP family assembler’s `.SECTION` directive.

Refer to [“LDF Programming Examples” on page 2-61](#) and to example DSP programs shipped with VisualDSP++ for example .LDF files supporting typical system models.

.LDF File Overview

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

```
symbol = label
```

```
symbol = function_name
```

```
symbol = variable_name
```

If you neither write an LDF nor import an LDF into your project, VisualDSP++ links the code using a default LDF. The default LDF is based on the processor specified in the VisualDSP++ environment's **Project Options** dialog box. Default .LDF files are packaged with your DSP tool distribution kit in a subdirectory specific to your target processor's family (SHARC). 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 1-19](#) for more information.

Example – Basic .LDF File

[Listing 2-1](#) is an example of a basic .LDF file (formatted for readability). Note the `MEMORY{}` and `SECTIONS{}` commands and refer to [“Notes on Basic .LDF File Example”](#). Other .LDF file examples are provided in [“LDF Programming Examples”](#) on page 2-61.

Listing 2-1. Example .LDF File

```
ARCHITECTURE(ADSP-21065L)
SEARCH_DIR($ADI_DSP\21k\lib)
$OBJECTS = main.doj, $COMMAND_LINE_OBJECTS;

MEMORY {
    /* Define and label system memory */
    /* List of global memory segments */
    mem_isr {TYPE(PM RAM) START(0x08000) END(0x080FF) WIDTH(48)}
    mem_pmco {TYPE(PM RAM) START(0x08100) END(0x087FF) WIDTH(48)}
    mem_pmda {TYPE(PM RAM) START(0x09000) END(0x09FFF) WIDTH(32)}
    mem_dmda {TYPE(DM RAM) START(0x0C000) END(0x0DFFF) WIDTH(32)}
}

PROCESSOR P0 { /* The only processor in the system */
    OUTPUT ( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS{
        dxe_isr { INPUT_SECTIONS ( $OBJECT1(isr_tbl))} > mem_isr
        dxe_pmco { INPUT_SECTIONS ( $OBJECT1(seg_pmco))} > mem_pmco
        dxe_pmda { INPUT_SECTIONS ( $OBJECT1(seg_pmda))} > mem_pmda
        dxe_dmda { INPUT_SECTIONS ( $OBJECT1(seg_dmda))} > mem_dmda
    } /* End of SECTIONS command for processor P0. */
} /* End of PROCESSOR command. */
```

Notes on Basic .LDF File Example

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

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

- `ARCHITECTURE(ADSP-21065L)` specifies the target architecture (processorID). This dictates possible memory widths and address ranges, the register set, and other structural information for use by the debugger, linker, and loader. The target architecture must be installed in VisualDSP++.
- `SEARCH_DIR()` specifies directory paths to be searched for libraries and object files. This example’s argument (`$ADI_DSP\21k\lib`) specifies one search directory. For more information, see [“SEARCH_DIR\(\)” on page 2-52](#).

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

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

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

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

- `$COMMAND_LINE_OBJECTS` ([on page 2-18](#)) is an LDF *command-line macro*, which expands at the linker command line into the list of input files. Each linker invocation from the VisualDSP++ IDDE has a command-line equivalent. In the VisualDSP++ IDDE, `$COMMAND_LINE_OBJECTS` represents the `.DOJ` file of every source file in the VisualDSP++ **Project** window.

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

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

The following examples are functionally identical.

```
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;
```

- The `MEMORY{}` command ([on page 2-26](#)) defines the target system's physical memory and connects the program to the target system. Its arguments partition the memory into memory segments. Each memory segment is assigned a distinct name, memory type (such as `mem_pmco`, `mem_pmda`, or `mem_dmda`), a start and end address (or segment length), and a memory width. These names occupy different

namespaces from input section names and output section names. Thus, a memory segment and an output section may have the same name.

- Each `PROCESSOR{}` command ([on page 2-50](#)) generates a single executable.
- The `OUTPUT()` command ([on page 2-51](#)) produces an executable (.DxE) file and specifies its file name.

In this example, the argument to the `OUTPUT()` command is the `$COMMAND_LINE_OUTPUT_FILE` macro ([on page 2-19](#)). 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 2-53](#)) specifies the placement of code and data in physical memory. The linker maps input sections (in object files) to output sections (in executables), and maps the output sections to memory segments specified by the `MEMORY{}` command.
- The `INPUT_SECTIONS()` statement specifies the object file the linker uses as an input to resolve the mapping to the appropriate memory segment declared in the .LDF file.

For example, the following `INPUT_SECTIONS()` statement directs the linker to place the `isr_tbl` input section in the `dxe_isr` output section and to map it to the `mem_isr` memory segment.

```
dxe_isr{ INPUT_SECTIONS (OBJECT1 (isr_tbl) ) } > mem_isr
```

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

LDF Structure

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

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

Declare common (shared) memory definitions in the global scope before the `PROCESSOR{ }` commands. See [“Command Scoping” on page 2-9](#) 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 1-20](#), [“Placing Code on the Target” on page 1-25](#), and [“LDF Programming Examples” on page 2-61](#).

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 2-17).

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

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

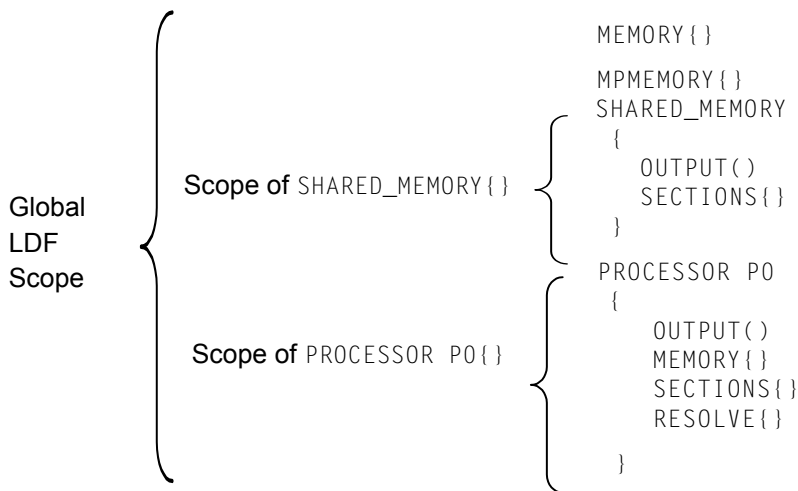


Figure 2-1. LDF Command Scoping Example

LDF Expressions

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

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

Table 2-1. Valid Items in Expressions

Convention	Description
<code>.</code>	Current location counter (a period character in an address expression). See “Location Counter (.)” on page 2-16.
<code>0xnumber</code>	Hexadecimal number (a 0x prefix)
<code>number</code>	Decimal number (a number without a prefix)

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

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

LDF Keywords, Commands, and Operators

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

- [“Miscellaneous LDF Keywords” on page 2-12](#)
- [“LDF Operators” on page 2-13](#)
- [“LDF Macros” on page 2-17](#)
- [“LDF Commands” on page 2-20](#)



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

Table 2-2. LDF File Keywords Summary

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

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

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

Miscellaneous LDF Keywords

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

Table 2-3. Miscellaneous LDF File Keywords

Keyword	Description
FALSE	A constant with a value of 0
TRUE	A constant with a value of 1
XREF	A cross-reference option setting. See “-xref filename” on page 1-40 .

For more information about other .LDF file keywords, see [“LDF Operators” on page 2-13](#), [“LDF Macros” on page 2-17](#) and [“LDF Commands” on page 2-20](#).

LDF Operators

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

Each LDF operator is described next.

ABSOLUTE() Operator

Syntax:

```
ABSOLUTE(expression)
```

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

- A symbolic expression in parentheses; for example:

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

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

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

- A 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 2-16](#)).

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

```
ldf_stack_space = .;
```

can also be written as:

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

- A symbol name

ADDR() Operator

Syntax:

```
ADDR(section_name)
```

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

Example

If an .LDF file defines output sections as,

```
dx_e_pmco
{
    INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
}> mem_pmco

dx_e_dmda
{
    INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
}> mem_seg_dmda
```

the .LDF file may contain the command:

```
ldf_start_dmda = ADDR(mem_seg_dmda)
```

The linker generates the constant `ldf_start_dmda` and assigns it the start address of the `mem_seg_dmda` output section.

DEFINED() Operator

Syntax:

```
DEFINED(symbol)
```

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

Example

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

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

MEMORY_SIZEOF() Operator

Syntax:

```
MEMORY_SIZEOF(segment_name)
```

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

Example

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

```
stackseg {  
    ldf_stack_space = .;  
    ldf_stack_length = MEMORY_SIZEOF(seg_stak);  
} > seg_stak
```

The `stackseg` section is defined to consume the entire `seg_stak` memory segment.

SIZEOF() Operator

Syntax:

```
sizeof(section_name)
```

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

Example

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

```
seg_dmda
{
    INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
    _sizeofdata1 = sizeof(seg_dmda);
} > seg_dmda
```

Location Counter (.)

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

Observe these rules.

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

LDF Macros

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

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

The linker:

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

LDF macros funnel input from the linker command line into predefined macros and provide support for user-defined macro substitutions. Linker macros are available globally in the LDF, regardless of where they are defined. For more information, see [“Command Scoping” on page 2-9](#) and [“LDF Macros and Command-Line Interaction” on page 2-20](#).

 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-Declared Macros

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

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

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

LDF Macros and Command-Line Interaction

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

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

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

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

LDF Commands

Commands in the `.LDF` file (called LDF commands) define the target system and specify the order in which the linker processes output for that system. LDF commands operate within a scope, influencing the operation of other commands that appear within the range of that scope. [For more information, see “Command Scoping” on page 2-9.](#)

The linker supports these LDF commands:

- [“ALIGN\(\)” on page 2-21](#)
- [“ARCHITECTURE\(\)” on page 2-22](#)
- [“ELIMINATE\(\)” on page 2-22](#)
- [“ELIMINATE_SECTIONS\(\)” on page 2-23](#)

- “INCLUDE()” on page 2-23
- “INPUT_SECTION_ALIGN()” on page 2-23
- “KEEP()” on page 2-25
- “LINK_AGAINST()” on page 2-25
- “MAP()” on page 2-26
- “MEMORY{}” on page 2-26
- “MPMEMORY{}” on page 2-29
- “OVERLAY_GROUP{}” on page 2-31
- “PACKING()” on page 2-37
- “PLIT{}” on page 2-44
- “PROCESSOR{}” on page 2-50
- “RESOLVE()” on page 2-52
- “SEARCH_DIR()” on page 2-52
- “SECTIONS{}” on page 2-53
- “SHARED_MEMORY{}” on page 2-58

ALIGN()

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

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

ARCHITECTURE()

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

The command's syntax is:

```
ARCHITECTURE(processorID)
```

The `ARCHITECTURE()` command is case sensitive. Thus, `ADSP-21062` is valid, but `adsp-21062` is not.

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



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

```
linker -proc processorID
```

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

ELIMINATE()

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

ELIMINATE_SECTIONS()

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

INCLUDE()

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

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

INPUT_SECTION_ALIGN()

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

The linker fills holes created by `INPUT_SECTION_ALIGN()` commands with zeros (by default), or with the value specified with the preceding `FILL` command valid for the current scope. See `FILL` under [“SECTIONS{}” on page 2-53](#)

The `INPUT_SECTION_ALIGN()` command is valid only within the scope of an output section. For more information, see [“Command Scoping” on page 2-9](#). For more information on output sections, see the syntax description for [“SECTIONS{” on page 2-53](#).

Example

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

```
SECTIONS
{
    seg_pmco
    {
        INPUT_SECTION_ALIGN(4)

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

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

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

    } >seg_pmco
}
```

KEEP()

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

LINK_AGAINST()

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



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

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

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

where:

- *Pn* is the processor name; for example, P0 or P1.
- *executable_file_names* is a list of one or more executable (.DxE) or shared memory (.SM) files. Be sure to 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 2-52), 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 2-65](#).

MAP()

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

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

MEMORY{}

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

The LDF must contain a `MEMORY{ }` command for global memory on the target system and may contain a `MEMORY{ }` command that applies to each processor’s scope. There is no limit to the number of memory segments you can declare within each `MEMORY{ }` command. [For more information, see “Command Scoping” on page 2-9.](#)

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

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

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

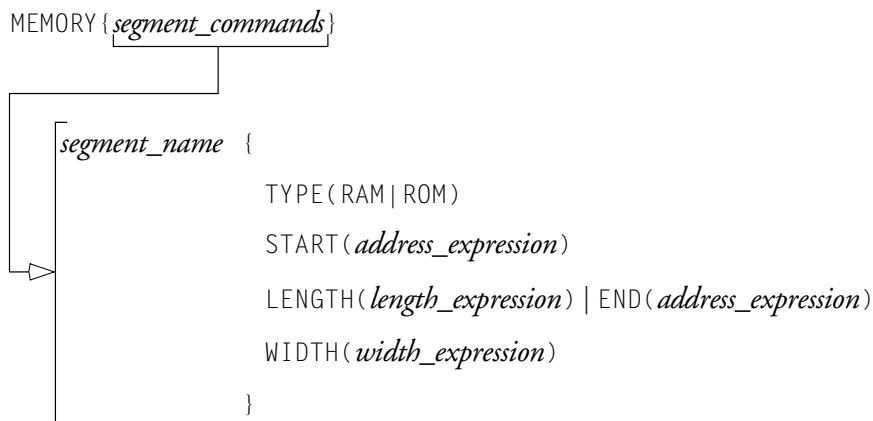


Figure 2-2. Syntax Tree of the MEMORY{} Command

Segment Declarations

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

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

Table 2-4. Parts of a Segment Declaration

Item	Description
<i>segment_name</i>	Identifies the memory region. <i>segment_name</i> must start with a letter, underscore, or point, and may include any letters, underscores, digits, and points, and must not conflict with LDF keywords.
TYPE()	Identifies the architecture-specific type of memory within the memory segment. (Note: not all target processors support all types of memory.) The linker stores this information in the executable for use by other development tools. Specify the functional or hardware locus (RAM or ROM). The RAM declarator specifies segments that need to be booted. ROM segments are not booted; they are executed/loaded directly from off-chip PROM space.
START(<i>address_number</i>)	Specifies the memory segment's start address. The <i>address_number</i> must be an absolute address.
LENGTH(<i>length_number</i>) or END(<i>address_number</i>)	Identifies the length of the memory segment (in words) or specifies the segment's end address. When stating the length, <i>length_number</i> is the number of addressable words within the region or an expression that evaluates to the number of words. When stating the end address, <i>address_number</i> is an absolute address.
WIDTH(<i>width_number</i>)	Identifies the physical width (number of bits) of the on-chip or off-chip memory interface. <i>width_number</i> must be a number. For SHARC DSPs, width must be 32.

MPMEMORY{}

The LDF `MPMEMORY{}` command specifies the offset of each processor's physical memory in a multiprocessor target system. After declaring the processor names and memory segment offsets with the `MPMEMORY{}` command, the linker uses the offsets during multiprocessor linking. Refer to [“Memory Management Using Overlays” on page 1-41](#) for a detailed description of overlay functionality.

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

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

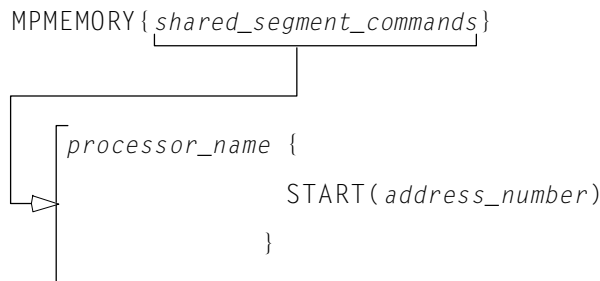


Figure 2-3. `MPMEMORY{}` Command Syntax Tree

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

- *shared_segment_commands*. Contains *processor_name* declarations with a `START{ }` address for each processor's offset in multiprocessor memory. Processor names follow the same rules as any linker label. For more information, refer to [“LDF Expressions” on page 2-10](#).
- *processor_name{placement_commands}*. Applies the *processor_name* offset for multiprocessor linking. Refer to [“PROCESSOR{ }” on page 2-50](#) for more information.

OVERLAY_GROUP{}



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

```
[Warning li2534] More than one overlay group or explicit
OVERLAY_GROUP command is detected in the output section
'seg_pmda'. Create a separate output section for each group
of overlays. Expert Linker makes the change automatically
upon reading the .LDF file.
```

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

Overlays may be *grouped* or *ungrouped*. Use the OVERLAY_INPUT{} command to support ungrouped overlays. Refer to [“Memory Management Using Overlays” on page 1-41](#) for a detailed description of overlay functionality.

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

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

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

```

    }
}

```

Figure 2-4 on page 2-32 demonstrates grouped overlays.

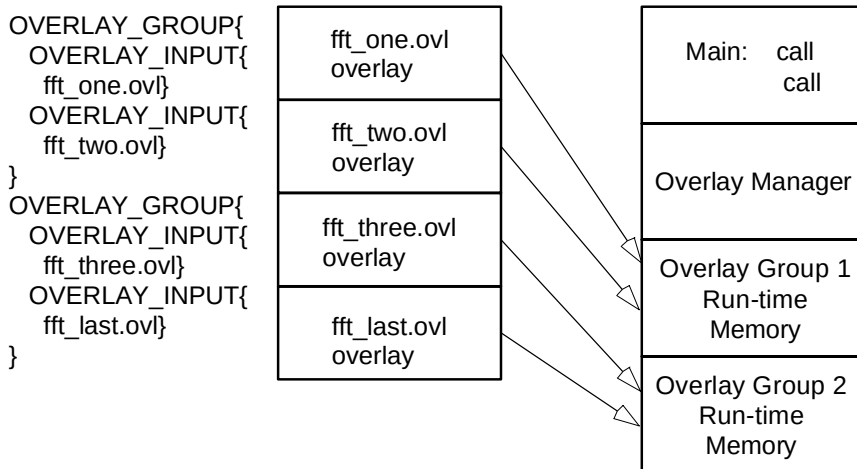


Figure 2-4. Example of Overlays – Grouped

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

Ungrouped Overlay Execution

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



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

Listing 2-2. LDF Overlays – Not Grouped

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

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

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

OVERLAY_INPUT {
    ALGORITHM        ( ALL_FIT )
    OVERLAY_OUTPUT ( fft_three.ovl)
    INPUT_SECTIONS ( Fft_3rd.doj(pm1_code) ) } >ovl_code

OVERLAY_INPUT {
```

LDF Guide

```
ALGORITHM      ( ALL_FIT )  
OVERLAY_OUTPUT ( fft_last.ovl )  
INPUT_SECTIONS ( Fft_last.doj(pm2_code) ) } >ovl_code
```

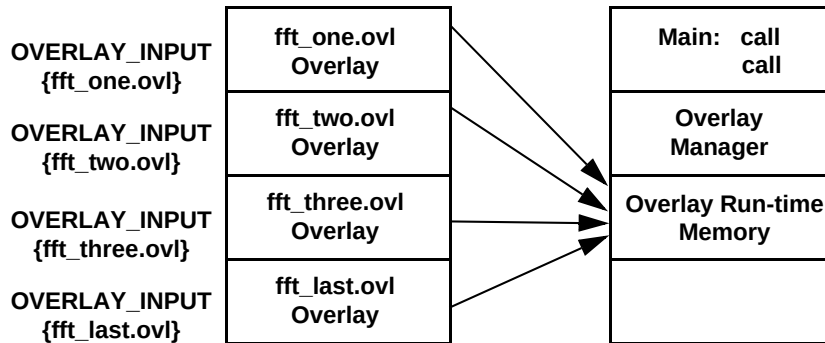



Figure 2-5. Example of Overlays – Not Grouped

Grouped Overlay Execution

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

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 environment (on-chip or off-chip) through several buses. The configuration, placement, and amounts of memory are determined by the application. Specify memory of width(s) and data transfer byte order(s) that suit your needs.

The linker places data in memory according to the constraints imposed by your system's architecture. The LDF's `PACKING()` command specifies the order the linker uses to place bytes in memory. This ordering places data in memory in the sequence the DSP uses as it transfers data.

The `PACKING()` command allows the linker to structure its executable output to be consistent with your installation's memory organization. It can be applied (scoped) on a segment-by-segment basis within the `.LDF` file, with adequate granularity to handle heterogeneous memory configurations. Any memory segment requiring more than one packing command may be divided into homogeneous segments.

Syntax

The syntax of the `PACKING()` command 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).
 - Parameters are whitespace-delimited.
 - The total number of non-null bytes is *number_of_bytes*.
 - If null bytes are included, they are labeled B0.



The first byte is B1 (not B0). The second byte is B2, and so on. For example:

```
PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
```

Non-default use of the `PACKING()` command reorders bytes in executable files (`.DXX`, `.SM`, or `.OVL`), so they arrive at the target in the correct number, alignment, and sequence. To accomplish this task, the command specifies the size of the reordered group, the byte order within the group, and whether and where “null” bytes must be inserted to preserve alignment on the target. The term “null” refers to usage – the target ignores a null byte; the linker sets these bytes to zeros.

The order used to place bytes in memory correlates to the order the DSP may use while unpacking the data when the DSP transfers data from external memory into its internal memory. The processor’s unpacking order can relate to the transfer method. On SHARC DSPs, `PACKING()` applies to the DSPs external port. Each external port buffer contains data packing logic that allows the packing of 8-, 16-, or 32-bit external bus words into 32- or 48-bit internal words. This logic is fully reversible.

Packing in SHARC DSPs

The following information describes how PACKING() may apply in an LDF for your ADSP-21xxx DSP.

 VisualDSP++ comes with the `packing.h` file in the ... 21k\include folder. This file provides macros that define packing commands for use in a Linker Description File (.LDF). The macros support various types of packing for DMA (used in overlays) and for direct external execution. To use these macros, place them in an .LDF file's `SECTIONS{}` command when a `PACKING()` command is needed.

In some DMA modes, SHARC processors unpack three 32-bit words to build two 48-bit instruction words when the processor receives data from 32-bit memory. For example, the unpacked order and storage order (Table 2-5) could apply to a DMA mode.

Table 2-5. DMA Packing Order

Transfer Order (from storage in a 32-bit external memory)	Unpacked Order Two 48-bit internal words (after the third transfer)
B1 and B2 (word 1, bits 47-32) B3 and B4 (word 1, bits 31-16) B11 and B12 (word 2, bits 15-0) B5 and B6 (word 1, bits 15-0) B7 and B8 (word 2, bits 47-32) B9 and B10 (word 2, bits 31-16)	 B1, B2, B3, B4, B5, B6 (word 1, bits 47-0) B7, B8, B9, B10, B11, B12 (word 2, bits 47-0)

The order of unpacked bytes does **not** match the transfer (stored) order. Because the DSP uses two bytes per short word, the above transfer translates into the format in Table 2-6.

Table 2-6. Storage Order vs. Unpacked Order

Storage Order (in 32-bit external memory)	Unpacked Order (two 48-bit internal words)
B1, B2, B3, B4, B11, B12 B5, B6, B7, B8, B9, B10	B1, B2, B3, B4, B5, B6 B7, B8, B9, B10, B11, B12

You specify to the linker how to accommodate DSP-specific byte packing (for example, non-sequential byte order) with the `PACKING()` syntax within the `OVERLAY_INPUT{}` command. The above example's byte ordering translates into the following `PACKING()` command syntax, which supports 48-bit to 32-bit packing over the DSP's external port.

```
PACKING (12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
```

The above `PACKING()` syntax places instructions in an overlay stored in a 32-bit external memory, but is unpacked and executed from 48-bit internal memory.

Refer to `fft_ovly.fft`, which uses a macro that defines the packing. This file is included with the overlay3 example that ships with VisualDSP++.

For SHARC processors, [Table 2-7](#) shows types of packing transfers and the corresponding `PACKING()` syntax. Note that null placeholders are omitted. Bytes indicated with `B0` in the `PACKING()` syntax are placeholders; the linker sets those bytes to zeros in the executable.

Table 2-7. Types of Packing Transfers

Packing Type	Command Syntax
48-bit to 32-bit instruction word	<code>PACKING(12 B1 B2 B3 B4 B11 B12 B5 B6 B7 B8 B9 B10)</code>
48-bit to 16-bit instruction word (optional, not required because byte ordering is sequential)	<code>PACKING(6 B1 B2 B3 B4 B5 B6)</code>

Table 2-7. Types of Packing Transfers

Packing Type	Command Syntax
32-bit to 16-bit instruction word (optional, not required because byte ordering is sequential)	PACKING(4 B1 B2 B3 B4)
8-bit packing (ADSP-21161N)	48-bit to 8-bit PACKING(6 B1 B2 B3 B4 B5 B6) 32-bit to 8-bit PACKING(4 B1 B2 B3 B4) 16-bit to 8-bit PACKING(2 B1 B2) 16-bit to 8-bit PACKING(2 B1 B2)

Overlay Packing Formats

Use the `PACKING()` command when:

- Data and instructions for overlays are executed from external memory (by definition those overlays “live” in external memory)
- The width or byte order of stored data differs from its run-time organization

The linker word-aligns the packing instruction as needed.

Table 2-8 indicates packing format combinations for SHARC DSP DMA overlays available under each of the two operations.

Table 2-8. Packing Formats for SHARC DSP DMA Overlays

Execution Memory type	Storage Memory type	Packing Instruction
32-bit PM	16-bit DM	PACKING(6 B0 B0 B1 B2 B5 B0 B0 B3 B4 B6)
32-bit DM	16-bit DM	PACKING(4 B0 B0 B1 B2 B0 B0 B3 B4 B5)

Table 2-8. Packing Formats for SHARC DSP DMA Overlays

Execution Memory type	Storage Memory type	Packing Instruction
48-bit PM	16-bit DM	PACKING(6 B0 B0 B1 B2 B0 B0 B0 B3 B4 B0 B0 B0 B5 B6 B0)
48-bit DM	32-bit DM	PACKING(12 B1 B2 B3 B4 B0 BB5 B6 B11 B12 B0 B7 B8 B9 B10 B0)

Table 2-9 indicates packing format combinations for ADSP-21161N DSPs' DMA overlays available for storage in 8-bit-wide memory. 8-bit packing is available on ADSP-2106x and ADSP-21160 DSPs during EPROM booting only.

Table 2-9. Additional Packing Formats for DMA Overlays

Execution Memory type	Storage Memory type	Packing Instruction
48-bit PM	8-bit DM	PACKING(6 B0 B0 B0 B0 B1 B0 B0 B0 B2 B0 B0 B0 B3 B0 B0 B0 B0 B4 B0 B0 B0 B0 B5 B0 B0 B0 B0 B6 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0)
32-bit DM	8-bit DM	PACKING(4 B0 B0 B0 B1 B0 B0 B0 B0 B2 B0 B0 B0 B0 B3 B0 B0 B0 B0 B4 B0)
16-bit DM	8-bit DM	PACKING(2 B0 B0 B0 B1 B0 B0 B0 B0 B2 B0)

External Execution Packing

External execution packing commands pack instructions into external memory for direct execution. The only two processors that support packed external execution are the ADSP-21161N and the ADSP-21065L. The ADSP-21161N DSP supports 48-, 32, 16-, and 8-bit-wide external memory. The ADSP-21065L DSP supports 32-bit external memory only.

Table 2-10 and Table 2-11 indicate the packing formats for packed external execution.

Table 2-10. External Execution Packing Formats for ADSP-21161N DSPs

Execution Memory type	Packing Instruction
48 bits in 32 bits	PACKING(6 B1 B2 B3 B4 B0 B0 B0 B0 B5 B6 B0 B0)
48 bits in 16 bits	PACKING(6 B0 B0 B1 B2 B0 B0 B0 B0 B3 B4 B0 B0 B0 B0 B5 B6 B0 B0 B0 B0 B0 B0)
48 bits in 8 bits	PACKING(6 B0 B0 B0 B1 B0 B0 B0 B0 B0 B2 B0 B0 B0 B0 B0 B3 B0 B0 B0 B0 B4 B0 B0 B0 B0 B5 B0 B0 B0 B0 B0 B6 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0)

Table 2-11. External Execution Packing Formats for ADSP-21065L DSPs

Execution Memory type	Packing Instruction
48-bit PM ¹	PACKING(6 B0 B0 B5 B6 B0 B0 B1 B2 B3 B4 B0 B0)

1 LDF memory commands define a “logical” segment rather than a physical (32-bit) segment.

 The packing order is different in these two processors for 32-bit external execution.

Default Packing – No Reordering

If the memory organization matches its run-time environment and will load and run without reordering, the linker uses (implicit) “default” packing. Only non-default packing need be specified in the LDF, though segments without reordering may be labeled as such, to retain segment-specific packing order visibility, and provide convenient locations to change the LDF when you change your target or memory configuration.

PLIT{}

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

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

`PLIT{}` commands provide a template from which the linker generates assembly code when a symbol resolves to a function in overlay memory. The code typically handles a call to a function in overlay memory by calling an overlay memory manager. Refer to [“Memory Management Using Overlays” on page 1-41](#) 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 1-63](#).

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 2-12](#) describes parts of the command.

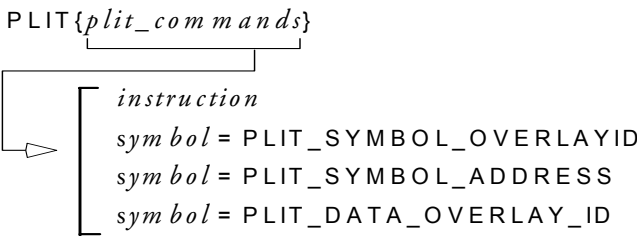


Figure 2-6. Syntax Tree of the `PLIT` Command

Table 2-12. 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.
<code>PLIT_SYMBOL_OVERLAYID</code>	Returns the overlay ID.
<code>PLIT_SYMBOL_ADDRESS</code>	Returns the absolute address of the resolved symbol in run-time memory.

Command Evaluation and Setup

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

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

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

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

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

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

Allocating Space for PLITs

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

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

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

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

PLIT Examples

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

Simple PLIT – States are Not Saved

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

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

```
PLIT
{
    R0 = PLIT_SYMBOL_OVERLAYID;
    R1 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

A PLIT that Saves Register Contents

The following `PLIT{}` command saves the contents of `R0` and `R1` before being overwritten.

```
PLIT
{
    j6 = save_j4;
    [j6+j31] = j4;
    j6 = save_j5;
    [j6+j31] = j5;
    R0 = PLIT_SYMBOL_OVERLAYID;
    R1 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

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

PLIT – Summary

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

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

The overlay manager can also:

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

implement this feature within the PLIT section of your LDF.

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

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

PROCESSOR{}

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

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

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

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

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

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

- `processor_name`

Assigns a name to the processor. Processor names follow the same rules as linker labels. [For more information, see “LDF Expressions” on page 2-10.](#)

- `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 2-9 and “MEMORY{” on page 2-26.](#)

- `PLIT{plit_commands}`

Defines procedure linkage table (PLIT) commands that apply only to this processor. [For more information, see “PLIT{” on page 2-44.](#)

- `SECTIONS{section_commands}`

Defines sections for placement within the executable (.DXE). [For more information, see “SECTIONS{” on page 2-53.](#)

- `RESOLVE(symbol, resolver)`

Ignores any `LINK_AGAINST()` command. For details, see [“RESOLVE{” on page 2-52.](#)

RESOLVE()

Use the LDF `RESOLVE(symbol_name, resolver)` command to ignore a `LINK_AGAINST()` command for a specific symbol. This overrides the search order for a specific variable or label. Refer to the “[LINK_AGAINST\(\)](#)” on [page 2-25](#) for more information.

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



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

SEARCH_DIR()

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

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

Example

```
ARCHITECTURE (ADSP-21062)
MAP (SINGLE-PROCESSOR.MAP)           // Generate a MAP file

SEARCH_DIR( $ADI_DSP\21k\lib; ABC\XYZ )
// $ADI_DSP is a predefined linker macro that expands
// to the VDSP install directory. Search for objects
```

```
// in directory 21k\lib relative to the install directory
// and to the ABC\XYZ directory.
```

SECTIONS{}

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

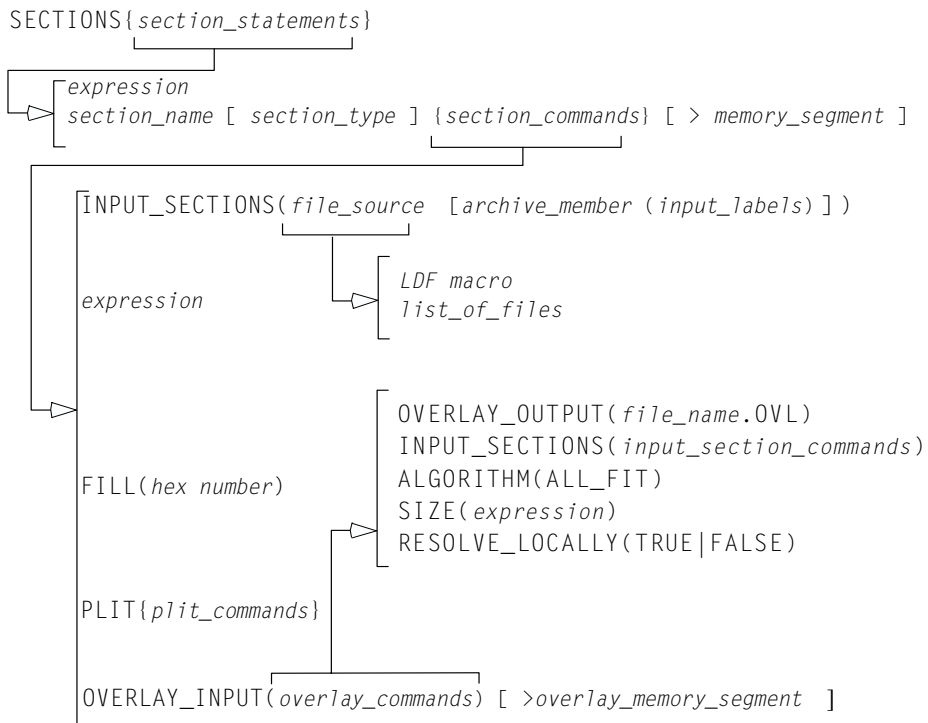


Figure 2-8. Syntax Tree of the SECTIONS{} Command

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

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

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

expressions

or

section_declarations

Use *expressions* to manipulate symbols or to position the current location counter. Refer to [“LDF Expressions” on page 2-10](#).

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

Figure 2-9. Parts of a Section Declaration

Item	Description
<i>section_name</i>	Must start with a letter, underscore, or period and may include any letters, underscores, digits, and points. A <i>section_name</i> must not conflict with any LDF keywords. The special <i>section_name</i> , <code>.PLIT</code> , indicates the procedure linkage table (PLIT) section that the linker generates when resolving symbols in overlay memory. Place this section in non-overlay memory to manage references to items in overlay memory.
<i>section_type</i>	(optional) Assigns an ELF section type. The only valid section type keyword is <code>SHT_NOBITS</code> (section header type no bits). This section type contains uninitialized data, so even if it is large, it can download quickly. Space is allocated but not written. For an example of <code>SHT_NOBITS</code> , see Listing 2-6 on page 2-65 .

Figure 2-9. Parts of a Section Declaration

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

INPUT_SECTIONS()

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

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

expression

In a *section_command*, an *expression* manipulates symbols or positions the current location counter. See [“LDF Expressions” on page 2-10](#) for details.

FILL(hex number)

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

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

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

```
FILL (0x0)
or
FILL (0xb3c0FFFF)
```

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 2-44.](#)

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 for Overlay Memory” on page 2-72.](#)

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 2-55.](#)
- `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{}

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

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

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

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

The syntax for the `SHARED_MEMORY{ }` command appears in [Figure 2-10](#) followed by definitions of its components in [Table 2-13](#).

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 2-13. 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)

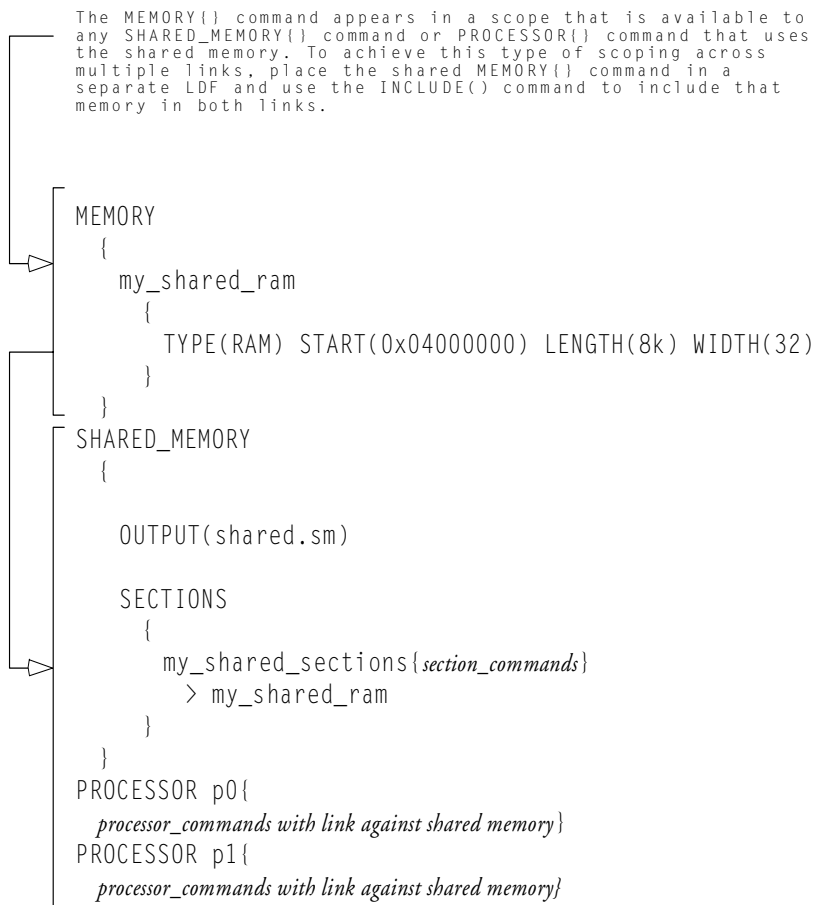


Figure 2-11. LDF Scopes for SHARED_MEMORY{}

LDF Programming Examples

This section shows several typical LDFs. As you modify these examples, refer to the syntax descriptions in [“LDF Commands” on page 2-20](#).

This section provides these examples:

- [“Example – Linking a Single-Processor SHARC System” on page 2-62](#)
- [“Example – Linking Large Uninitialized Variables” on page 2-64](#)
- [“Example – Linking for MP and Shared Memory” on page 2-65](#)
- [“Example – Linking for Overlay Memory” on page 2-72](#)



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 directory.

```
<VisualDSP++ InstallationPath>\21k\Examples
```



A variety of processor-specific default .LDF files come with the development software, providing information about each processor's internal memory architecture. Default .LDF files are in the following directory.

```
<VisualDSP++ InstallationPath>\21k\ldf
```

Example – Linking a Single-Processor SHARC 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 file. Note the following commands in this file:

- `ARCHITECTURE()` defines the processor type.
- `SEARCH_DIR()` adds the `lib` and current working directory to the search path.
- `$OBJ`s and `$LIB`s 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

```
// Link for the ADSP-21062
ARCHITECTURE(ADSP-21062)
SEARCH_DIR ( $ADI_DSP\21k\lib )
MAP (SINGLE-PROCESSOR.MAP) // Generate a MAP file

// $ADI_DSP is a predefined linker macro that expands
// to the VDSP installation directory. Search for objects
// in directory 21k\lib relative to the installation directory
```

```
SEARCH_DIR( $ADI_DSP\21k\lib )

// lib060.dlb is an ADSP-2106x-specific library
// and must precede libc.dlb, the C library
// to link the 2106x-specific routines.

$LIBS = lib060.dlb, libc.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

// 060_hdr.doj is the standard initialization file for 2106x
$OBSJ = $COMMAND_LINE_OBJECTS, 060_hdr.doj;

// 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("21062_memory.h") }

    SECTIONS // Specify the output sections
    {
        INCLUDE( "21062_sections.h" )
    } // 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. This is especially useful for ADSP-21065L DSPs with large amounts of SDRAM.

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    sdram_area;    /* 1Mx32 SDRAM */  
.VAR huge_buffer[0x100000];
```

Listing 2-6. Large Uninitialized Variables: LDF Source

```

ARCHITECTURE(ADSP-21061)
$OBJECTS = $COMMAND_LINE_OBJECTS; // Libraries & objects from
                                     // the command line

MEMORY {
    mem_sdram {
        TYPE(DM RAM) START(0x3000000) END(0x30FFFFFF) WIDTH(32)
        } // 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 {
        sdram_output SHT_NOBITS {
            INPUT_SECTIONS ( $OBJECTS ( sdram_area ) ) } > mem_sdram
        } //end section
    } // end processor P0

```

Example – Linking for MP and Shared Memory

When linking executable files for a multiprocessor system using shared memory, the .LDF file describes the multiprocessor memory offsets, shared memory, each processor's memory, and places code for each processor. Note the following in the example .LDF file in [Listing 2-7 on page 2-66](#):

- The `ARCHITECTURE()` command defines the processor type, which can be one type only.
- The `SEARCH_DIR()` command adds the `lib` and current working directory to the search path.

LDF Programming Examples

- The `$OBJ`s and `$LIBS` macros get object (`.DOJ`) and library (`.DLB`) file input.
- The `MPMEMORY{}` command defines each processor's offset within multiprocessor memory.
- The `SHARED_MEMORY{}` command identifies the output for the shared memory items.
- The `MAP()` command outputs map files.
- The `MEMORY{}` command defines memory for the processors.
- The `PROCESSOR{}` and `SECTIONS{}` commands define each processor and place program sections using memory definitions for each processor's output file.
- The `LINK_AGAINST()` commands resolve symbols within multiprocessor memory.

Listing 2-7. LDF for a Multiprocessor System With Shared Memory

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )

// Allocate multiprocessor memory space with the MPMEMORY{}
// command. The values represent an "offset" that the linker
// uses to resolve undefined symbols in one DXE file to symbols
// defined in another DXE. That is, the offset is added
// to each defined symbol's value.
// For example, PROCESSOR project PSH0 references the undefined
// symbol "buffer", and PROCESSOR project PSH1 defines "buffer"
// at address 0x22000. The linker will "fix up" the reference
// to "buffer" in PSH0's code to address:
// 0x22000 + MPMEMORY(PSH1) = 0x22000 + 0x28000 = 0x2A2000
```



```
MPMEMORY
{
    PSH0 { START (0x200000) }
    PSH1 { START (0x280000) }
}
MEMORY
{
    // This is used for all processors. Alternatively,
    // a PROCESSOR can describe its own memory.

    mem_rth {TYPE(PM RAM) START(0x20000) END(0x200FF) WIDTH(48)}
    mem_init {TYPE(PM RAM) START(0x20100) END(0x201FF) WIDTH(48)}
    mem_pmco {TYPE(PM RAM) START(0x20200) END(0x249FF) WIDTH(48)}
    mem_pmda {TYPE(DM RAM) START(0x24A00) END(0x24FFF) WIDTH(32)}
    mem_dmda {TYPE(DM RAM) START(0x28000) END(0x2BFFF) WIDTH(32)}
    mem_heap {TYPE(DM RAM) START(0x2E000) END(0x2EFFF) WIDTH(32)}
    mem_stak {TYPE(DM RAM) START(0x2F000) END(0x2FFFF) WIDTH(32)}
}

$LIBRARIES = lib060.dlb, libc.dlb;
    // Three link projects are specified in one LDF file.
    // The first link project is a shared memory link project
    // against which the PROCESSOR projects are linked.

SHARED_MEMORY {
    // The file containing the shared data buffers
    // is defined in "shared.c".

$SHARED_OBJECTS = shared.doj;

    // The output name of this shared object is used
    // subsequently in the LINK_AGAINST command
    // of the PROCESSOR projects.
```

LDF Programming Examples

```
OUTPUT(shared.sm)
```

```
// shared.c has data declarations only. There is no need
// to specify any output other than "seg_dmda".
```

```
SECTIONS {
sec_dmda {
    INPUT_SECTIONS ($SHARED_OBJECTS(seg_dmda))
    } >mem_dmda
    } // end shared sections
} //end shared memory
```

```
// The second link project described in this .LDF file
// is a DXE project which will be linked
// against the SHARED link project defined above.
```

```
PROCESSOR_PSH0 {
    $PSH0_OBJECTS = psh0.doj, 060_hdr.doj;
    LINK_AGAINST(shared.sm)
    OUTPUT (psh0.dxe)

    SECTIONS {
dx_e_pmco{ INPUT_SECTIONS(
    $PSH0_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco)
    } >mem_pmco
dx_e_pmda{ INPUT_SECTIONS(
    $PSH0_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda)
    } >mem_pmda
dx_e_dmda{ INPUT_SECTIONS(
    $PSH0_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda)
    } >mem_dmda
dx_e_init{ INPUT_SECTIONS(
    $PSH0_OBJECTS(seg_init) $LIBRARIES(seg_init)
    } >mem_init
```

```
dxerth { INPUT_SECTIONS(  
    $PSH0_OBJECTS(seg_rth) $LIBRARIES(seg_rth)  
    } >mem_rth  
stackseg {    // Allocate a stack for the application.  
    ldf_stack_space = .;  
    ldf_stack_length = 0x2000;  
    } >mem_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;  
    } >mem_heap  
  
} // end PSH0 sections  
} // end PSH0 processor  
  
// The third and final link project defined in this LDF file  
// is another DXE project and will be linked against  
// both the SHARED project and the PSH0 DXE project.  
  
PROCESSOR_PSH1 {  
    $PSH1_OBJECTS = psh1.doj, 060_hdr.doj;  
    LINK_AGAINST(shared.sm, psh0.dxe)  
    OUTPUT (psh1.dxe)  
  
    SECTIONS {  
        dxepmco{ INPUT_SECTIONS(  
            $PSH1_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco)  
            } >mem_pmco  
        dxepmda{ INPUT_SECTIONS(  
            $PSH1_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda)  
            } >mem_pmda  
        dxedmda{ INPUT_SECTIONS(  
            $PSH1_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda)  
            } >mem_dmda  
    }
```

LDF Programming Examples

```
        $PSH1_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda)
    } >mem_dmda
dx_init { INPUT_SECTIONS(
        $PSH1_OBJECTS(seg_init) $LIBRARIES(seg_init)
    } >mem_init
dx_rth { INPUT_SECTIONS(
        $PSH1_OBJECTS(seg_rth) $LIBRARIES(seg_rth)
    } >mem_rth
stackseg {    // Allocate a stack for the application.
    ldf_stack_space = .;
    ldf_stack_length = 0x2000;
    } >mem_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;
    } >mem_heap

// The following definition sections are needed only
// for pre-VisualDSP compatibility with COFF objects.
.coff.stringstab {INPUT_SECTIONS(
    $PSH1_OBJECTS(.coff.stringstab)
    $LIBRARIES(.coff.stringstab))
}
.coff.SDB {INPUT_SECTIONS(
    $PSH1_OBJECTS(.coff.SDB)
    $LIBRARIES(.coff.SDB))
}
.lnno.seg_pmco {INPUT_SECTIONS(
    $PSH1_OBJECTS(.lnno.seg_pmco)
    $LIBRARIES(.lnno.seg_pmco))
}
} // end PSH1 sections
} // end PSH1 processor
```

Reflective Semaphores

Semaphores may be used in multiprocessor (MP) systems to permit processors to share resources such as memory or I/O. A semaphore is a flag that can be read and written by any of the processors sharing the resource. A semaphore's value indicates when the processor can access the resource. *Reflective semaphores* permit communication among processors that share a multiprocessor memory space.

Use broadcast writes to implement reflective semaphores in an MP system. Broadcast writes allow simultaneous transmission of data to all the SHARC DSPs in an MP system. The master processor can broadcast writes to the same memory location or IOP register on all the slaves. During a broadcast write, the master also writes to itself unless the broadcast is a DMA write.

Broadcast writes can also be used to simultaneously download code or data to multiple processors.

Bus lock can be used in combination with broadcast writes to implement reflective semaphores in an MP system. The reflective semaphore should be located at the same address in internal memory (or IOP register) of each SHARC DSP.

SHARC DSPs have a "broadcast" space. Use .LDF file (or header files) to define a memory segment in this space, just as in internal memory or any processor MP space. The broadcast space aliases internal space, so if there is a memory segment defined in the broadcast space, the .LDF file cannot have a memory segment at the corresponding address in the internal space (or in the MP space of any processor). Otherwise, the linker generates an error indicating that the memory definition is not valid.

To check the semaphore, each SHARC DSP reads from its own internal memory. Any object in the project can be mapped to an appropriate memory segment defined in the broadcast space for use as a reflective semaphore. If an object defining symbol `SemA` is mapped to a broadcast space, when the program writes to `SemA`, the written value appears at the

LDF Programming Examples

aliased internal address of each processor in the cluster. Each processor may read the value using `SemA`, or read it from internal memory by selecting (`SemA-0x380000`), thus avoiding bus traffic.

To modify the semaphore, a SHARC DSP requests bus lock and then performs a broadcast write to the semaphore address (for example, `SemA`).

 The DSP should read the semaphore before modifying it to verify that another processor has not changed it.

For more information on semaphores, refer to your DSP's *Hardware Reference* manual.

Example – Linking for Overlay Memory

When you link executable files for an overlay memory system, the `.LDF` file describes the overlay memory, the processors that use the overlay memory, and each processor's unique memory. The `.LDF` file places code for each processor and the `.PLIT` section.

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )

MAP(overlay.map)

/*
This simple overlay example uses internal memory as overlay
storage memory. Typically, overlays are never stored
in internal memory, but are loaded there at runtime.
*/

MEMORY {
    mem_rth   {TYPE(PM RAM) START(0x20000) END(0x200FF) WIDTH(48)}
    mem_init  {TYPE(PM RAM) START(0x20100) END(0x201FF) WIDTH(48)}
    mem_pmco  {TYPE(PM RAM) START(0x20200) END(0x20723) WIDTH(48)}
    mem_ovly  {TYPE(PM RAM) START(0x20724) END(0x23FFF) WIDTH(48)}
```

```

mem_pmda {TYPE(PM RAM) START(0x26000) END(0x27FFF) WIDTH(32)}
mem_dmda {TYPE(DM RAM) START(0x2A000) END(0x2BFFF) WIDTH(32)}
mem_heap {TYPE(DM RAM) START(0x2E000) END(0x2EFFF) WIDTH(32)}
mem_stak {TYPE(DM RAM) START(0x2F000) END(0x2FFFF) WIDTH(32)}
}

/* mem_pmco is the “run” memory segment */
/* mem_ovly is the “live” memory segment */

// Processor and application-specific assembly language
// instructions. One instance of these instructions is
// generated for each symbol resolved in overlay memory.

PLIT {
    // Each of five instructions are duplicated
    // for each symbol in an overlay.
    R0 = PLIT_SYMBOL_OVERLAYID;
    // Assigns overlay ID of the resolved symbol to R0.
    R1 = PLIT_SYMBOL_ADDRESS;
    // Assigns “run” address of resolved symbol to R1.
    dm(_overlayID) = R0;
    dm(_pf) = R1;
    JUMP _OverlayManager;
}

$LIBRARIES = lib060.dlb, libc.dlb;
$OBJECTS = 060_hdr.doj;

PROCESSOR P0 {
    $P0_OBJECTS = main.doj, manager.doj;

    OUTPUT(mgrovly.dxe)

    SECTIONS {
        // .text output section

```

LDF Programming Examples

```
seg_pmco {
  INPUT_SECTIONS(
    $PO_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco) )

  // Specify the first overlay. This overlay is stored
  // in the memory defined by "mem_ovly". It runs in the
  // memory space defined by "mem_pmco".
  OVERLAY_INPUT {

    // The output archive file (overlay1.olv) contains the code
    // and symbol table for this overlay.
    OVERLAY_OUTPUT (overlay1.ovl)

    // Take the code from overlay1.doj only. If there is data
    // that this code needs, it must be the INPUT of a data
    // overlay or the INPUT to non-overlay data memory.
    INPUT_SECTIONS (overlay1.doj (program) )

    // Tell the linker that all code must fit
    // into the "run" memory all at once. Other ALGORITHM
    // commands provide more optimized overlay support.

    ALGORITHM( ALL_FIT)

  } > mem_ovly

  // This is the second overlay. Note that the OVERLAY_INPUT
  // commands must be contiguous in the LDF file to occupy
  // the same run-time memory.

  OVERLAY_INPUT {
    OVERLAY_OUTPUT (overlay2.ovl )
    INPUT_SECTIONS (overlay2.doj (seg_pmco) )
  } > mem_ovly
```



```
} > dxe_pmco

// Instructions generated by the linker in the .plit section
// must be placed in non-overlay memory.
// Here is the one-and-only specification
// telling the linker where to place these instructions.

.plit {
    // linker inserts instructions here.
    } > mem_pmco

dxe_pmda {
    INPUT_SECTIONS(
        $P0_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda) )
    } > mem_pmda

dxe_dmda {
    INPUT_SECTIONS(
        $P0_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda) )
    } > mem_dmda

seg_init {
    INPUT_SECTIONS(
        $P0_OBJECTS(seg_init) $LIBRARIES(seg_init) )
    } > mem_init

dxe_rth {
    INPUT_SECTIONS(
        $P0_OBJECTS(seg_rth) $LIBRARIES(seg_rth) )
    } > mem_rth

stackseg {    // Allocate a stack for the application.
    ldf_stack_space = .;
    ldf_stack_length = 0x2000;
```

LDF Programming Examples

```
    } > mem_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;
} > mem_heap

}

}
```

3 EXPERT LINKER

The linker (`linker.exe`) combines object files into a single executable object module. Using the linker, you can create a new Linker Description File (LDF), modify an existing LDF, and produce an executable file(s). The linker is described in Chapter 1, “[Linker 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++ .LDF file format by providing a visualization capability enabling new users to take immediate advantage of the powerful LDF format flexibility.



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

This chapter contains:

- “[Expert Linker Overview](#)” on page 3-2
- “[Launching the Create LDF Wizard](#)” on page 3-4
- “[Expert Linker Window Overview](#)” on page 3-9
- “[Using the Input Sections Pane](#)” on page 3-10
- “[Using the Memory Map Pane](#)” on page 3-16
- “[Managing Object Properties](#)” on page 3-42

Expert Linker Overview

Expert Linker is a graphical tool that allows you to:

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

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

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

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

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

The Expert Linker window appears.

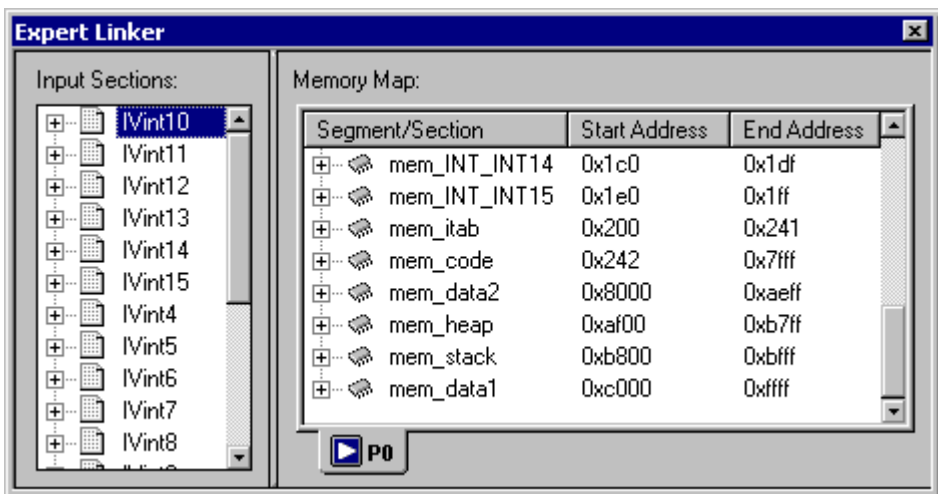


Figure 3-1. Expert Linker Window

Launching the Create LDF Wizard

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

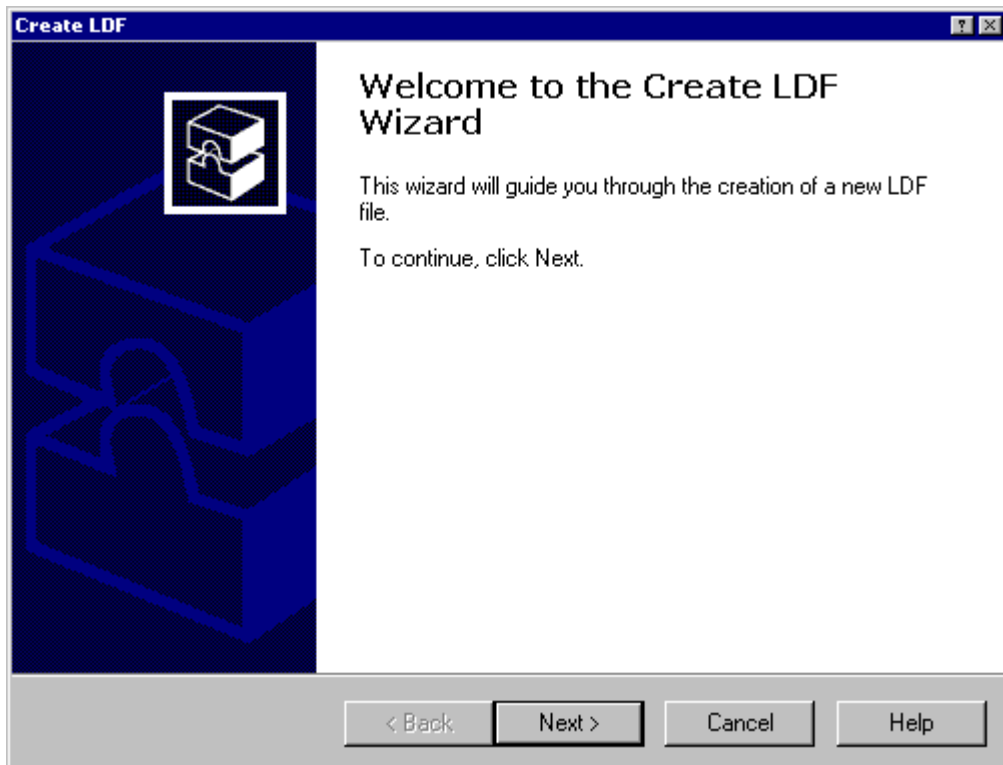


Figure 3-2. Welcome Page of the Create LDF Wizard

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

Step 1: Specifying Project Information

The first wizard window is displayed.

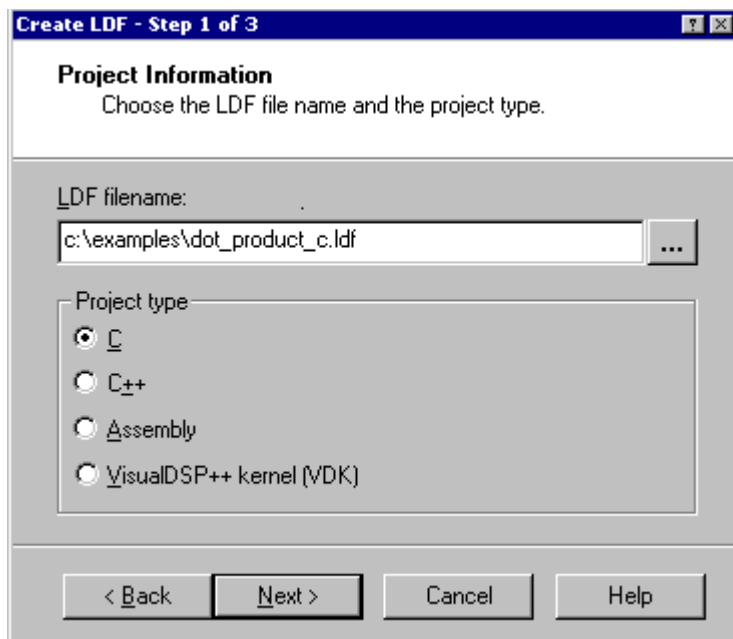


Figure 3-3. Selecting File Name and Project Type

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

Launching the Create LDF Wizard

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

Press **Next**.

Step 2: Specifying System Information

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

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

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

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

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

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

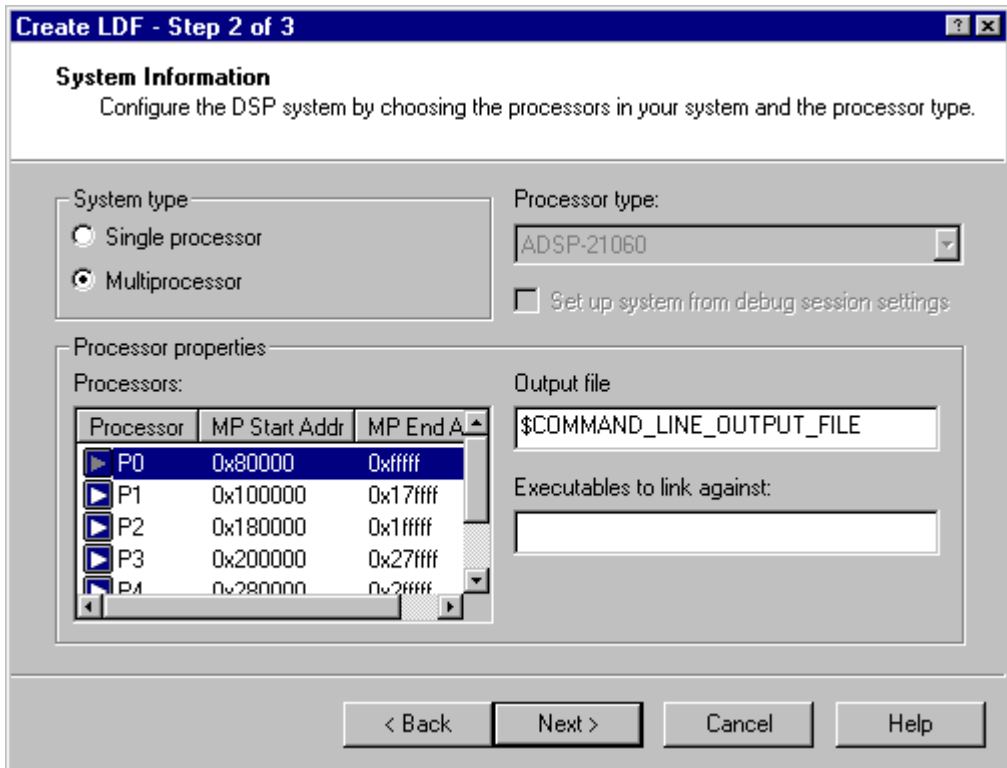


Figure 3-4. Selecting System and Processor Types

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



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

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

Step 3: Completing the LDF Wizard

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

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

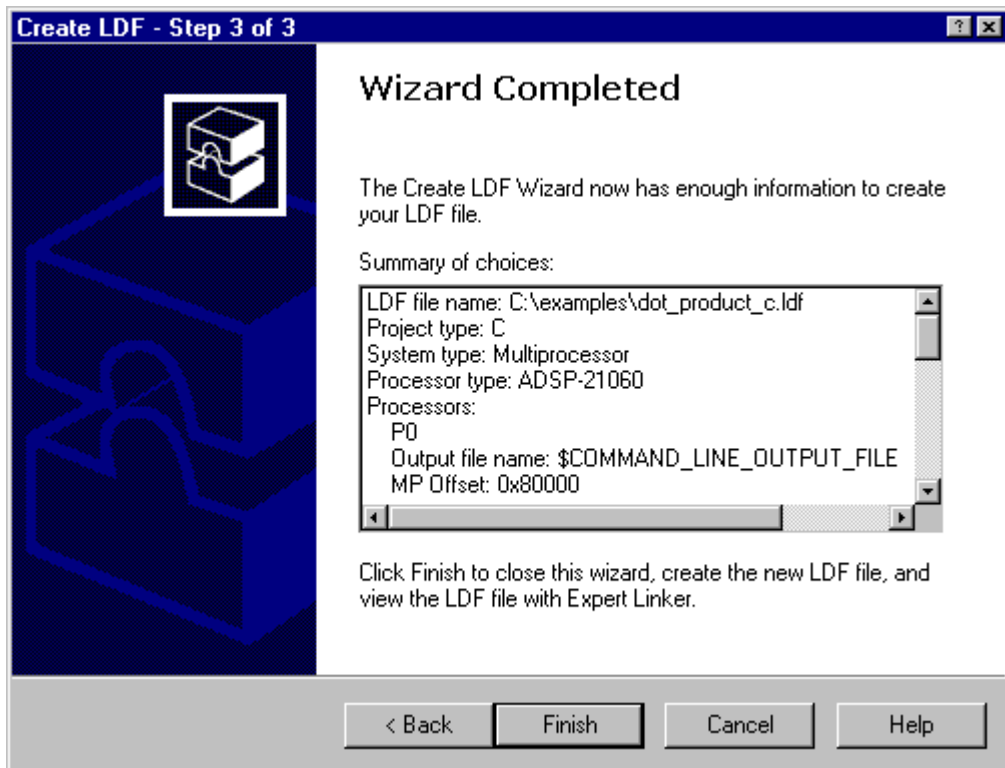


Figure 3-5. Wizard Completed Page of the Create LDF Wizard

Expert Linker Window Overview

The Expert Linker window contains two panes:

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

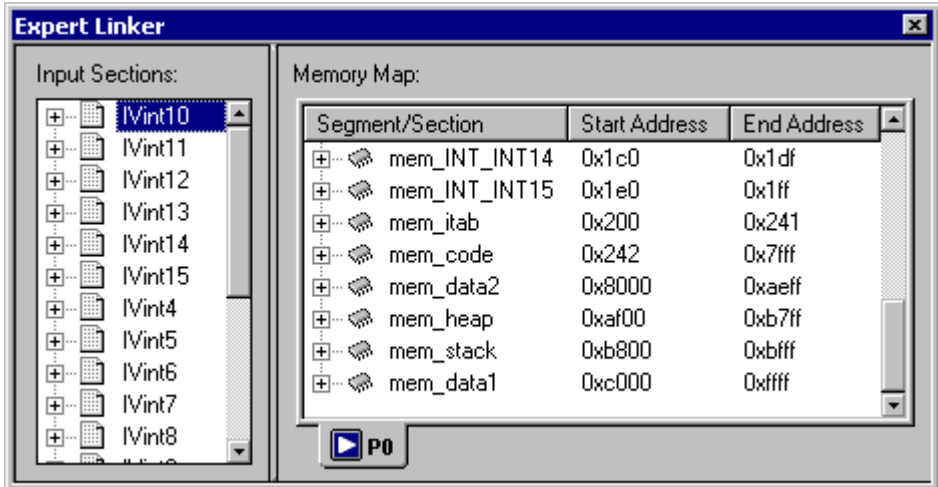


Figure 3-6. Expert Linker Window

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

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

Using the Input Sections Pane

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

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

Using the Input Sections Pane

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

Using the Input Sections Menu

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

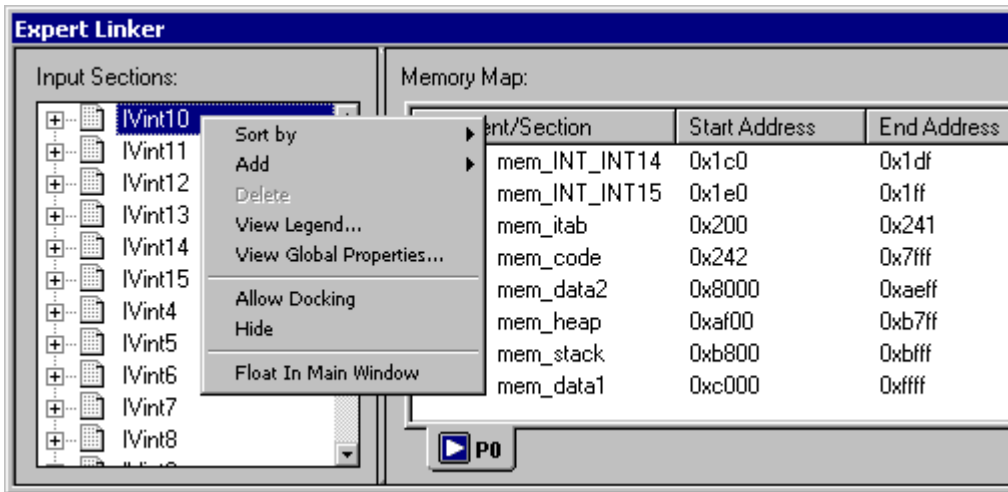


Figure 3-7. Input Sections Right-Click Menu

The main menu functions include:

- **Sort by** – Sorts objects by input sections or LDF macros. These selections are mutually exclusive.
- **Add** – Adds input sections, object/library files, and LDF macros. Appropriate menu selections are grayed out if you right-click on a position (area) in which you cannot create a corresponding object.

You can create an input section as a shell, without object/library files or LDF macros in it. You can even map this section to an output section. However, input sections without data are grayed out.

- **Delete** – Deletes the selected object (input section, object/library file, or LDF macro).
- **View Legend** – Displays the **Legend** dialog box showing icons and colors used by the Expert Linker.

Using the Input Sections Pane

- **View Section Contents** – Opens the **Section Contents** dialog box, which displays the section contents of the object file, library file, or .DXE file. This command is available only after you link or build the project and then right-click on an object or output section.
- **View Global Properties** – Displays the **Global Properties** dialog box that provides the map file name (of the map file generated after linking the project) as well as access to various processor and setup information (see [Figure 3-32 on page 3-43](#)).

Mapping an Input Section to an Output Section

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

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

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

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

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

Viewing Icons and Colors

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

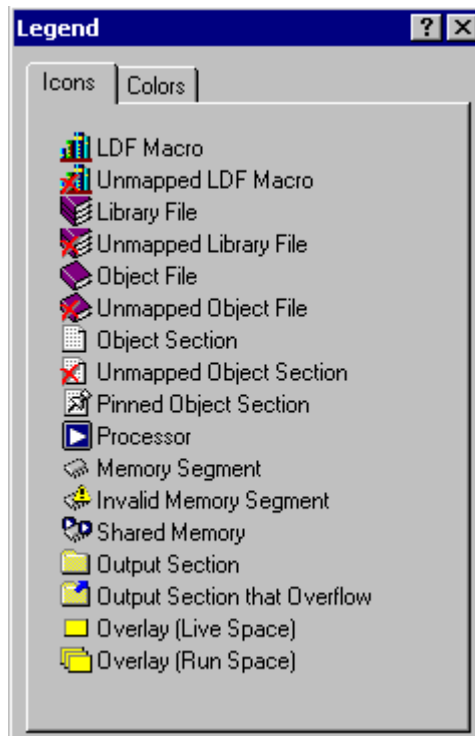


Figure 3-8. Legend Dialog Box – Icons Page



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

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

Using the Input Sections Pane

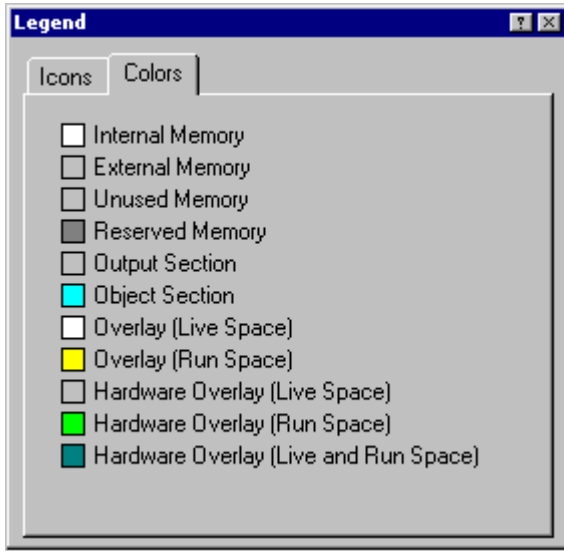


Figure 3-9. Legend Dialog Box – Colors Page

To change a color:

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

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

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

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

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



Figure 3-10. Selecting Colors

Sorting Objects

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

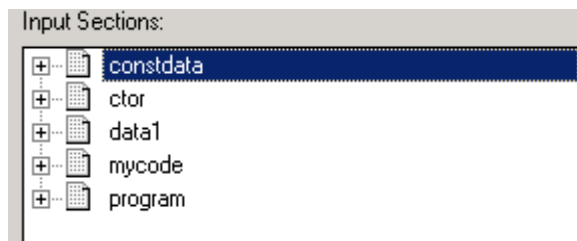


Figure 3-11. Expert Linker Window – Sorted by Input Sections

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



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

Using the Memory Map Pane

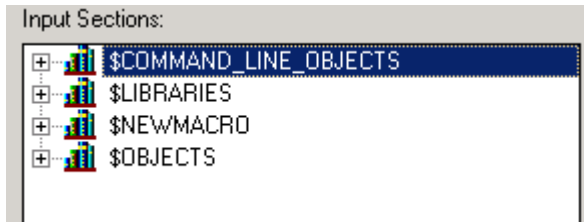


Figure 3-12. Expert Linker Window – Sorted by LDF Macros

Using the Memory Map Pane

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

For TigerSHARC DSPs, the combo box (located to the right of the **Memory Map** label) is not available.

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

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

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

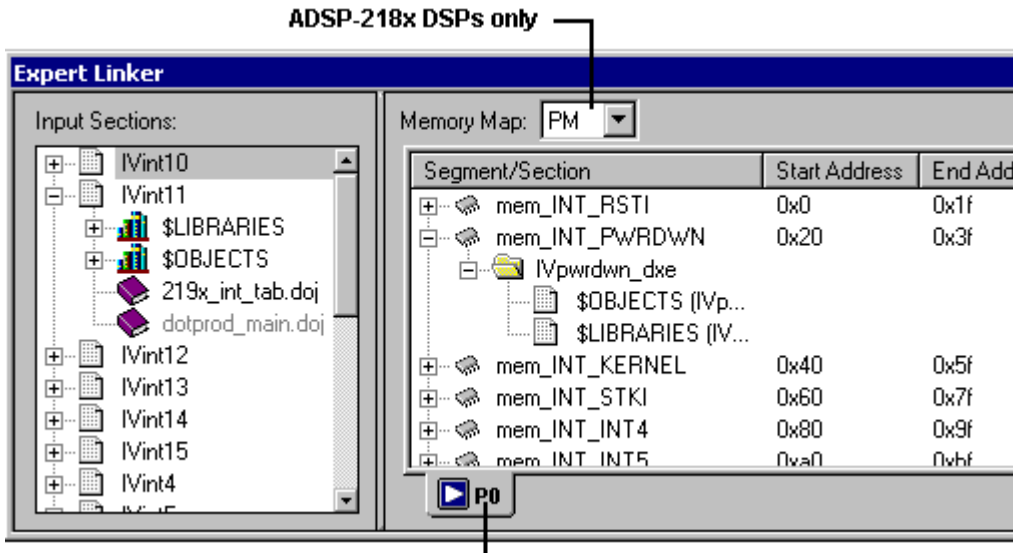


Figure 3-13. Expert Linker Window – Memory Map

Invalid Memory Segment Notification:

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

Using the Context Menu

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

Using the Memory Map Pane

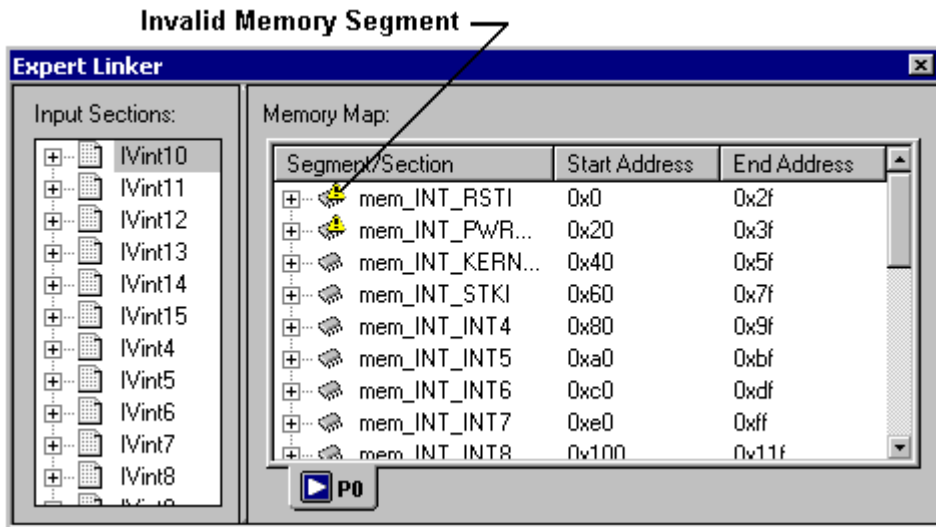


Figure 3-14. Memory Map With Invalid Memory Segments

View Mode

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

View

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

New

- **Memory Segment** – Allows you to specify the name, address range, type, size, and so on for memory segments you want to add.
- **Output Section** – Adds an output section to the selected memory segment. (Right-click on the memory segment to access this command.) If you do not right-click on a memory segment, this option is disabled.

The options are: **Name**, **Overflow** (name of output section to catch overflow), **Packing**, and **Number of bytes** (number of bytes to be reordered at one time).

- **Shared Memory** – Adds a shared memory to the memory map.
- **Overlay** – Invokes a dialog box that allows you to add a new overlay to the selected output section or memory segment. The selected output section is the new overlay's run space (see [Figure 3-43 on page 3-60](#)).

Delete – Deletes the selected object.

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

View Section Contents – Available only after linking or building the project and then right-clicking on an input or object section. This view invokes a dialog box that displays the contents of the input or output section (see [Figure 3-28 on page 3-38](#)).

View Symbols – Available only after linking the project and then right-clicking on a processor, overlay, or input section. Invokes a dialog box that displays the symbols for the project, overlay, or input section (see [Figure 3-31 on page 3-41](#)).

Using the Memory Map Pane

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

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

View Global Properties – Displays a **Global Properties** dialog box that lists the map file generated after linking the project. It also provides access to some processor and setup information (see [Figure 3-32 on page 3-43](#)).

Tree View Memory Map Representation

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

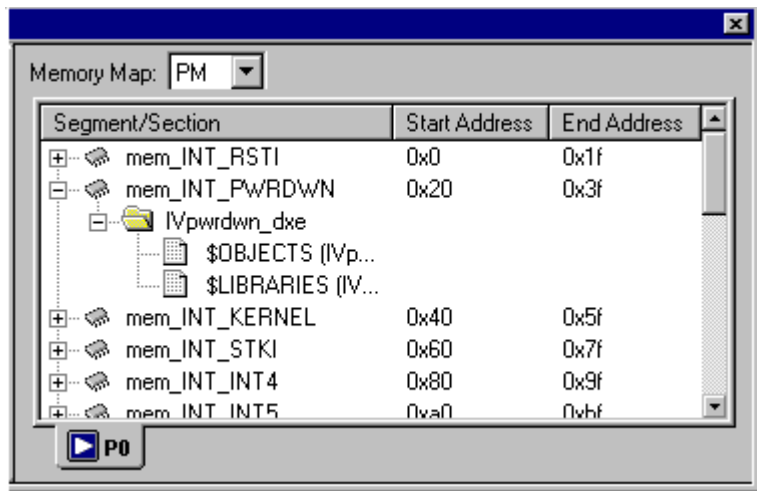


Figure 3-15. Expert Linker Window – Memory Map

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

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

Graphical View Memory Map Representation

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

View the memory map from two perspectives—pre-link view and post-link view (see [“Specifying Pre- and Post-Link Memory Map View” on page 3-30](#)).

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

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

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

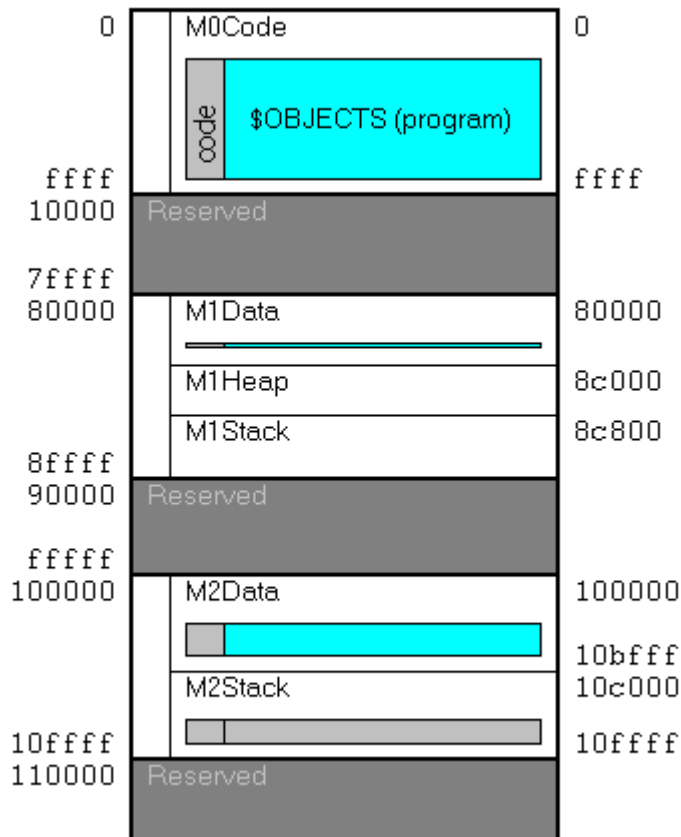


Figure 3-16. Graphical Memory Map Representation

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

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

Using the Memory Map Pane

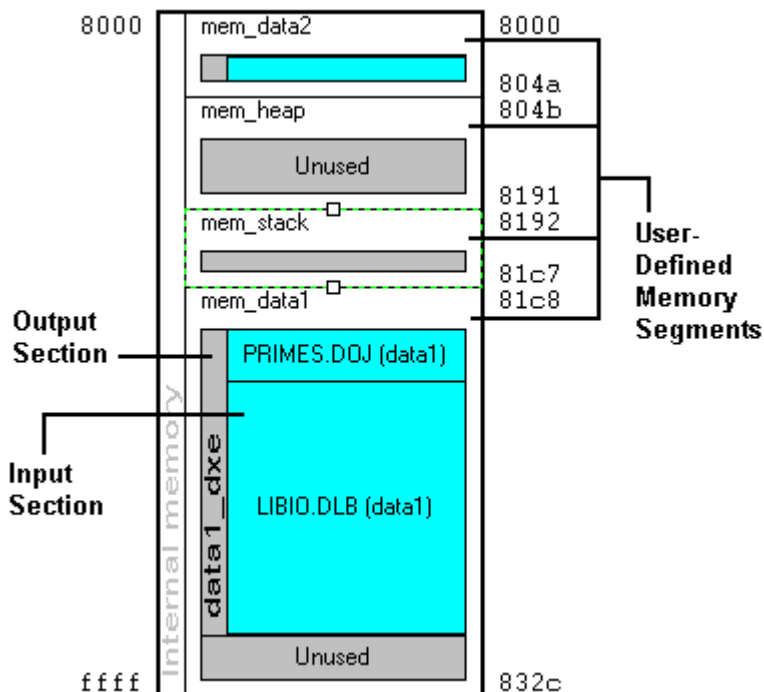


Figure 3-17. Viewing Sections and Segments in Memory Map

Three buttons at the top right of the **Memory Map** pane permit zooming. If there is not enough room to display the memory map when zoomed in, horizontal and/or vertical scroll bars allow you to view the entire memory map (for more information, see [“Zooming In and Out on the Memory Map”](#) on page 3-30).

You can drag and drop any object except memory segments. See [Figure 3-19](#).

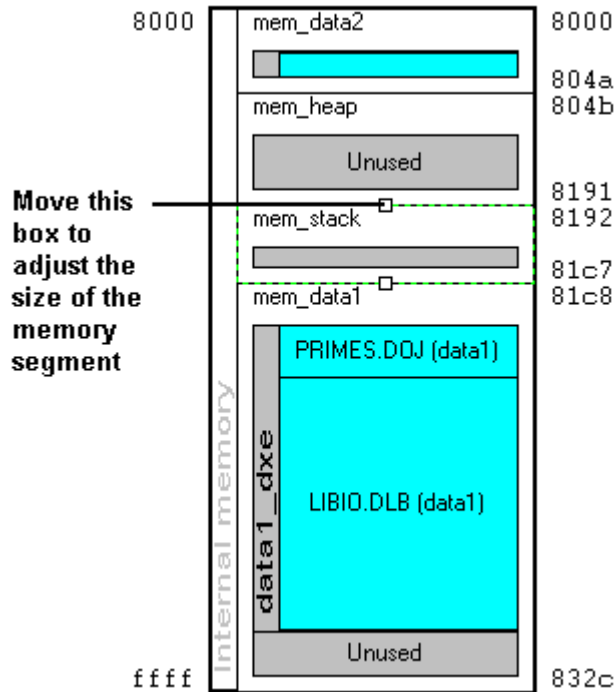


Figure 3-18. Adjusting the Size of a Memory Segment

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

Using the Memory Map Pane

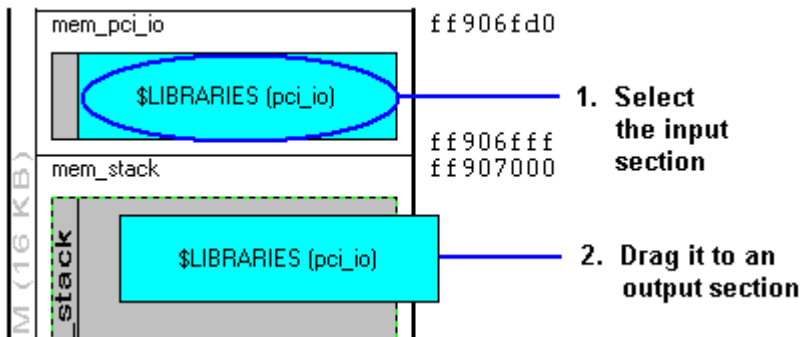


Figure 3-19. Dragging and Dropping an Object

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



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

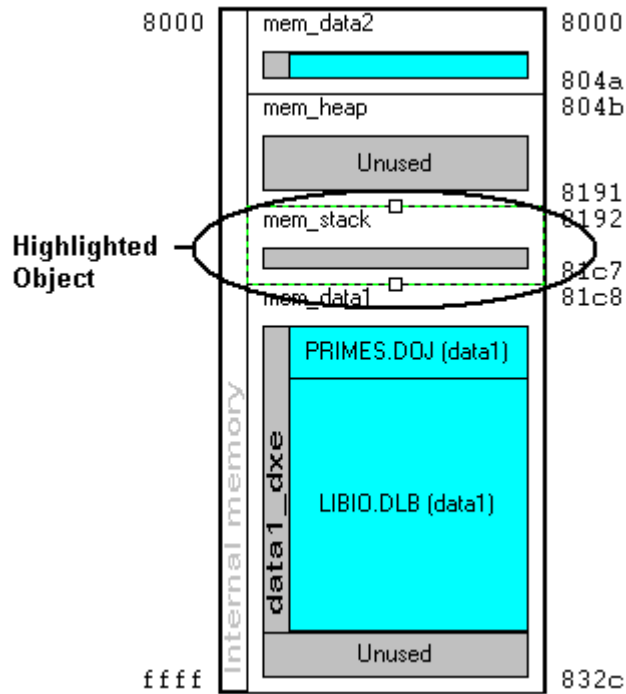


Figure 3-20. A Highlighted Memory Segment in the Memory Map

Using the Memory Map Pane

Unavailable Memory Regions

For SHARC DSPs, the graphical memory map may display “Unavailable for 32-bit Memory” regions. Refer to [Figure 3-21](#).

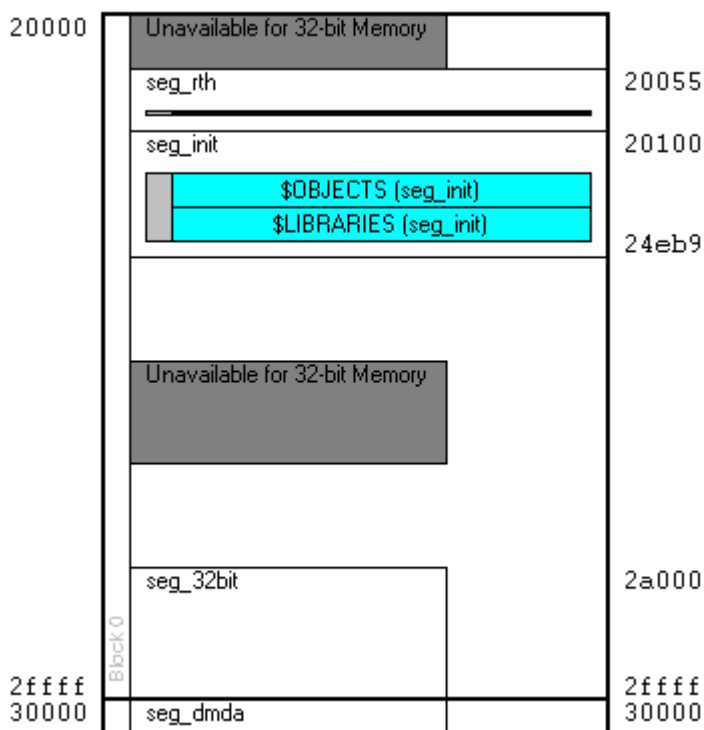


Figure 3-21. Unavailable Memory Regions

Two situations cause unavailable memory regions to appear:

- An unallocated memory exists between the start of an internal memory block and the first 48-bit memory segment. You are restricted from creating a 32-bit memory in this area because a 32-bit memory segment cannot precede a 48-bit memory segment.
- Memory is not available for 32-bit memory because part of the memory is occupied by a 48-bit memory segment. This occurs when a 48-bit memory segment ends on an odd-numbered column.

Regions labeled “Unavailable for 32-bit Memory” limit a 32-bit memory segment can be resized in the graphical memory map. For example, if you “grab” the top border of seg_32bit in [Figure 3-21](#) and drag it up, the seg_32bit border cannot be resized above the unavailable memory region.

Specifying Pre- and Post-Link Memory Map View

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

- To enable pre-link view from the **Memory Map** pane, right-click and choose **View and Mapping Strategy (Pre-Link)**. [Figure 3-22 on page 3-31](#) illustrates memory map before linking.
- To enable post-link view from the **Memory Map** pane, right-click and choose **View and Link Results (Post-Link)**. [Figure 3-23 on page 3-32](#) 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.

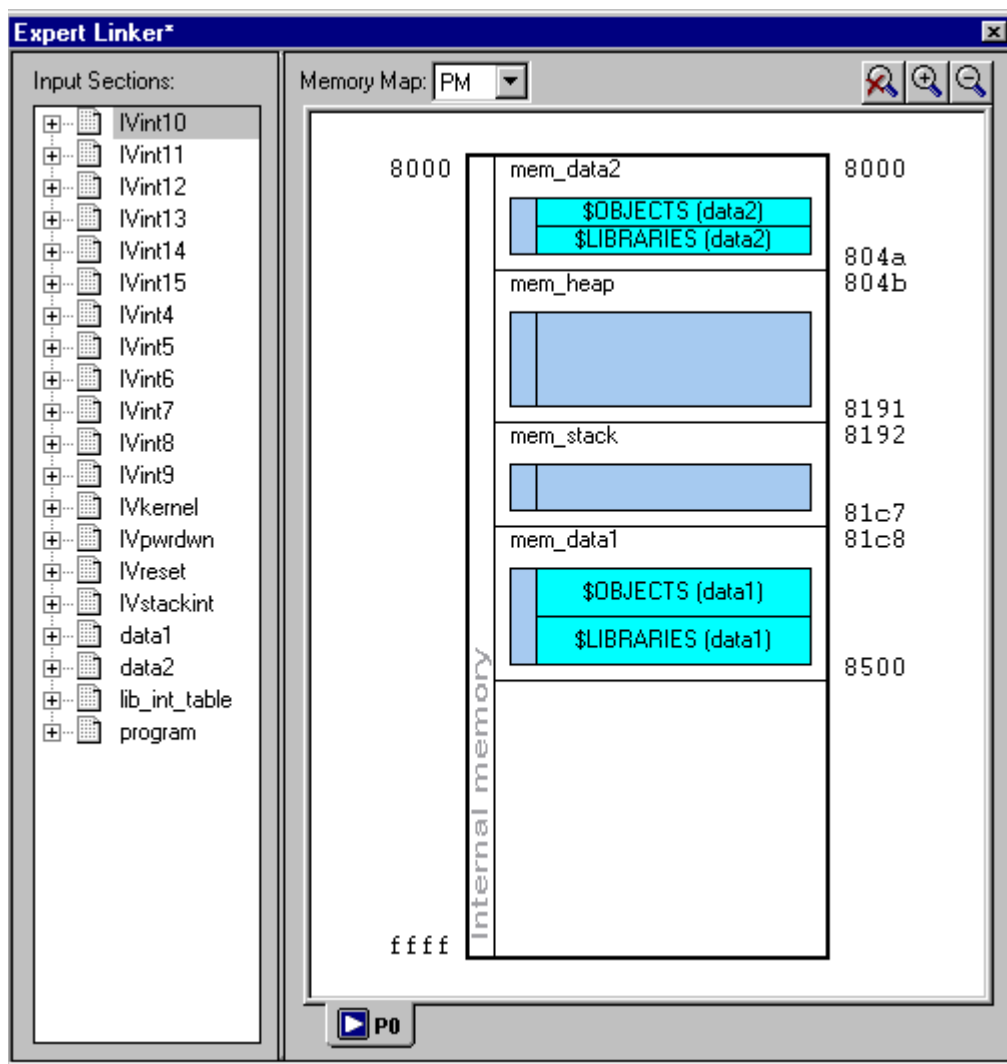


Figure 3-22. Memory Map Pane in Pre-Link View

Using the Memory Map Pane

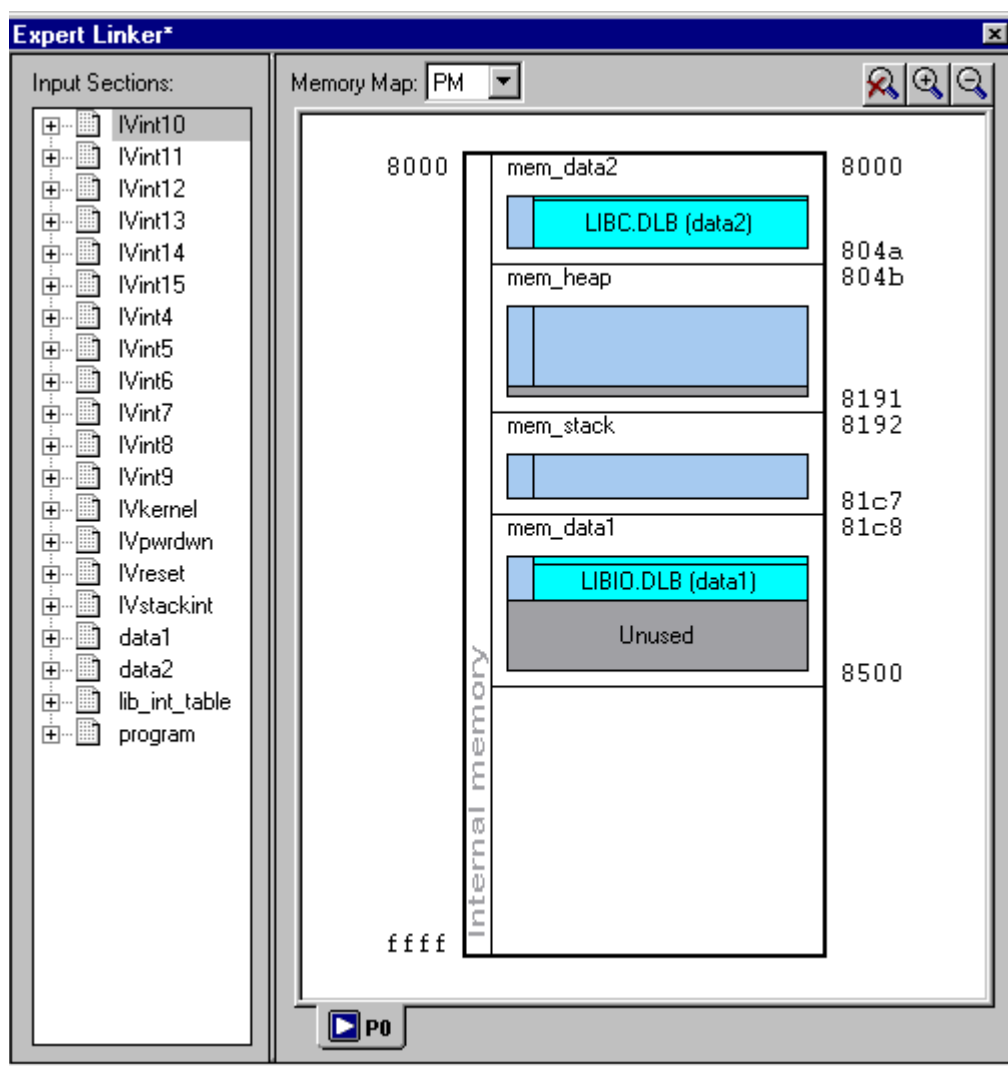


Figure 3-23. Memory Map Pane in Post-Link View

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



Figure 3-24. Memory Map – Zoom Options

Inserting a Gap into a Memory Segment

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

To insert a gap:

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

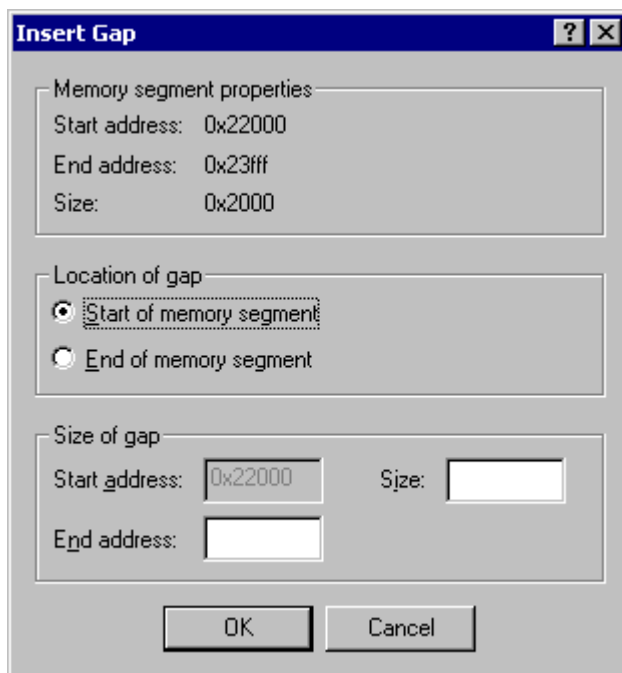


Figure 3-25. Insert Gap Dialog Box

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

Working With Overlays

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

[Figure 3-26](#) shows an overlay in live space, and [Figure 3-27](#) shows an overlay in run space.

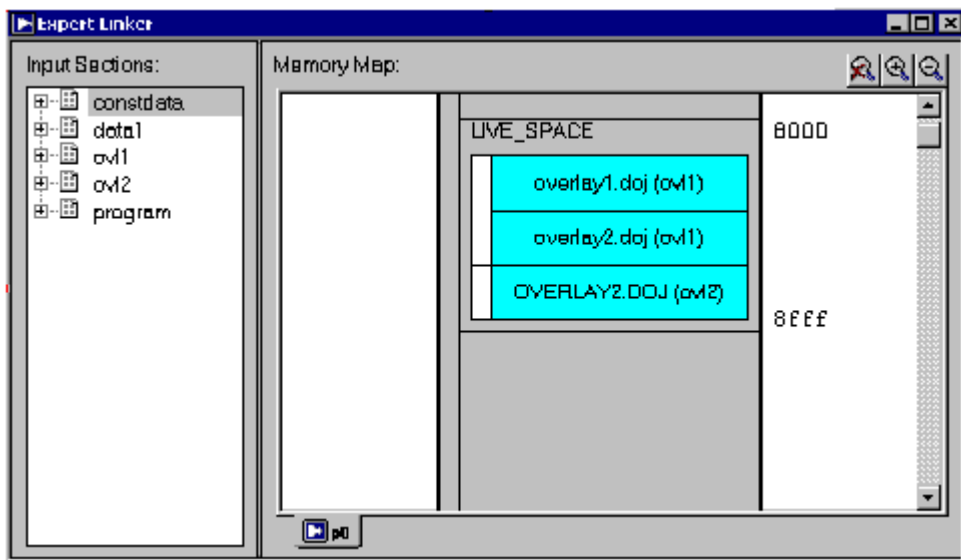


Figure 3-26. Graphical Memory Map Showing Overlay Live Space

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.

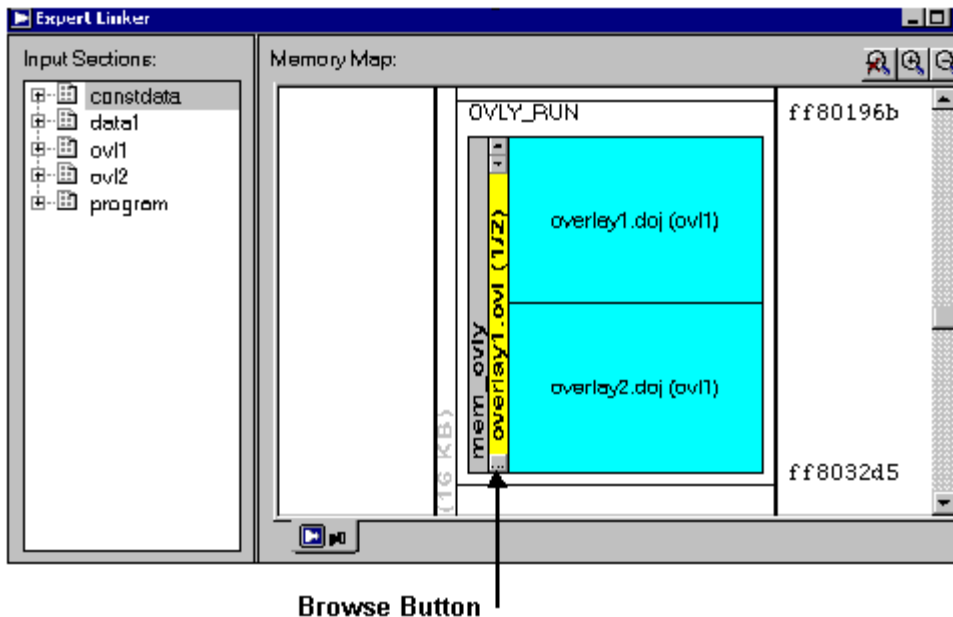


Figure 3-27. Graphical Memory Map Showing Overlay Run Space

To create an overlay in the “run” space:

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

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

Viewing Section Contents

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

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

To display the contents of an output section:

1. In the **Memory Map** pane, right-click an output section.
2. Choose **View Section Contents** from the menu.
The **Section Contents** dialog box appears.

By default, the memory section content displays in **Hex** format.
3. Right-click anywhere in the section view to display a menu with these selections:
 - **Go To** – Displays an address in the window.
 - **Select Format** — Provides a list of formats: **Hex**, **Hex and ASCII**, and **Hex and Assembly**. Select a format type to specify the memory format.

Figure 3-28, Figure 3-29, and Figure 3-30 illustrate memory data formats available for the selected output section.

Using the Memory Map Pane

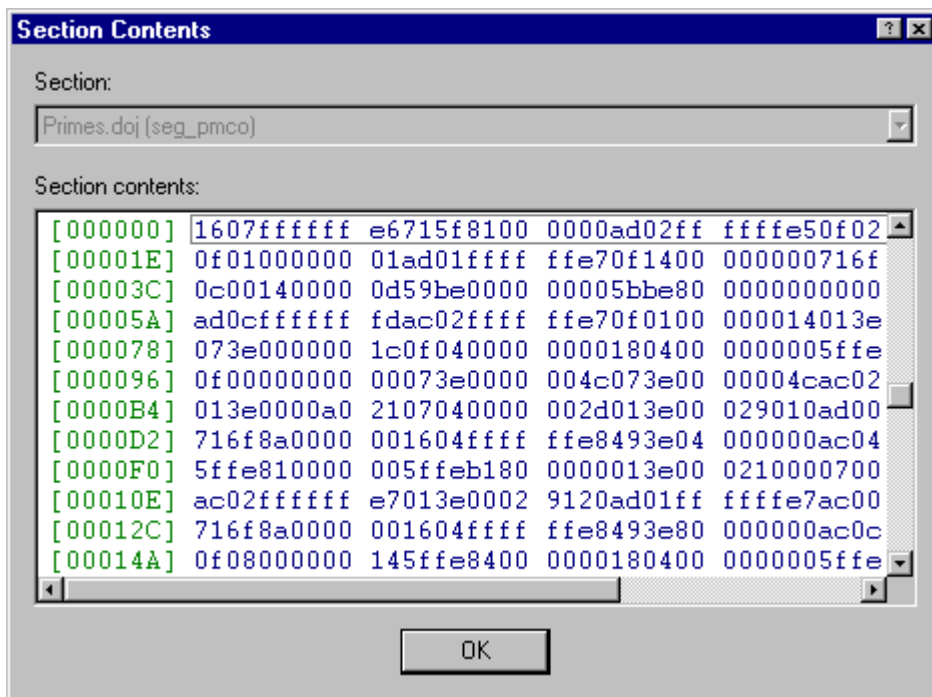


Figure 3-28. Output Section Contents in Hex Format

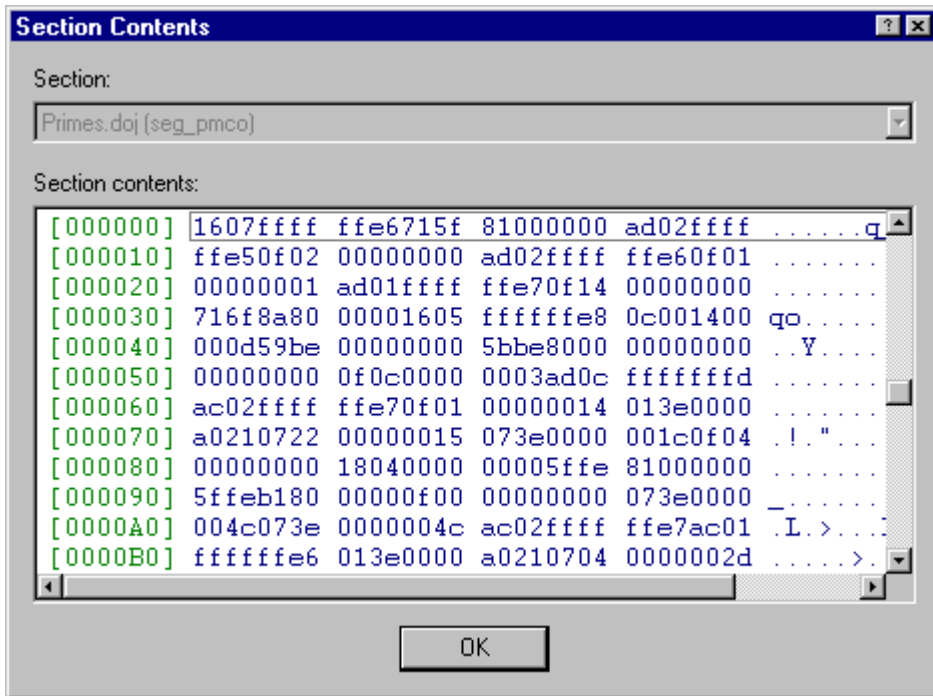


Figure 3-29. Output Section Contents in Hex and ASCII Format

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

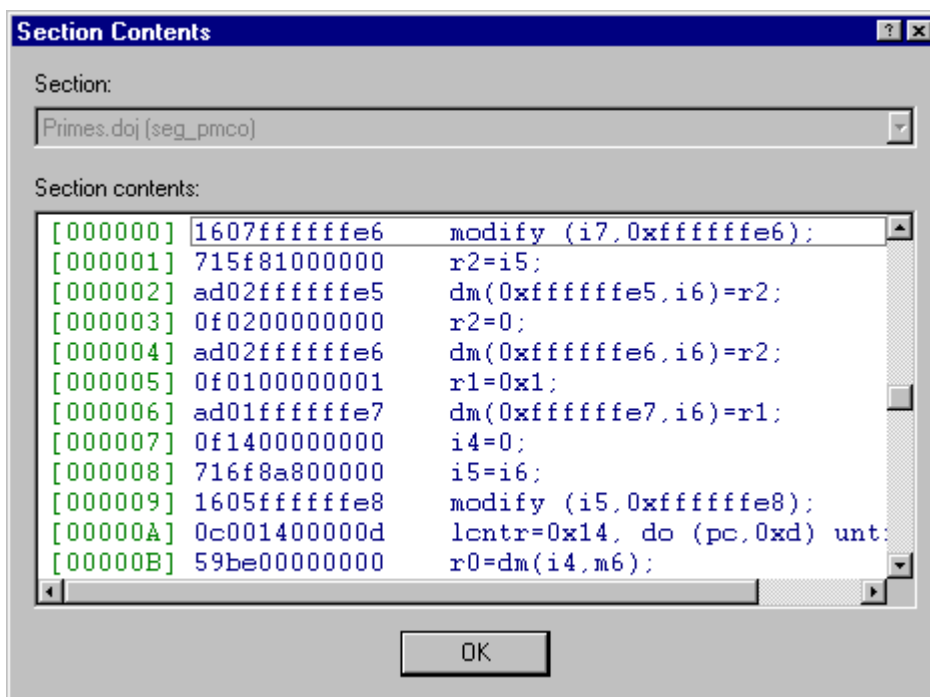


Figure 3-30. Output Section Contents in Hex and Assembly Format

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.

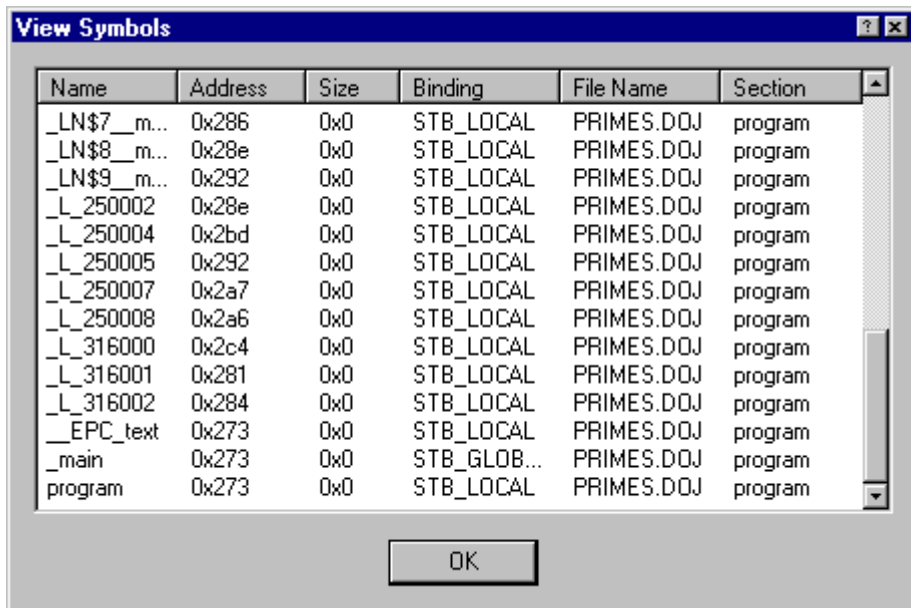


Figure 3-31. View Symbols Dialog Box

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 3-43](#)
- [“Managing Processor Properties” on page 3-44](#)
- [“Managing PLIT Properties for Overlays” on page 3-46](#)
- [“Managing Elimination Properties” on page 3-47](#)
- [“Managing Symbols Properties” on page 3-49](#)
- [“Managing Memory Segment Properties” on page 3-53](#)
- [“Managing Output Section Properties” on page 3-55](#)
- [“Managing Packing Properties” on page 3-57](#)
- [“Managing Alignment and Fill Properties” on page 3-58](#)
- [“Managing Overlay Properties” on page 3-60](#)
- [“Managing Stack and Heap in DSP Memory” on page 3-62](#)

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.

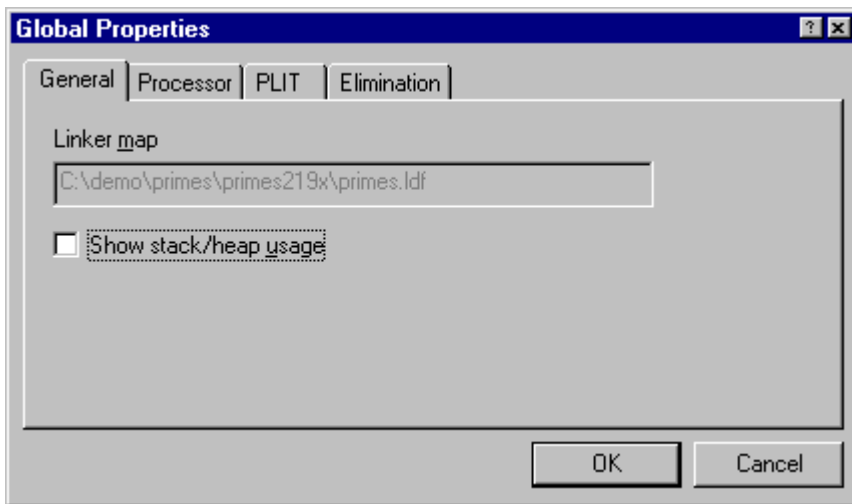


Figure 3-32. General Page of the Global Properties Dialog Box

The **Global Properties** dialog box provides these selections:

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

Managing Processor Properties

To specify processor properties:

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

The **Processor Properties** dialog box appears.

2. Click the **Properties** tab.

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

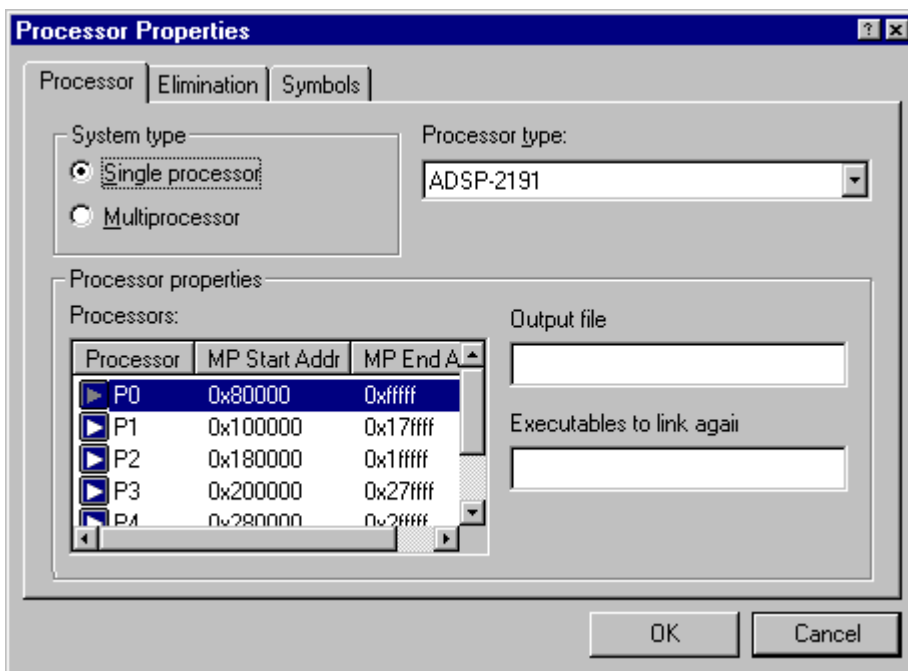


Figure 3-33. Processor Page of the Processor Properties Dialog Box

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

- Specify **System Type** – Use the **Single processor** selection.
- Select a **Processor type** (such as ADSP-21062).
- Specify an **Output file** name – The file name may include a relative path and/or LDF macro.
- Specify **Executables to link against** – Multiple files names are permitted, but must be separated with space characters. Only .SM, .DLB, and .DXE files are permitted. A file name may include a relative path and/or LDF macro.

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

Managing PLIT Properties for Overlays

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

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

To view and edit **PLIT** information:

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

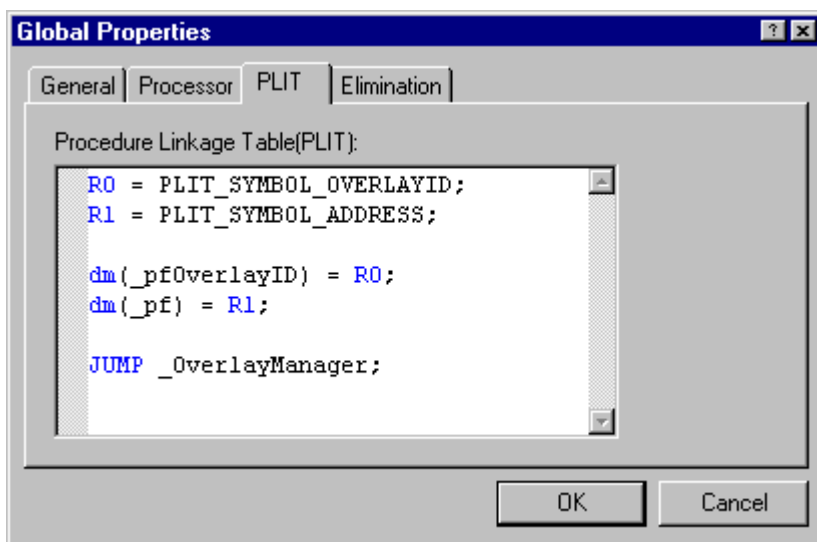


Figure 3-34. PLIT Page of the Global Properties Dialog Box

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.

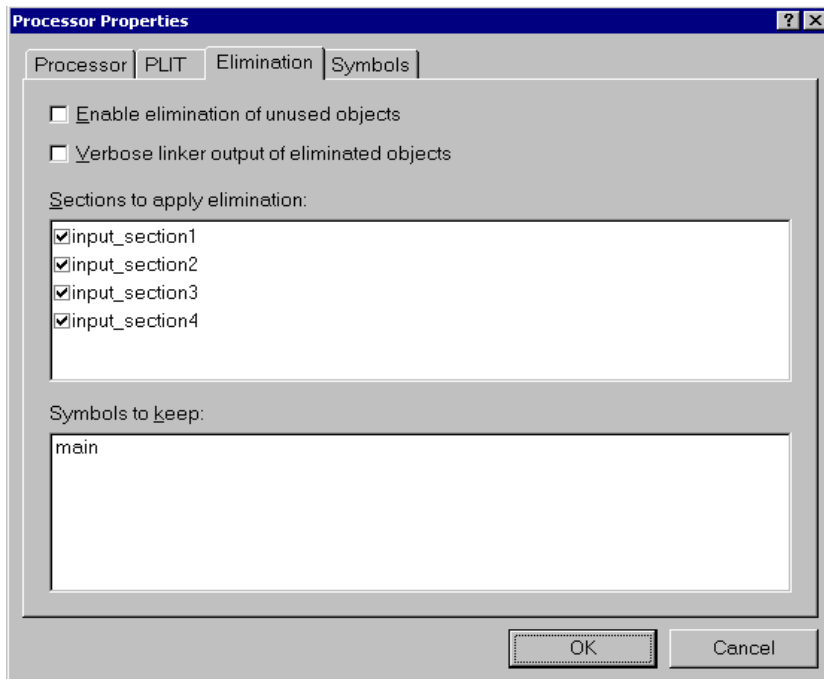


Figure 3-35. Processor Properties Dialog Box – Elimination Tab

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

Managing Object Properties

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

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

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

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

Managing Symbols Properties

You can view the list of symbols resolved by the linker. You can also add and remove symbols from the list of symbols kept by the linker. The symbols can be resolved to an absolute address or to a program file (.DxE). It is assumed that you have enabled the elimination of unused code.

To add or remove a symbol:

1. Right-click in the **Input Sections** pane of the Expert Linker window.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. Click the **Elimination** tab to add or remove a symbol.
4. Right-click in the **Symbols to keep** window.

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

To specify symbol resolution:

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

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

Managing Object Properties

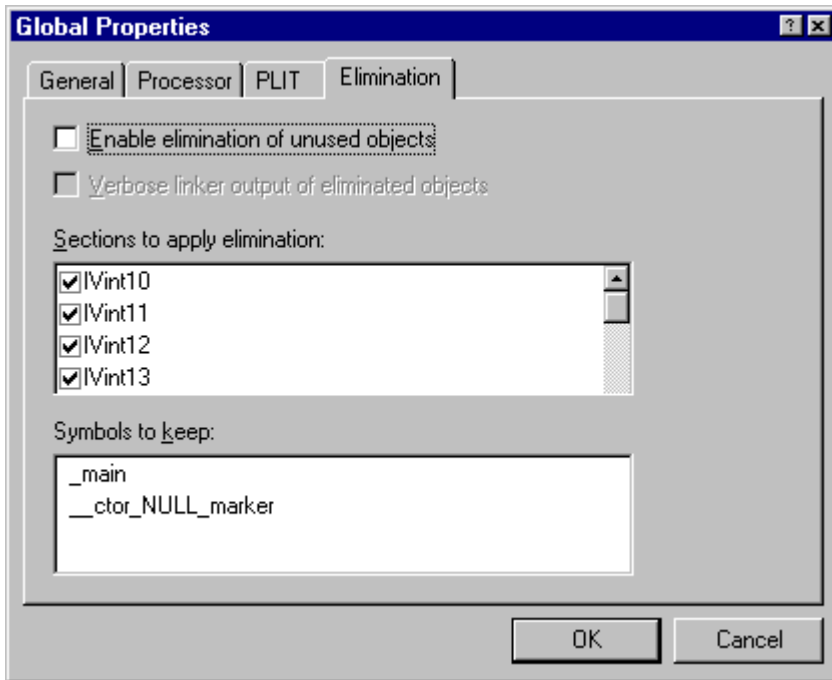


Figure 3-36. Elimination Page of the Global Properties Dialog Box

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

The **Browse** button is grayed out when no symbol list is available; for example, if the project has not been linked. When this button is active, click it to display the **Browse Symbols** dialog box, which shows a list of all the symbols.

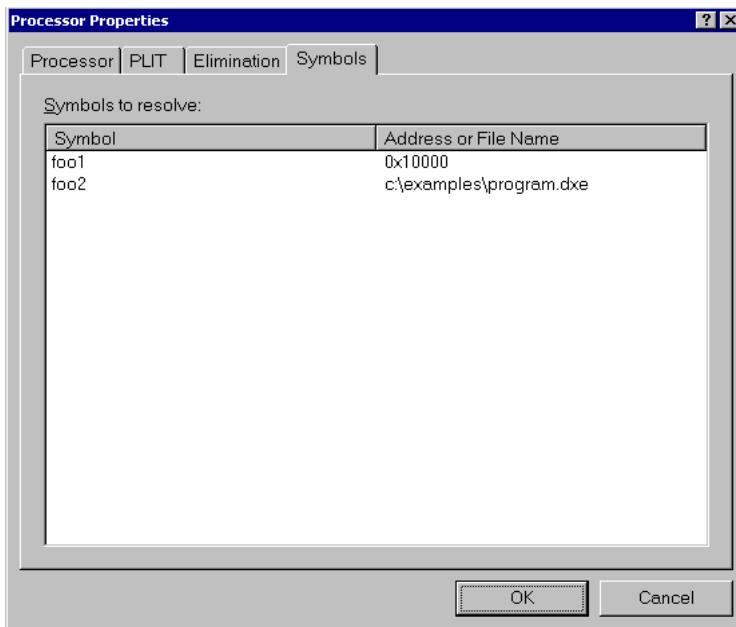


Figure 3-37. Processor Properties Dialog Box – Symbols Tab

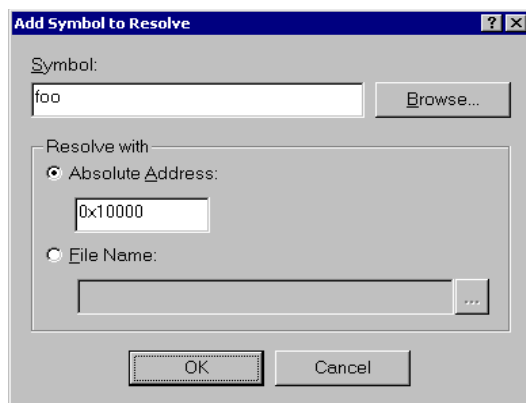


Figure 3-38. Add Symbol to Resolve Dialog Box

Managing Object Properties

Selecting a symbol from that list places it in the **Symbol** box of the **Edit Symbol to Resolve** dialog box.

To delete a symbol from the resolve list:

1. Click **Browse** to display the **Symbols to resolve** list in the **Symbols** pane ([Figure 3-37](#)).
2. Select a symbol you want to delete.
3. Right-click and choose **Remove Symbol**.

Managing Memory Segment Properties

You can specify/change the memory segment's name, start address, end address, size, width, memory space, memory type, and internal/external flag.



The BM memory space option applies only to ADSP-218x DSPs. The DATA64 memory space option applies only to ADSP-21160 and ADSP-21161N DSPs.

To display the Memory Segment Properties dialog box:

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

Managing Object Properties

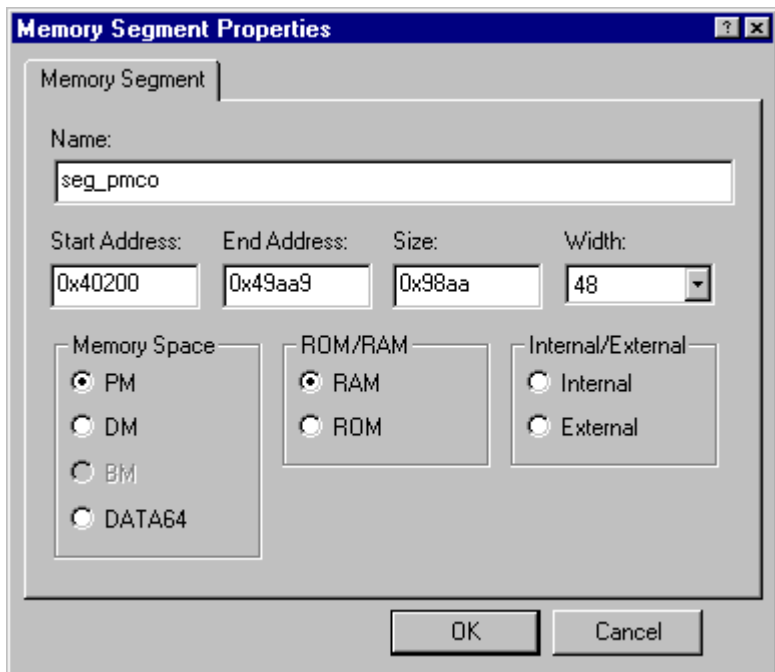


Figure 3-39. Memory Segment Properties Dialog Box

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 by right-clicking the object and then choosing **Pin to output section**.

You can:

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

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

Specify the **Packing** and **Alignment** (with **Fill Value**) properties as needed.

Managing Packing Properties

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

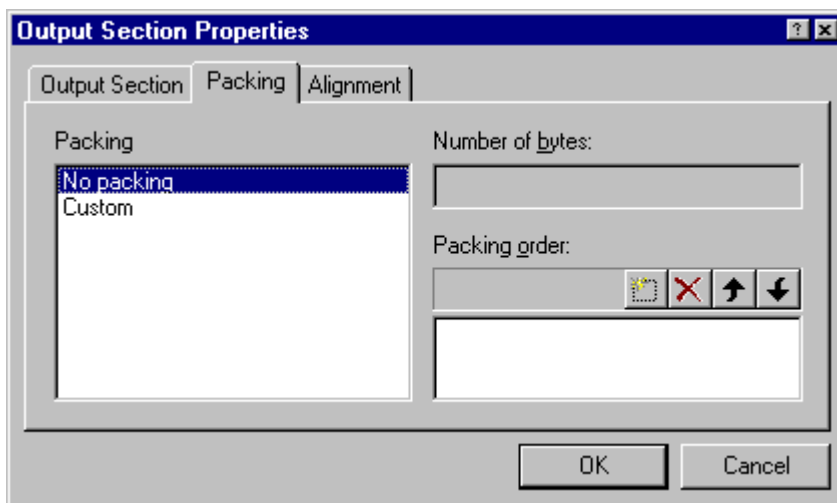


Figure 3-41. Memory Segment Properties Dialog Box – Packing Tab

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 zeros (0), NOP instructions, or a specified value.

To specify alignment properties:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties**.
3. Click the **Alignment** tab.

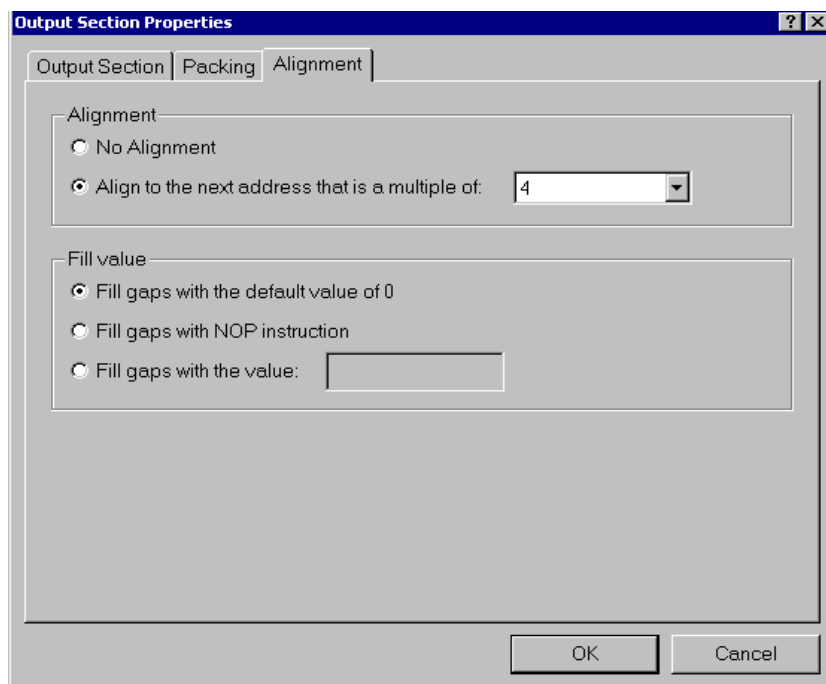


Figure 3-42. Output Section Properties Dialog Box – Alignment Tab

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**.

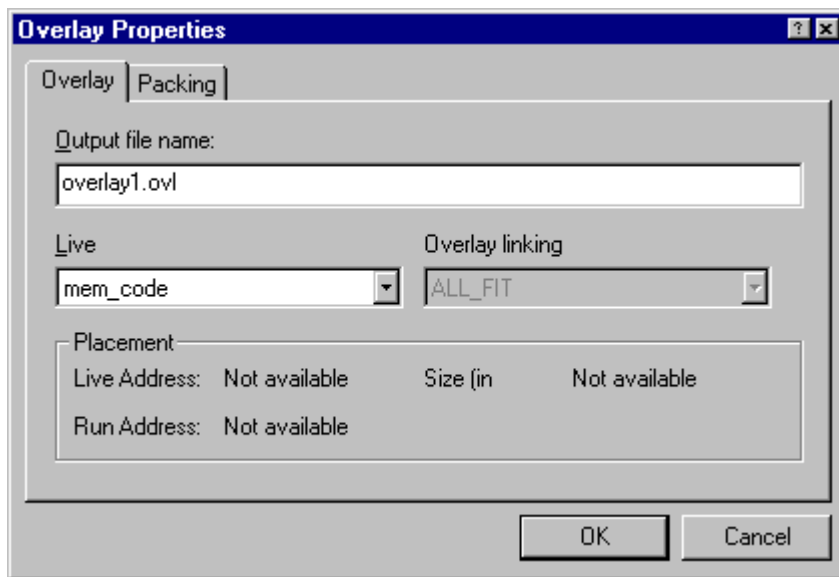


Figure 3-43. Overlay Properties Dialog Box – Overlay Tab

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.

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

Managing Stack and Heap in DSP Memory

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

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

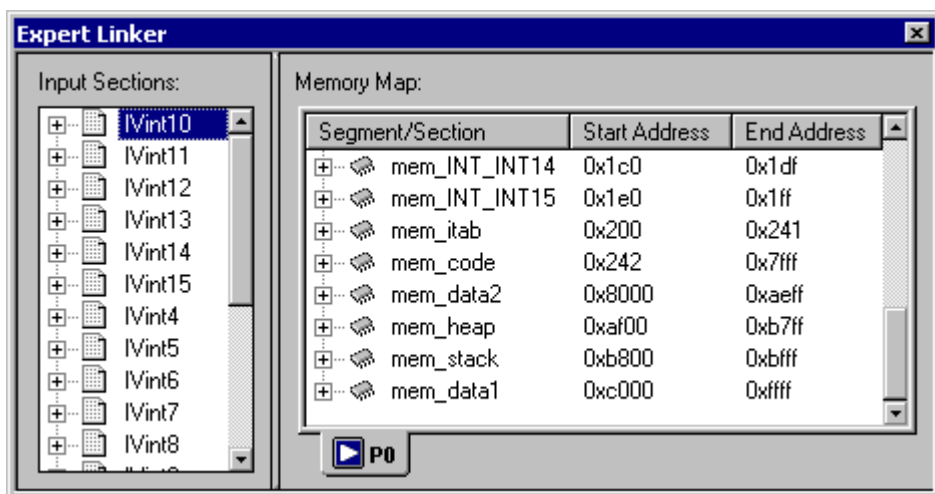


Figure 3-44. Memory Map Window With Stack/Heap Sections

Use the **Global Properties** dialog box to select **Show stack/heap usage**. This option graphically displays the stack/heap usage in memory (Figure 3-45).

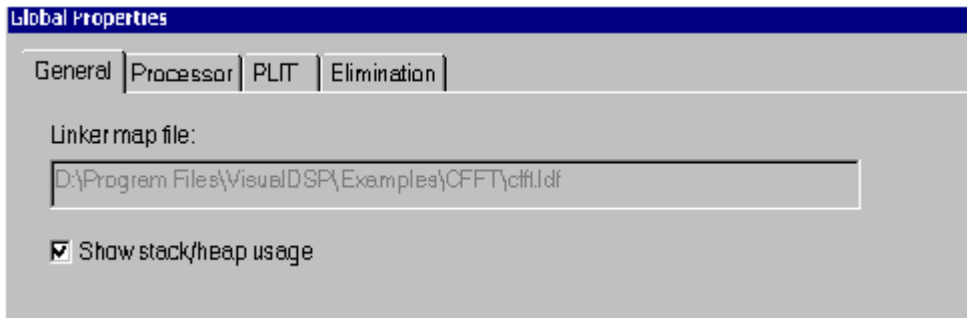


Figure 3-45. Global Properties – Selecting Stack/Heap Usage

The Expert Linker can:

- Locate stacks/heaps and fill them with a marker value.
This occurs after you load the program into a DSP target. The stacks and heaps are located by their output section names, which may vary across processor families.
- Search the heap and stack for the highest memory locations written to by the DSP program.

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

Use this information to adjust the size of your stack and heap. This information helps make better use of the DSP memory, so the stack and heap segments do not use too much memory.

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

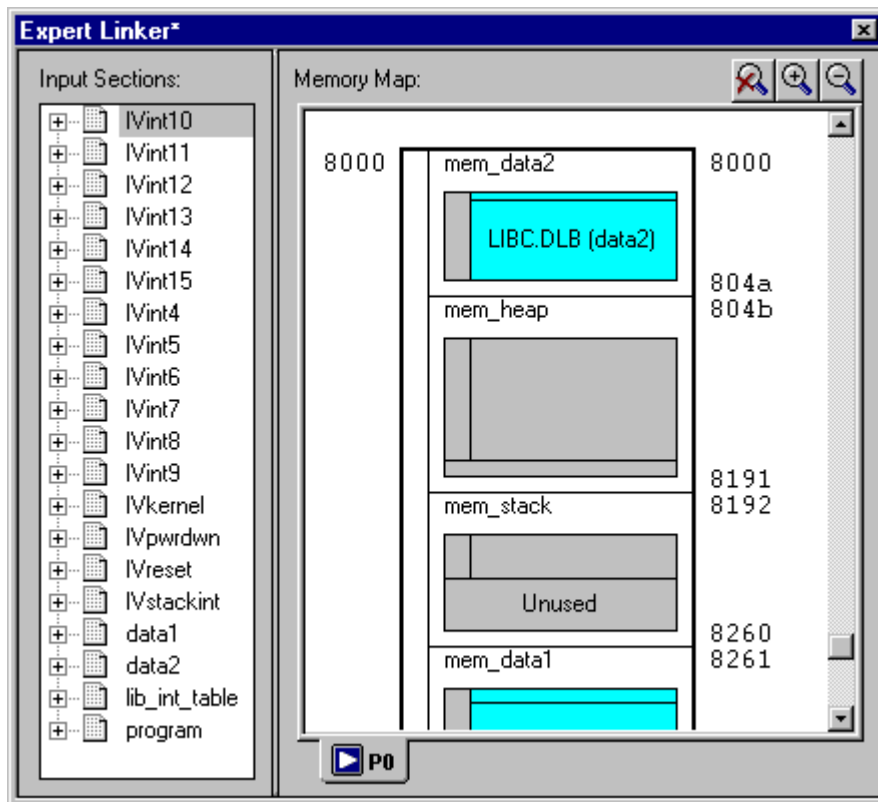


Figure 3-46. Graphical Memory Map Showing Stack/Heap Usage

4 ARCHIVER

The VisualDSP++ archiver, `elfar.exe`, combines object files (`.OBJ`) into library files, which serve as reusable resources for code development. The VisualDSP++ linker rapidly searches library files for routines (library members) referred to by other object files and links these routines into your executable program.

This chapter provides the following archiver information:

- “[Archiver Guide](#)” on page 4-2 introduces the archiver’s functions
- “[Archiver Command-Line Reference](#)” on page 4-7 reference information on archiver operations

The archiver can be run from a command line, or you can produce an archive file as the output of a VisualDSP++ project.

Archiver Guide

The `elfar.exe` utility combines and indexes object files (or any other files), producing a searchable library file. It performs the following operations, as directed by options on the `elfar` command line:

- Creates a library file from a list of object files
- Appends one or more object files to an existing library file
- Deletes file(s) from a library file
- Extracts file(s) from a library file
- Prints the contents of a specified object file of an existing library file to `stdout`
- Replaces file(s) in an existing library file
- Encrypts symbol(s) in an existing library file

The archiver can run only one of these operations at a time. However, for commands that take a list of file names as arguments, the archiver can input a text file that contains the names of object files (separated by white space). The operation makes long lists easily manageable.

The archiver, which is sometimes called a librarian, is general-purpose. It can combine and extract arbitrary files. This manual refers to DSP object files (`.DOJ`) because they are relevant to DSP code development.

Creating a Library From VisualDSP++

Within the VisualDSP++ development environment, you can choose to create a library file as your project's output. To do so, specify **DSP library file** as the target type on the **Project** page of the **Project Options** dialog box.

VisualDSP++ writes its output to `<projectname>.DLB`. To modify or list the contents of a library file, or perform any other operations on it, run the archiver from the `elfar` command line.

File Name Conventions

To maintain code consistency, use the conventions in [Table 4-1](#).



When creating a library, VisualDSP++ writes `<projectname>.DLB`.

Table 4-1. File Name Extensions

Extension	File Description
.DLB	Library file
.DOJ	Object file. Input to archiver.
.TXT	Text file used as input with the <code>-i</code> switch

Making Archived Functions Usable

In order to use the archiver effectively, you must know how to write archive files which make your DSP functions available to your code (via the linker), and how to write code that accesses these archives.

Archive usage consists of two tasks:

- Writing *archive routines*, functions that can be called from other programs
- Accessing archive routines from your code

Writing Archive Routines: Creating Entry Points

An archive routine is a routine (or function) in code that can be accessed by other programs. Each routine must have a globally visible start label (*entry point*). Code that accesses that routine must declare the entry point's name as an external variable in the calling code.

To create entry points

1. Declare the start label of each routine as a global symbol with the assembler's `.GLOBAL` directive. This defines the entry point.

The following code fragment has two entry points, `dIriir` and `FAE`.

```
...
.global dIriir;
.var/dm/seg=seg_dmda FAE[2];
.init FAE[2]: 0x1234, 0x4321;
...
.global FAE;
dIriir: CNTR=N-2;
    I2 = ^FAE;
...
```

2. Assemble and archive the code containing the routines. Use either of the following methods.
 - Direct VisualDSP++ to produce a library (see [“Creating a Library From VisualDSP++”](#)). When building the project, the object code containing the entry points is packaged in `<projectname>.DLB`. You can extract an object file (`.DOJ`), for example, to incorporate it in another project.
 - When creating executable or unlinked object files from VisualDSP++, archive them afterwards from the `elfar` command line.

Accessing Archived Functions From Your Code

Programs that call a library routine must use the assembler's `.EXTERN` directive to specify the routine's start label as an external label. When linking the program, specify one or more library files (`.DLB`) to the linker, along with the names of the object files (`.DOJ`) to link. The linker then searches the library files to resolve symbols and links the appropriate routines into the executable file.

Any file containing a label referenced by your program is linked into the executable output file. Linking libraries is faster than using individual object files, and there is no need to enter all the file names, just the library name.

Archiver Guide

In the following example, the archiver creates the `filter.dlb` library, containing the object files: `taps.doj`, `coeffs.doj`, and `go_input.doj`.

```
elfar -c filter.dlb taps.doj coeffs.doj go_input.doj
```

If you then run the linker with the following command line, the linker links the object files `main.doj` and `sum.doj`, uses the default Linker Description File (for example, `ADSP-21062.ldf`), and creates the executable file (`main.dxe`).

```
linker -DADSP-21062 main.doj sum.doj filter.dlb -o main.dxe
```

Assuming that one or more library routines from `filter.dlb` are called from one or more of the object files, the linker searches the library, extracts the required routines, and links the routines into the executable.

Archiver File Searches

File searches are important in the archiver's process. The archiver supports relative and absolute directory names, default directories, and user-specified directories for file search paths. File searches include:

- *Specified path* – If you include relative or absolute path information in a file name, the archiver searches only in that location for the file.
- *Default directory* – If you do not include path information in the file name, the archiver searches for the file in the current working directory.

Archiver Command-Line Reference

The archiver processes object files into a library file with a `.DLB` extension, which is the default extension for library files. The archiver can also append, delete, extract, or replace member files in a library, as well as list them to `stdout`. This section provides reference information on the archiver command line and linking.

elfar Command Syntax

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

```
elfar [-a|c|d|e|p|r] [-v] [-M] [-MM] [-i filename]
      lib_file obj_file [obj_file ...]
```

[Table 4-2](#) describes each switch.

Symbol Encryption

When using symbol encryption, use the following syntax.

```
elfar -s [-v] out-archive in-archive exclude-file type-letter
```

Refer to [“Archiver Symbol Name Encryption Reference” on page 4-11](#) for details.

Archiver Command-Line Reference

Example elfar Command

```
elfar -v -c my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

This command line runs the archiver as follows:

-v – Outputs status information

-c my_lib.dlb – Creates a library file named my_lib.dlb

fft.doj sin.doj cos.doj tan.doj – Places these object files in the library file

[Table 4-1 on page 4-3](#) lists typical file types, file names, and extensions.

Parameters and Switches

[Table 4-2](#) describes each archiver part of the command. Switches must appear before the name of the archive file.

Table 4-2. elfar.exe Command-Line Items

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

Table 4-2. elfar.exe Command-Line Items (Cont'd)

Item	Description
-MM	(available only with the -c switch) Prints dependencies and creates the library.
-p	Prints a list of the <i>obj_file(s)</i> (.DOJ) in the selected <i>lib_file</i> (.DLB) to standard output.
-r	Replaces the specified object file in the specified library file. The object file in the library and the replacement object file must have identical names.
-s	Specifies symbol name encryption. Refer to “Archiver Symbol Name Encryption Reference” on page 4-11 .
-v	Verbose. Outputs status information as the archiver processes files.

Constraints

The `elfar` command is subject to the following constraints:

- Select one action switch (a, c, d, e, p, r, s, M, or MM) only in a single command.
- Do not place the verbose operation switch, -v, in a position where it can be mistaken for an object file. It may not follow the *lib_file* on an append or create operation.
- The file include switch, -i, must immediately precede the name of the file to be included.

The archiver's -i switch lets you input a list of members from a text file, instead of listing each member on the command line.

- Use the library file name first, following the switches. -i and -v are not operational switches, and can appear later.
- When using the archiver's -p switch, there is no need to identify members on the command line.

Archiver Command-Line Reference

- Enclose file names containing white space or colons within straight quotes.
- Append the appropriate file extension to each file. The archiver assumes nothing, and will not do it for you.
- Wildcards are not permitted. To perform an archive operation on a list of member files, write the list in a text file and use the text file as input to the command line with the `-i` switch.
- *obj_file* — The name of an object file (`.D0J`) to be added, removed, or replaced in the *lib_file*.
- The archiver's command line is *not* case sensitive.

Archiver Symbol Name Encryption Reference

Symbol name encryption protects intellectual property contained in an archive library (.DLB) that might be revealed by the use of meaningful symbol names. Code and test a library with meaningful symbol names, and then use archive library encryption on the fully tested library to disguise the names.



Source file names in the symbol tables of object files in the archive are not encrypted. The encryption algorithm is not reversible. Also, encryption does not guarantee a given symbol will be encrypted the same way when different libraries, or different builds of the same library, are encrypted.

Command Syntax

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

```
elfar -s [-v] out-archive in-archive exclude-file type-letter
```

where:

-s – Selects the encryption operation.

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

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

in-archive – Specifies the name of the archive file (.DLB) to be encrypted. This file is not altered by the encryption process, unless in-archive is the same as out-archive.

Archiver Command-Line Reference

`exclude-file` – Specifies the name of a text file containing a list of symbols not to be encrypted. The symbols are listed one or more to a line, separated by white space.

`type-letter` – The initial letter of `type-letter` provides the initial letter of all encrypted symbols.

Constraints

All local symbols can be encrypted, unless they are correlated (see below) with a symbol having external binding that should not be encrypted. Symbols with external binding can be encrypted when they are used only within the library in which they are defined. Symbols with external binding that are not defined in the library (or are defined in the library and referred to outside of the library) should not be encrypted. Symbols that should not be encrypted must be placed in a text file, and the name of that file given as the `exclude-file` command line argument.

Some symbol names have a prefix or suffix that has special meaning. The debugger does not show a symbol starting with “.” (period), and a symbol starting with “.” and ending with “.end” is correlated with another symbol. For example, “.bar” will not be shown by the debugger, and “._foo.end” is correlated with the symbol “_foo” appearing in the same object file. The encryption process encrypts only the part of the symbol after any initial “.”, and before any final “.end”. This part is called the root of the symbol name. Since only the root is encrypted, a name with a prefix or suffix having special meaning retains that special meaning after encryption.

The encryption process ensures that a symbol with external binding will be encrypted the same way in all object files contained in the library. This process also ensures that correlated symbols (see explanation above) within an object file will be encrypted the same way, so they remain correlated.

The names listed in the `exclude-file` are interpreted as root names. Thus, “_foo” in the `exclude-file` prevents the encryption of the symbol names “_foo”, “._foo”, “_foo.end”, and “._foo.end”.

The `type-letter` argument, which provides the first letter of the encrypted part of a symbol name, ensures that the encrypted names in different archive libraries can be made distinct. If different libraries are encrypted with the same `type-letter` argument, there is a possibility that unrelated external symbols of the same length will be encrypted identically.

5 ADSP-2106X/ADSP-21160 LOADER

Use the loader utility (`elfloader.exe`) to convert executable files into bootloadable files for SHARC DSPs. A bootloadable file is transported to (and run from) DSP internal memory.

 This chapter describes loader and booting processes for ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 DSPs. Refer to [“ADSP-21161N Loader” on page 6-1](#) for information about ADSP-21161N DSPs.

The loader processing executable (`.DxE`) files to generate a bootloadable file with an `.LDR` extension. To make a loadable file, the loader processes data from a boot-kernel file (`.DxE`) and one or more other executable files (`.DxE`). After fully debugged a program, use the loader to generate a set of bootloadable files for your target system.

This chapter contains the following information.

- [“Loader Guide”](#)
- [“Running the Loader”](#)

Refer to EE-notes on the Analog Devices DSP Web site for related information.

Loader Guide

The loader (`elfloader.exe`) processes executable files, producing a boot-loadable file. This file is used to automatically download programs to the processor's internal memory after power-up or after a software reset. This process is called **booting**.

ADSP-2106x/21160 DSPs support three booting modes: EPROM, host, and link. The DSPs have a special hardware feature that boot-loads a small, 256-instruction program called a boot-kernel (or boot-loader) into the DSP's internal memory at chip reset. When executed, the boot-kernel facilitates booting the user's application code.

This section describes the following topics:

- [“Booting”](#)
- [“Boot Types” on page 5-15](#)
- [“Multiprocessor Booting” on page 5-24](#)

Booting

Before selecting options for the loader's operation, understand how the loader's features apply to the boot-type that you are using. This section describes boot-types for SHARC DSPs and relates loader features that support the appropriate boot-types.

ADSP-2106x/21160 DSPs support three boot types (modes): EPROM, host, and link, as shown in [Table 5-3 on page 5-6](#) and [Table 5-4 on page 5-6](#). Boot-loadable files for these modes pack boot data into 48-bit instructions and use an appropriate DMA channel of the DSP's DMA controller to boot-load the instructions.



ADSP-2106x DSPs use DMA channel 6 (DMAC6) when booting. In ADSP-21160 DSPs, DMAC8 is used for link port booting and DMAC10 is used for the host and EPROM booting.

- When booting from an EPROM through the external port, the SHARC DSP reads boot data from an 8-bit external EPROM.
- When booting from a host processor through the external port, the SHARC DSP accepts boot data from a 16- or 32-bit host microprocessor.
- When booting through the link port, SHARC DSPs receive boot data through the link port as 4-bit-wide data in link buffer 4.
- For the no boot mode, SHARC DSPs begin executing instructions from external memory.

After the boot process has loaded 256 instructions into the appropriate memory location (starting at 0x20000 for ADSP-21060/61/62 DSPs, 0x8000 for ADSP-21065L DSPs, and 0x40000 for ADSP-21160 DSPs), the processor begins to execute instructions. Because most applications require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. This loading routine (Loader Kernel or Boot Kernel) is used to load an entire program.

Software developers who use the loader should be familiar with the following operations:

- [“General Power-Up Booting Process” on page 5-4](#)
- [“Boot Mode Selection” on page 5-5](#)

Loader Guide

- [“Boot-Loading Process”](#) on page 5-8
- [“Boot-Kernel Changes and Software Issues”](#) on page 5-10
- [“Boot-Kernels”](#) on page 5-13
- [“Interrupt Vector Table Location”](#) on page 5-14

General Power-Up Booting Process

At chip reset, SHARC DSPs boot-load a small, 256-instruction program (boot-kernel) into the DSP’s internal memory. When executed, the boot-kernel loads the user application code. The combination of the boot-kernel and the application code comprise a boot-loadable file.

Each DSP model has a start address and reset vector, as shown in [Table 5-1](#). Different DMA channels are used by the various DSP models, as shown in [Table 5-2](#).

Table 5-1. DSP Addresses

DSP Model	Start Address	Reset Vector
ADSP-21060	0x20000	0x20004
ADSP-21061	0x20000	0x20004
ADSP-21062	0x20000	0x20004
ADSP-21065L	0x8000	0x8004
ADSP-21160	0x40000	0x40004

Table 5-2. DSP DMA Channels

DSP Model	PROM Booting	Host Booting	Link Booting
ADSP-21060	6	6	6
ADSP-21061	6	6	not supported
ADSP-21062	6	6	6

Table 5-2. DSP DMA Channels (Cont'd)

DSP Model	PROM Booting	Host Booting	Link Booting
ADSP-21065L	8 (DMAC0 programs DMA channel 8)	8 (DMAC0 programs DMA channel 8)	not supported
ADSP-21160	10	10	8

At power-up, the general booting process includes the following steps.

1. A DMA channel is automatically configured to transfer 256 48-bit instructions starting at the address shown in [Table 5-1](#).
These instructions are called a “boot-kernel” or “loader-kernel”.
2. The boot-kernel then runs and starts up 48-bit word DMAs to load the application executable code and data.
3. The boot-kernel overwrites itself with the application’s first 256 words and then begins to execute the user application code from locations 0x20000 to 0x200FF (ADSP-21060/61/62), 0x8000 to 0x80FF (ADSP-21065L), and 0x40000 to 0x400FF (ADSP-21160).



The reset vector address (refer to [Table 5-1](#)) must not contain a valid instruction since it is not executed during the booting sequence. Place a NOP or IDLE instruction at this location.

The boot-kernel can come from an external PROM, a host processor, or another SHARC DSP. Choosing the boot type directs the system to prepare the appropriate boot-kernel.

Boot Mode Selection

The state of various pins select the DSP’s boot mode (boot type). For the ADSP-21060, ADSP-21061, ADSP-21062, and ADSP-21160 DSPs, refer to [Table 5-3](#) and [Table 5-4](#). For ADSP-21065L DSPs, refer to [Table 5-3](#) and [Table 5-4](#).

Table 5-3. Boot Mode Pins - ADSP-21060, 061, 062, and ADSP-21160

Pin	Type	Description
EBOOT	I	EPROM Boot – When EBOOT is high, the DSP boot-loads from an 8-bit EPROM through the DSP's external port. When EBOOT is low, the LBOOT and $\overline{\text{BMS}}$ pins determine the booting mode.
LBOOT	I	Link Boot – When LBOOT is high and EBOOT is low, the DSP boots from another SHARC through the link port. When LBOOT is low and EBOOT is low, the DSP boots from a host processor through the DSP's external port.
$\overline{\text{BMS}}$	I/O/T ¹	Boot Memory Select – When boot-loading from an EPROM (EBOOT=1 and LBOOT=0), this pin is an <i>output</i> and serves as the chip select for the EPROM. In an MP system, $\overline{\text{BMS}}$ is output by the bus master. When host-booting or link-booting (EBOOT=0), $\overline{\text{BMS}}$ is an <i>input</i> and must be high.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 5-4. Boot Mode - ADSP-21060, 061, 062, and ADSP-21160

EBOOT	LBOOT	$\overline{\text{BMS}}$	Boot Mode
0	0	0 (Input)	No boot (processor executes from external memory)
0	0	1 (Input)	Host processor
0	1	0 (Input)	Reserved
0	1	1 (Input)	Link port
1	0	Output	EPROM ($\overline{\text{BMS}}$ is chip select)
1	1	x (Input)	Reserved

Table 5-5. Boot Mode Pins - ADSP-21065L

Pin	Type	Description
$\overline{\text{BMS}}$	I/O/T ¹	Boot Memory Select When BSEL is low, $\overline{\text{BMS}}$ is an input pin and selects between host boot mode and no boot mode. In no boot mode, the processor executes from external memory. For no boot mode, connect $\overline{\text{BMS}}$ to ground. For host boot mode, connect $\overline{\text{BMS}}$ to VDD. When BSEL is high, $\overline{\text{BMS}}$ is an output pin and the processor starts up in EPROM boot mode. Connect $\overline{\text{BMS}}$ to the EPROM's chip select.
BSEL	I	EPROM Boot Select Hardwire this signal; it is used for system configuration. When BSEL is high, the processor starts up in EPROM boot mode. The processor assumes the EPROM's data bus is 8 bits wide. Connect BSEL to the processor's data bus in LSB alignment. When BSEL is low, $\overline{\text{BMS}}$ determines the booting mode. Connect BSEL to ground.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 5-6. Boot Mode - ADSP-21065L

BSEL	$\overline{\text{BMS}}$	Description
0	1	No boot mode. The processor executes from external memory at location 0x20004.
0	1	Host boot mode. The processor defaults to an 8-bit host bus width.
1	Output	EPROM boot mode. The processor assumes an 8-bit EPROM data bus width. Connect to the data bus in LSB alignment.

Boot-Loading Process

Typically, the first 256 instructions brought into the DSP is the loader kernel. The development tools provides default kernels for each boot mode in the VisualDSP\...\ldr directory. Once the kernel has been loaded successfully into the DSP, the kernel follows the following sequence:

1. Each kernel begins with general initializations for the DAG registers, appropriate interrupts, processor ID information, and various SDRAM or WAIT state initializations.
2. Once the kernel has finished the task of initializing the DSP environment, the kernel's main job is to initialize DSP memory, both internal and external, with user application code.
3. The loader file enables the kernel to load code and data section by section. In the loader file, each section of code or data is preceded by a block header, which describes how long the block is, where it belongs, and what type of data or instruction it is. [Table 5-7](#) includes the block header format. After the block header, the loader outputs the block body, which includes the actual data or instructions that need to be placed.

Table 5-7. Boot Data Block Format

Block Header	First word	Bits 16-47 are not used. Bits 0-15 define the type of data block
	Second word	Bits 16-47 are not used. Bits 0-15 define the type of data block
Block Body (if not a zero block)		Word 1 (48 bits) Word 2 (48 bits) :

4. The loader identifies the data type in the block header with a tag. The 16-bit tag that precedes the block identifies each initialization block. Each type of initialization has a unique tag number (tag number - initialization type) as shown in [Table 5-8](#).

Table 5-8. ADSP-2106x/21160 DSP Loader Tags

Tag Number	Block Type	Tag Number	Block Type
0x0000	final init	0x000A	zero pm48
0x0001	zero dm16	0x000B	init pm16
0x0002	zero dm32	0x000C	init pm32
0x0003	zero dm40	0x000D	init pm40
0x0004	init dm16	0x000E	init pm48
0x0005	init dm32	0x000F	zero dm64 (21160 only)
0x0006	init dm40	0x0010	init dm64 (21160 only)
0x0007	zero pm16	0x0011	zero pm64 (21160 only)
0x0008	zero pm32	0x0012	init pm64 (21160 only)
0x0009	zero pm40		

5. The loader kernel enables the boot port (external or link) to read the block header. Then, based on that information, the kernel places the block body information in the appropriate place in memory using the core.
6. The final section, which is identified by a tag of 0x0, is called the final initialization stage. This section is self modifying code that, when executed, facilitates a DMA over the kernel, replacing it with the user application code that actually belongs in that space at runtime. The final initialization code also takes care of interrupts and returns the DSP registers such as SYSCON and DMAC or LCTL to their default values.

7. When the loader detects this tag, it reads the next 48-bit word, indicating the instruction when the loading is completing. This instruction is loaded into the 48-bit `PX` register after the boot-loader finishes initializing memory.
8. The boot kernel requires that the interrupt, External Port, or Link Port address (depending on the boot type) contains an RTI instruction. This RTI is inserted automatically by the loader utility and is needed to ensure that the kernel executes from the reset vector once the DMA that overwrites the kernel is complete. A last remnant of the kernel code is left at the reset vector location to replace the RTI with the user's intended code. Because of this last kernel remnant, user application code should not use the first location of the reset vector. This first location should be a NOP or IDLE instruction. The kernel will automatically complete, and the PC will begin sequencing the user application code at the second location in the DSPs reset vector space.
9. When the boot process is complete, the DSP automatically executes the user application code. The only remaining evidence of the boot kernel will be at the first location of the interrupt vector. Almost no memory is sacrificed to the boot code.

Boot-Kernel Changes and Software Issues

Some systems require boot-kernel customization. The operation of other tools (such as the C/C++ compiler) is influenced by whether the loader is used. This section describes loader usage issues.

When producing a boot-loadable file, the loader reads a DSP executable file and uses information in it to initialize memory correctly. However, the loader cannot determine how the DSP's `SYSCON` and `WAIT` registers are to be configured for external memory loading in the system.

If you modify the boot-kernel by inserting values for your system, you must rebuild it before generating the boot-loadable file. The boot-kernel contains default values for `SYSCON`. The initialization code may be found in the comments for the boot-kernel source file.

After modifying a copy of the applicable boot-kernel source file (such as `060_link.asm`, `060_host.asm`, or `060_prom.asm`), refer to [“SHARC Loader Command-Line Items” on page 5-30](#) for the list of kernel files. Use the loader-kernel (-l) command-line switch to appropriately initialize the `SYSCON` and `WAIT` registers.



When using VisualDSP++, specify the name of the modified kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

If you modify the boot-kernel for EPROM-, host-, or link-booting modes, ensure that the `seg_ldr` memory segment is defined in the `.LDF` file. Refer to the source of this segment in the `.LDF` file located in the `...\21k\ldr\` or `(...\211xx\ldr\)` directory.

Because the loader uses this address for the first location of the reset vector during the boot-load process, avoid placing code at this address. When using any of the DSP's power-up booting-modes, ensure that this address does not contain a critical instruction, because this address is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location. The loader generates a warning when vector address (`0x20004` for ADSP-21060/61/62 DSPs, `0x40004` for ADSP-21160 DSPs, or `0x8004` for ADSP-21065L DSPs) does not contain `NOP` or `IDLE`.

When working with the C/C++ compiler and using the loader, do not use the `mem21k` utility, which initializes DSP run-time memory for compiled C/C++ code. Instead, use the compiler's `-no-mem` switch to disable `mem21k`. For code compiled with the `-no-mem` switch, the `seg_init` segment defined in the `.LDF` file need only contain 16 words (address locations).

Loader Guide

When working with the C/C++ compiler and using the splitter instead of the loader, you must disable the loader and use the `mem21k` utility. Disable the loader from VisualDSP++ by selecting **mem21k** on the **Load** page of the **Project Options** dialog box.

The load kernel project can be rebuild from the VisualDSP++ IDDE. You may also use command lines to rebuild various default boot-kernels for ADSP-2106x DSPs.

EPROM Booting

The default boot-kernel source file for EPROM booting is `060_prom.asm`. Copy this file to `my_prom.asm` and modify it to suit your system. Then, use the following commands to rebuild the boot-kernel.

```
easm21k -21060 my_prom.asm
```

- or -

```
easm21k -proc ADSP-21060 my_prom.asm  
linker -T 060_ldr.ldf my_prom.doj
```

Host Booting

The default boot-kernel source file for host booting is `060_host.asm`. Copy this file to `my_host.asm` and modify it to suit your system. Then, use the following commands to rebuild the boot-kernel:

```
easm21k -21060 my_host.asm
```

- or -

```
easm21k -proc ADSP-21060 my_host.asm  
linker -T 060_ldr.ldf my_host.doj
```

Link Port Booting

The default boot-kernel source file for link port booting is `060_link.asm`. Copy this file to `my_link.asm` and modify it to suit your system. Then, use the following commands to rebuild the boot-kernel:

```
easm21k -21060 my_link.asm
```

- or -

```
easm21k -proc ADSP-21060 my_link.asm  
linker -T 060_ldr.ldr my_link.doj
```

Rebuilding Boot Kernels

To rebuild boot-kernels for ADSP-21065L DSPs, use these commands.

```
easm21k -21065L my_prom.asm
```

- or -

```
easm21k -proc ADSP-21065L my_prom.asm  
linker -T 065L_ldr.ldr my_prom.doj
```

To rebuild boot-kernels for ADSP-21160 DSPs, use these commands.

```
easm21k -21160 my_prom.asm
```

- or -

```
easm21k -proc ADSP-21160 my_prom.asm  
linker -T 160_ldr.ldr my_prom.doj
```

Boot-Kernels

For all boot-kernels, the boot-loading process begins by downloading the boot-kernel into the DSP's memory. The boot-kernel sets up the DSP as necessary and loads the boot data. After the boot-kernel has finished initializing the rest of the system, it performs a final DMA transfer and loads boot data over itself.

Loader Guide

Boot-kernels are loaded at reset into program segment `seg_ldr`, which is defined in `06x_ldr.ldf` (for ADSP-2106x DSPs) and in `160_ldr.ldf` (for ADSP-21160 DSPs).

This file is stored in the DSP tools installation directory in `(...\21k\ldr)` for ADSP-2106x DSPs and `(...\211xx\ldr)` for ADSP-21160 DSPs.

For example,

- `060_prom.asm` (for ADSP-2106x PROM boot-kernels)
- `060_host.asm` (for ADSP-2106x host boot-kernels)
- `060_link.asm` (for ADSP-2106x link boot-kernels)



The source file for the ADSP-21065L PROM boot-kernel is `065L_prom.asm`. For the ADSP-21160 PROM boot-kernel, the source file is `160_prom.asm`, and so on.

Interrupt Vector Table Location

If a SHARC DSP is booted from an external source (EPROM, host, or another SHARC processor), the interrupt vector table is located in internal memory. If, however, the DSP is not booted and executes from external memory, the vector table must be located in the external memory.

The `IIVT` bit of the `SYSCON` control register can be used to override the booting mode in determining where the interrupt vector table is located. If the DSP is not booted (no boot mode), setting `IIVT` to 1 selects an internal vector table, and setting `IIVT` to 0 selects an external vector table. If the DSP is booted from an external source (any mode other than no boot mode), `IIVT` has no effect. The `IIVT` default initialization value is 0.

Boot Types

The following sections describe available boot types.

- “EPROM Booting”
- “Host Booting” on page 5-19
- “Link Booting” on page 5-23
- “No Boot Mode” on page 5-24

For MP booting, refer to “Multiprocessor Booting” on page 5-24.

EPROM Booting

EPROM booting through the external port is selected when the $\overline{\text{EB00T}}$ pin is high and the $\overline{\text{LB00T}}$ pin is low. These settings cause the $\overline{\text{BMS}}$ pin to become an output, serving as chip select for the EPROM. Table 5-9 lists all PROM connections to DSPs.

Table 5-9. PROM Connections to ADSP-21xxx DSPs

ADSP-	Connection
21060/61/62	PROM/EPROM connected to DATA23-16 pins
21065L	PROM/EPROM connected to DATA0-7 pins
21160	PROM/EPROM connected to DATA32-39 pins
21xxx	Address pins of PROM connect to lowest address pins of any DSP
21xxx	Chip select connect to the $\overline{\text{BMS}}$ pin
21060/61/62/65L	Output enable connect to the $\overline{\text{RD}}$ pin
21160	Output enable connect to $\overline{\text{RDH}}$ pin

Loader Guide

During reset, the ACK line is pulled high internally with a 2-Kohm equivalent resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

The DMA channel parameter registers are initialized at reset for EPROM booting as shown in [Table 5-10](#) and [Table 5-11](#). The count is initialized to 0x0100 to transfer 256 words to internal memory. The external count register (ECx), which is used when external addresses are generated by the DMA controller, is initialized to 0x0600 (0x100 words at six bytes per word).

Table 5-10. ADSP-2106x DSPs – DMA Settings for EPROM Booting

		Processor	
		ADSP-21060/61/62	ADSP-21065L
SYSCON		0x00000010	0x00000020
DMA Channel		DMAC6	DMAC8
II6	IIEP0	0x20000	0x8000
IM6	IMEP0	0x1 (Implied)	0x1 (Implied)
C6	CEP0	0x100	0x100
EI6	EIEP0	0x400000	0x00000000
EM6	EMEP0	0x1 (Implied)	0x1 (Implied)
EC6	ECEP0	0x600	0x600
IRQ Vector		0x20040	0x8040

Table 5-11. ADSP-21160 DSPs – DMA Settings for EPROM Booting

		ADSP-21160
SYSCON		0x00
DMA Channel		DMAC10 (EPB0)

Table 5-11. ADSP-21160 DSPs – DMA Settings for EPROM Booting

	ADSP-21160
II10	0x40000
IM10	0x1 (Implied)
C10	0x100
EI10	0x800000
EM10	0x1 (Implied)
EC10	0x600
IRQ Vector	0x40050

After the DSP's RESET pin goes inactive on start-up, a SHARC system configured for EPROM booting undergoes the following boot-loading sequence:

1. The DSP's $\overline{\text{BMS}}$ pin becomes the boot EPROM chip select.
2. The DSP goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to the DSP reset vector address (refer to [Table 5-1 on page 5-4](#)).
3. The DMA controller reads 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them into internal memory (low-to-high byte packing order) until the 256 words are loaded.
4. The DMA parameter registers for appropriate DMA channels are initialized, as shown in [Table 5-10](#) and [Table 5-11](#). The external port DMA channel (6 or 10) becomes active following reset; it is initialized to set external port DMA enable and selects DTYPE for instruction words. The packing mode bits (PMODE) are ignored, and 48-to-8 bit packing is forced with least-significant word first. The UBWS and UBWM fields of the WAIT register are initialized to generate six wait states for the EPROM access in unbanked external memory space.

5. The DSP begins 8-bit DMA transfers from the EPROM to internal memory using the following external port data bus lines:

D23-D16 (ADSP-21060/61/62 DSPs)

D0-D8 (ADSP-21065L DSPs)

D32-D39 (ADSP-21160 DSPs)

6. Data transfers begin and increment after each access. The external address lines (ADDR 31-0), start at:

0x40 0000 (ADSP-21060/61/62 DSPs)

0x00 0000 (ADSP-21065L DSPs)

0x80 0000 (ADSP-21160 DSPs)

7. The DSP's $\overline{RD}$ pin asserts as in a normal memory access, with six wait states (seven cycles).
8. After finishing DMA transfers to load the boot-kernel into the DSP, the BS0 bit is cleared in the SYSCON register, deactivating $\overline{BMS}$ and activating normal external memory select.

The boot-kernel uses three copies of SYSCON – one that contains the original value of SYSCON, a second that contains SYSCON with the BS0 bit set (allowing the DSP to gain access to the boot EPROM), and a third with the BS0 bit cleared.

When BS0=1, the EPROM packing mode bits in the DMACx control register are ignored and 8-to-48 bit packing is forced. (8-bit packing is available only during EPROM booting or when BS0 is set.)

When an external port DMA channel is being used in conjunction with the BS0 bit, none of the other three channels may be used. In this mode, $\overline{BMS}$ is not asserted by a core processor access, but only by a DMA transfer. This allows the boot-kernel to perform other external accesses to non-boot memory.

The EPROM is automatically selected by the $\overline{\text{BMS}}$ pin after reset, and other memory select pins are disabled. The DSP's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The DMA external count register decrements after each EPROM transfer, and when zero is reached, DMA transfer stops and EP_{xx} is activated.

`RTI` returns the program counter to the address where kernel begins.

To EPROM boot a single processor system, include the executable on the command line without a switch. Do not use the `-id#exe` switch with `ID=0`.

The `WAIT` register `UBWM` (used for EPROM booting) is initialized at reset to both internal wait and external acknowledge required. The internal keeper-latch on the `ACK` pin initially holds acknowledge high (asserted). If acknowledge is driven low by another device during an EPROM boot, the keeper latch may latch acknowledge low.

The DSP views the deasserted (low) acknowledge as a hold off from the EPROM. In this condition, wait states are continually inserted, preventing completion of the EPROM boot. When writing a custom boot-kernel, change the `WAIT` register early within the boot-kernel so `UBWM` is set to internal wait mode (01).

Host Booting

ADSP-2106x/21160 DSPs accept data from a 16-bit host microprocessor (or other external device) through the external port `EPB0` and pack boot data into 48-bit instructions using an appropriate DMA channel. The host is selected when the `EBOOT` and `LBOOT` inputs are low and $\overline{\text{BMS}}$ is high. Configured for host booting, the DSP enters the slave mode after reset and waits for the host to download the boot program. [Table 5-12](#) lists host connections to DSPs.

Table 5-12. Host Connections to ADSP-2106x/21160 DSP

ADSP-	Connection/Data Bus Pins
21060/61/62	Host connected to DATA31-16 pins
21065L	Host connected to DATA15-0 pins
21160	Host connected to DATA47-32 pins
21xxx	PROM address pins connect to lowest address pins of any DSP
21xxx	Chip select connect to the $\overline{\text{BMS}}$ pin
21060/61/62/65L	Output enable connect to the $\overline{\text{RD}}$ pin
21160	Output enable connect to $\overline{\text{RDH}}$ pin

After reset, the DSP goes into an idle stage with:

- PC set to address 0x20004 (ADSP-21060/61/62 DSPs)
- PC set to address 0x8004 (ADSP-21065L DSP)
- PC set to address 0x40004 (ADSP-21160 DSP)

The parameter registers for the external port DMA channel (0, 8, or 10) are initialized as shown in [Table 5-10](#) and [Table 5-11](#), except that registers GPx, IEx and EMx are not initialized. However, no DMA transfers start.

The DMA Channel Control register (DMAC0 for ADSP-21060/61/62), (DMAC8 for ADSP-21065L), or (DMAC0 for ADSP-21160) is initialized, which allows external port DMA enable and selects DTYPE for instruction words, PMODE for 16-to-48 bit word packing, and least-significant-word first.

Because the host processor is accessing the EPB0 external port buffer, the HPM host packing mode bits of the SYSCON register must correspond to the external bus width specified by the PMODE bits of DMACx control register.

For a different packing mode, the host must write to `DMACx` and `SYSCON` to change the `PMODE` and `HPM` setting. The host boot file created by the loader requires the host processor to perform the following sequence of actions:

1. The host initiates the booting operation by asserting the DSP's $\overline{\text{HBR}}$ input pin, informing the DSP that the default 16-bit bus width is used. The host may optionally assert the $\overline{\text{CS}}$ chip select input to allow asynchronous transfers.
2. After the host receives the $\overline{\text{HBG}}$ signal from the DSP, the host can start downloading instructions by writing directly to the external port DMA buffer 0 or the host can change the reset initialization conditions of the DSP by writing to any of the `IOP` control registers. The host must use data bus pins as shown in [Table 5-12](#).
3. The host continues to write 16-bit words to `EPB0` until the entire program is boot-loaded. The host must wait between each host write to external port DMA buffer 0. A pause of at least 12 cycles must occur between the host releasing bus and next host bus-request.

After the host boot-loads the first 256 instructions (boot-kernel), the initial DMA transfers stop, and the boot-kernel:

1. Activates external port DMA channel interrupt (`EP0I`), stores the `DMACx` control setting in `R2` for later restore, clears `DMACx` for new setting, and sets the `BUSLCK` bit in the `MODE2` register to lock out the host.
2. Stores the `SYSCON` register value in `R12` for restore.
3. Enables interrupts and nesting for DMA transfer, sets up the `IMASK` register to allow DMA interrupts, and sets up the `MODE1` register to enable interrupts and allow nesting.

4. Loads the DMA Control register with `0x00A1` and sets up its parameters to read the data word by word from external buffer 0.

Each word is read into the reset vector address (refer to [Table 5-1 on page 5-4](#)) for dispatching. The data through this buffer has structure of boot section which could include more than one initialization block.

5. Clears the `BUSLCK` bit in the `MODE2` register to let the host write in the external buffer 0 right after the appropriate DMA channel is activated.

For information on the data structure of the boot section and initialization, see [“EPROM Booting” on page 5-15](#).

6. Loads the first 256-words of target the executable file during the final initialization stage, and then the kernel overwrites itself.

The final initialization works the same way as with EPROM booting, except that the `BUSLCK` bit in the `MODE2` register is cleared to allow the host to write to the external port buffer.

The default boot-kernel for host booting assumes `IMDW` is set to 0 during boot-loading, except during the final initialization stage. When using any power-up booting mode, the reset vector address (refer to [Table 5-1 on page 5-4](#)) must not contain a valid instruction because it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

If the kernel is going to initialize external memory, create a custom boot-kernel that sets appropriate values in the `SYSCON` and `WAIT` register. Be aware that the value in the DMA channel register is non-zero, and `IMASK` is set to allow DMA channel register interrupts. Because the DMA interrupt remains enabled in `IMASK`, this interrupt must be cleared before using the DMA channel again. Otherwise, unintended interrupts may occur.

A master SHARC DSP may boot a slave SHARC DSP by writing to its `DMACx` control register and setting the packing mode (`PMODE`) to 00. This allows instructions to be downloaded directly without packing. The wait state setting of 6 on the slave DSP does not affect the speed of the download since wait states affect bus master operation only.

Link Booting

 Link booting is supported on all SHARC DSPs except ADSP-21061 and ADSP-21065L DSPs.

When link booting SHARC DSPs, the DSP receives data from 4-bit link buffer 4 and packs boot data into 48-bit instructions using the appropriate DMA channels (`DMAC6` for ADSP-2106x, `DMAC8` for ADSP-21160).

Link mode is selected when the `EBOOT` is low and `LB00T` and $\overline{\text{BMS}}$ are high. The external device must provide a clock signal to the link port assigned to link buffer 4. The clock can be any frequency, up to a maximum of the DSP clock frequency. The clock's falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first. The link port acknowledge signal generated by the DSP can be ignored during booting since the link port cannot be pre-empted by another DMA channel.

Link booting is similar to host booting – the parameter registers (`IIx` and `Cx`) for DMA channels are initialized to the same values. The DMA Channel 6 Control register (`DMAC6`) is initialized to `0x00A0`, and the DMA Channel 10 Control register (`DMAC10`) is initialized to `0x100000`. This disables external port DMA and selects `DTYPE` for instruction words. The `LCTL` and `LCOM` link port control registers are overridden during link booting to allow link buffer 4 to receive 48-bit data.

After booting completes, the `IMASK` remains set, allowing DMA channel interrupts. This interrupt must be cleared before link buffer 4 is again enabled; otherwise, unintended link interrupts may occur.

Loader Guide

Refer to [“Host Booting” on page 5-19](#) for more information on booting process.

No Boot Mode

No boot mode causes the processor to start fetching and executing instructions at address 0x400004 (ADSP-2106x), 0x20004 (ADSP-21065L) and 0x800004 (ADSP-21160) in external memory space. All DMA control and parameter registers are set to their default initialization values.

Multiprocessor Booting

A multiprocessor system can be booted from a host processor, external EPROM, through a link port, or from external memory.

 Currently, the loader generates single-processor loader files for host and link port booting. As a result, the loader supports MP EPROM booting only. You must modify the application code to properly set up MP booting in host and link port booting modes.

To set up DMA processes to boot a single processor, refer to [“Boot Types” on page 5-15](#).

Multiprocessor EPROM Booting

The loader can produce boot-loadable files that permit the SHARC DSPs in a multiprocessor (MP) system to boot from a single EPROM. In such a system, the $\overline{\text{BMS}}$ signals from each SHARC DSP are ORed together to drive the chip select pin of the EPROM. Each SHARC DSP boots in turn, according to its priority. When the last DSP finishes booting, it must inform the DSPs to begin program execution.

Besides taking turns when booting, EPROM booting of multiple DSPs is similar to single-processor EPROM booting.

When booting an MP system through a single EPROM:

- Connect all $\overline{\text{BMS}}$ pins to EPROM.
- Processor ID#1 boots first. The other DSPs follow.
- The EPROM boot-kernel accepts multiple .EXE and .DXE files and reads the ID field in SYSTAT to determine which area of EPROM to read.
- All processors require a software flag or hardware signal (FLAG pins) to indicate that booting is complete.

When booting an MP system through an external port:

- The host can use the host interface.
- A SHARC DSP that is EPROM-, host-, or link-booted can boot the other DSPs through the external port (host boot mode).

For MP EPROM booting, you must select the **Multiprocessor** check box on the **Load** page of the **Project Options** dialog box or specify the `-id#exe` switch on the loader's command-line. These options specify the executable file targeted for a specific DSP.

Do not use the `-id#exe` switch to EPROM-boot a single processor whose ID is 0. Instead, name the executable on the command line without a switch. For a single processor with ID=1, use the `-id1exe` switch.

Multiple .DXE files can be specified in EPROM booting mode only. Multiple files may not be specified for host-port booting or link-port booting.

Running the Loader

This section provides reference information on loader options. You specify options via a loader command-line switches (command invocation) or via dialog box settings from the VisualDSP++ environment.

Load Page

When developing a “DSP loader file” project from VisualDSP++, modify the loader’s default options from the **Load** page (also called Load property page) of the **Projects Options** dialog box. These settings are used when you build the loader project. VisualDSP++ invokes the `elfloader` utility to build the output file.

The **Load** page buttons and fields correspond to loader command-line switches and parameters. Refer to [Figure 5-13](#). Use the **Additional Options** box to enter options that do not have dialog box equivalents.

For ADSP-21020 DSPs, the only permitted boot mode is JTAG; **-bJTAG** is automatically entered in the **Additional Options** box.

For more information, refer to VisualDSP++ online Help.

Loader Command Line

This section provides reference information about the loader command line. Switches and their descriptions appear in [Table 5-15 on page 5-30](#).

Invoke the `elfloader.exe` loader utility from the command line to generate a bootloader file. The loader uses the following command-line syntax:

```
elfloader sourcefile -proc processor -switch [-switch ...]
```

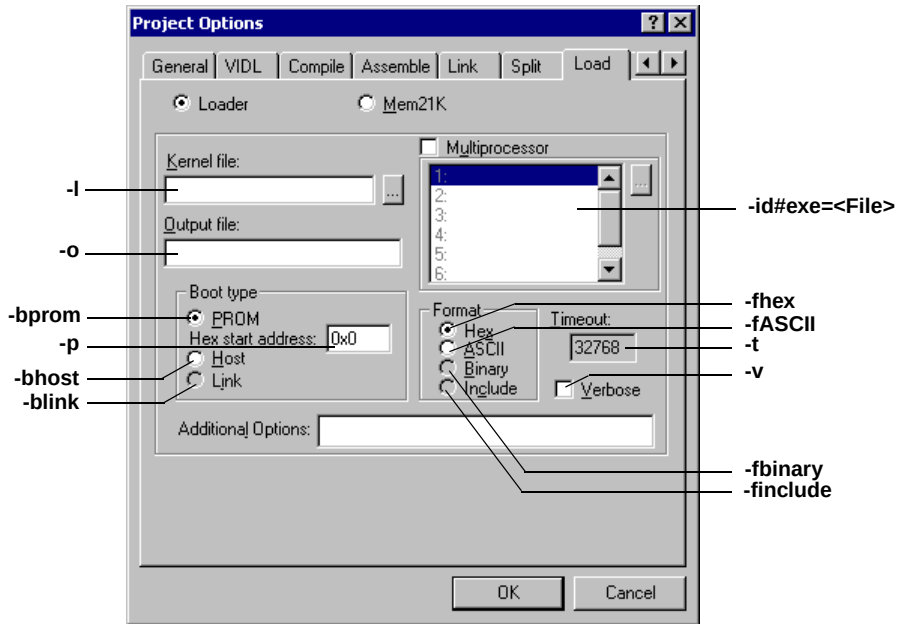


Table 5-13. Load Page of Project Options Dialog Box

Running the Loader

where:

- *sourcefile* – Identifies the executable file (.DxE) to be processed for a single-processor bootloadable file. You may include the drive, directory, file name and file extension. Enclose long file names within straight-quotes, “long file name”.
- *processor* – Specifies the processor (for example, ADSP-21065L) for which the bootloader file is to be built. If the processor is not specified, the default is ADSP-21062.
- *-switch* – Processes the optional switch. The loader has many switches to select operations and modes.



Command-line switches may be placed in the command line in any order.

Example

```
elfloader p0.dxe -bprom -fhex -l 060_prom.dxe -proc ADSP-21060
```

In the above example, the command line runs the loader with:

- *p0.dxe* – Identifies the executable file to process into a bootloadable file. Note the absence of the *-o* switch causes the output file name to default to *p0.ldr*.
- *-bprom* – Specifies EPROM booting as the boot-type for the bootloadable file.
- *-fhex* – Specifies Intel hex-32 format for the bootloadable file.
- *-l 060_prom.exe* – Specifies *060_prom.exe* as the boot-kernel file to be used in the bootloadable file.
- *-proc ADSP-21060* – Specifies the DSP type as ADSP-21060.

File Names

Many loader switches take a file name as an optional parameter.

[Figure 5-15](#) lists the expected file types, names, and extensions.

File searches are important in the loader's process. The loader supports relative and absolute directory names, and default directories. File searches occur as follows.

- *Specified path* – If you include relative or absolute path information in a file name, the loader searches only in that location for the file.
- *Default directory* – If you do not include path information in the file name, the loader searches for the file in the current working directory.

When providing an input or output file as a command-line parameter, use the following guidelines:

- Enclose long file names within straight quotes, "long file name".
- Append the appropriate file extension to each file.

File Extensions

The loader follows the conventions shown in [Table 5-14](#) for file extensions.

Table 5-14. File Extensions

Extension	File Description
.DXE	Executable files and boot-kernel files. The loader recognizes overlay memory (.OVL) files, but does not expect these files on the command line. Place .OVL files in the same directory as the .DXE file that refers to them so the loader can find them when processing the .LDR file.
.LDR	Loader output file for EPROMs

Running the Loader

Loader Command-Line Switches

Descriptions for each loader switch and file name appear in [Table 5-15](#).

Table 5-15. SHARC Loader Command-Line Items ¹

Switch	Description
<i>sourcefile</i>	A file name without a switch specifies an executable file for a single-processor bootloadable file. For MP bootloadable files, use the <code>-id#exe=file</code> switch.
<code>-proc processor</code> or <code>-dprocessor</code>	Specifies the processor. <code>-proc</code> is the preferred switch, and <code>-d</code> is for legacy support. <i>processor</i> is ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, or ADSP-21160.
<code>-bprom</code> <code>-bhost</code> <code>-blink</code> <code>-bJTAG</code>	Specifies the boot mode. The <code>-b</code> switch directs the loader to prepare a bootloadable file for the specified boot mode. Valid boot modes (boot types) include PROM, host, and link. For ADSP-21020 DSPs, JTAG is the only permitted boot mode. If <code>-b</code> does not appear on the command line, the default is <code>-bprom</code> . To use a custom kernel, the boot type specified with the <code>-b</code> switch must correspond to the boot-kernel selected with the <code>-l</code> switch.
<code>-caddress</code>	Custom option. This switch directs the loader to use the specified address. Valid addresses are: 20004 and 20040 (ADSP-2106x DSPs) 8004 and 8040 (ADSP-21065L DSPs) 40000 and 40050 (ADSP-21160 DSPs) The loader obtains the proper address even when this switch is absent from the command line.
<code>-e filename</code>	Except shared memory. This switch omits specified shared memory (<code>.SM</code>) file from the output loader file. Use this option to omit the shared parts of the executable from MP boot files. To omit multiple <code>.SM</code> file, use this switch/parameter multiple times in the command line. For example, to omit two files, use: <code>-e fileA.SM -e fileB.SM</code> . When preparing a bootloadable file for multiple processors that contains shared memory, each processor's boot image contains the shared memory items.

Table 5-15. SHARC Loader Command-Line Items ¹ (Cont'd)

Switch	Description
-fhex -fASCII -fbinary -finclude	Specifies the boot file's format. The -f switch prepares a bootloadable file in the specified format. Valid selections (Intel hex 32, ASCII, binary, or include) depend on the target processor and boot-type selection (-b switch). For a PROM boot-type, select hex or ASCII. For host or link booting, select ASCII, binary, or include format. If the -f switch does not appear on the command line, the default boot-type format is hex for PROM, and ASCII for HOST or LINK.
-h or -help	Command-line help. This switch outputs the list of command-line switches to standard output and exits. Type -proc ADSP-21xxx -h, where xxx is 060, 061, 062, 065L, or 160 to obtain help for SHARC DSPs. By default, the -h switch alone provides help for the loader driver.
-id#exe=file	Specifies the multiprocessor ID. The -id switch directs the loader to use the processor ID (#) for the corresponding executable file when producing a bootloadable file for EPROM or host booting an MP system. This switch is used only to produce a bootloadable file that boots multiple DSPs from a single EPROM. Valid values for # are 1, 2, 3, 4, 5, and 6. Do not use this switch for single-processor systems. For single-processor systems, use the executable filename as a parameter without a switch. For more information, refer to “Processor ID Numbers” on page 5-33 .
-NoKernel	Produces an .LDR file that does not include a kernel

Running the Loader

Table 5-15. SHARC Loader Command-Line Items ¹ (Cont'd)

Switch	Description
<code>-l kernelfile</code>	Directs the loader to use the specified <i>kernelfile</i> for the bootloading routine in the output bootloadable file. The boot-kernel file selected with this switch must correspond to the boot-type selected with the <code>-b</code> switch. If the <code>-l</code> switch does not appear on the command line, the loader searches for a default boot-kernel file in the installation directory. Based on the boot type (<code>-b</code> switch parameter), the loader searches in the processor-specific loader directory (...\\21k\\ldr) or (...\\211xx\\ldr) for the boot-kernel file. For <code>-bprom</code>, the default boot-kernel file is: 060_prom.dxe (ADSP-21060, ADSP-21061, and ADSP-21062 DSPs) 065L_prom.dxe (ADSP-21065L DSPs) 160_prom.dxe (ADSP-21160 DSPs) For <code>-bhost</code>, the default boot-kernel file is: 060_host.dxe (ADSP-21060, ADSP-21061, and ADSP-21062 DSPs) 065L_host.dxe (ADSP-21065L DSPs) 160_host.dxe (ADSP-21160 DSPs) For <code>-blink</code>, the default boot-kernel file is: 060_link.dxe (ADSP-21060 and ADSP-21062 DSPs) 065L_prom.dxe (ADSP-21065L DSPs) 160_prom.dxe (ADSP-21160 DSPs)
<code>-o filename</code>	Directs the loader to use the specified <i>filename</i> as the name for the loader's output file. If not specified, the default name is <i>sourcefile.LDR</i>.
<code>-p address</code>	PROM start address. Places the bootloadable file at the specified address in the EPROM. If the <code>-p</code> switch does not appear on the command line, the loader starts the EPROM file at address 0x0; this EPROM address corresponds to 0x400000 in ADSP-2106x DSPs.

Table 5-15. SHARC Loader Command-Line Items ¹ (Cont'd)

Switch	Description
-t <i>timeout</i>	(Host boot only) Specifies timeout cycles. This limits the number of cycles that the DSP spends initializing external memory. Valid values range from 3 to 32765 cycles; 32765 is the default. <i>timeout</i> is directly related to the number of cycles the DSP locks the bus for bootloading, instructing the DSP to lock the bus for no more than two times the <i>timeout</i> number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small <i>timeout</i> value.
-v	Verbose loader messages. This switch outputs status information as the loader processes files.

¹ Switches are optional. Items in *italics* are user-definable.

Processor ID Numbers

When used with a single-processor system, there is only one input (.DXE) file without any prefix and suffix to the input file name, for example:

```
elfloader -proc ADSP-21060 -bprom Input.dxe <switches>
```

A multiprocessor system requires a distinct processor ID number for each input file in the command line. A processor ID is provided via the -id#exe= switch, where # is an ID number (1 to 6).

In the following example, the loader processes the input file Input1.dxe for the processor with an ID of 1, and it processes the input file Input2.dxe for the processor with an ID of 2.

```
elfloader -proc ADSP-21060 -bprom -id1exe=Input1.dxe  
-id2exe=Input2.dxe <switch>
```

Running the Loader

6 ADSP-21161N LOADER

Use the loader utility (`elfloader.exe`) to convert executable files into bootloadable files for SHARC DSPs. A bootloadable file is transported to (and run from) DSP memory.

 This chapter describes loader and booting processes for ADSP-21161N DSPs only. Refer to [“ADSP-2106x/ADSP-21160 Loader” on page 5-1](#) for information about ADSP-21060, ADSP-21061, ADSP-21062, ADSP-21065L, and ADSP-21160 DSPs.

The loader processes executable (`.DXE`) files to generate a bootloadable file with an `.LDR` extension. To make a loadable file, the loader processes data from a boot-kernel file (`.DXE`) and one or more other executable files (`.DXE`). After fully debugging a program, use the loader to generate a set of bootloadable files for your target system.

This chapter contains the following information.

- [“Loader Guide”](#)
- [“Running the Loader”](#)

Refer to EE-notes on the Analog Devices DSP Web site for related information.

Loader Guide

The loader (`elfloader.exe`) processes executable files, producing a boot-loadable file. This file is used to automatically download programs to the processor's memory after power-up or a software reset. This process is called **booting**.

SHARC DSPs supports four booting modes: EPROM, host, link, and SPI. ADSP-21161 DSPs have a special hardware feature that boot-loads a small, 256-instruction program (called a boot-kernel or loader-kernel) into the DSP's internal memory at chip reset. When executed, the boot-kernel facilitates the booting of application code.

Before selecting options for the loader's operation, understand how the loader's features apply to the boot-type that you are using. This section describes boot-types for SHARC DSPs and relates loader features that support the appropriate boot-types.

This section described the following topics:

- [“Booting”](#)
- [“Boot Types” on page 6-12](#)
- [“Multiprocessor Booting” on page 6-24](#)

Bootling

ADSP-21161N DSPs support four bootling modes: EPROM, host processor, link port, and SPI, as shown in [Table 6-4 on page 6-9](#) and [Table 6-5 on page 6-15](#). Boot-loadable files for these modes pack boot data into 48-bit instructions and use an appropriate DMA channel of the DSP's DMA controller to boot-load the instructions.

- When bootling from an EPROM through the external port, the ADSP-21161 DSP reads boot data from an 8-bit external EPROM.
- When bootling from a host processor through the external port, the ADSP-21161 DSP accepts boot data from a 16- or 32-bit host microprocessor.
- When bootling through the link port, ADSP-21161 DSPs receive boot data through the link port as 4-bit-wide data in link buffer 4.
- When bootling through the SPI port, the ADSP-21161 DSP uses DMA channel 8 of the I/O processor to transfer instructions to internal memory. In this boot mode, the DSP receives data in the SPIRX register.
- For no boot mode, the ADSP-21161 DSP begin executing instructions from external memory.

After the boot process has loaded 256 words (called a boot-kernel or loader kernel) into memory, the processor begins to execute instructions. Because most applications require more than 256 words of instructions and initialization data, the boot-kernel typically serves as a loading routine for the application to load an entire program.

Software developers who use the loader should be familiar with the following operations:

- [“Power-Up Bootling Process” on page 6-4](#)
- [“Boot Mode Selection” on page 6-4](#)

Loader Guide

- [“Boot-Kernels” on page 6-5](#)
- [“Boot-Kernel Modification and Loader Issues” on page 6-6](#)
- [“Loader File Header Words” on page 6-9](#)
- [“Interrupt Vector Table Location” on page 6-10](#)

Power-Up Booting Process

ADSP-21161 DSPs have a hardware feature that boot-loads a small, 256-instruction program (called a boot-kernel or a loader kernel) into the DSP’s internal memory at chip reset. When executed, the boot-kernel facilitates booting user application code. The combination of the boot-kernel and the application code comprise the boot-loadable file.

At power-up, the general booting process includes the following steps.

1. A DMA channel is automatically configured for a 256 instruction (48-bit) transfer.

The first 256 instructions are called the boot-kernel.

2. The boot-kernel then runs and loads the application executable code and data.
3. The boot-kernel overwrites itself with the application’s first 256 words.

The boot-kernel can come from an external PROM, a host processor, or another SHARC DSP. The boot type selection directs the system to prepare the appropriate boot-kernel.

Boot Mode Selection

The boot mode is selected using the `LBOOT`, `EBOOT`, and `$\overline{\text{BMS}}$` pins. [Table 6-1](#) and [Table 6-2](#) show how the booting mode is selected for ADSP-21161N DSPs.

Table 6-1. Boot Mode Pins (Descriptions)

Pin	Type	Description
EBOOT	I	EPROM Boot – When EBOOT is high, the DSP boot-loads from an 8-bit EPROM through the DSP's external port. When EBOOT is low, the LBOOT and $\overline{\text{BMS}}$ pins determine booting mode.
LBOOT	I	Link Boot – When LBOOT is high (and EBOOT is low), the DSP boots from another SHARC DSP through the DSP's link port. When LBOOT is low (and EBOOT is low), the DSP boots from a host processor through the DSP's external port.
$\overline{\text{BMS}}$	I/O/T ¹	Boot Memory Select – When boot-loading from EPROM (EBOOT=1 and LBOOT=0), this pin is an <i>output</i> and serves as the chip select for the EPROM. In an MP system, $\overline{\text{BMS}}$ is output by the bus master. When host-booting, link-booting, or SPI-booting, (EBOOT=0), $\overline{\text{BMS}}$ is an <i>input</i> and must be high.

1 Three-statable in EPROM boot mode (when $\overline{\text{BMS}}$ is an output).

Table 6-2. Boot Mode Pins (States)

EBOOT	LBOOT	$\overline{\text{BMS}}$	Booting Mode
1	0	Output	EPROM (connect $\overline{\text{BMS}}$ to EPROM chip select)
0	0	1 (Input)	Host processor
0	1	1 (Input)	Link port
0	1	0 (Input)	Serial port (SPI)
0	0	0 (Input)	No boot (processor executes from external memory)

Boot-Kernels

Boot-loading begins by downloading the boot-kernel into the DSP's memory. The boot-kernel sets up the DSP and loads boot data. After the boot-kernel finishes initializing the rest of the system, the boot-kernel loads boot data over itself with a final DMA transfer.

Four boot-kernels ship with VisualDSP++; refer to [Table 6-3](#).

Table 6-3. Default Boot-Kernel Files

PROM	Link	Host	SPI
161_prom.asm	161_link.asm	161_host.asm	161_spi.asm

At DSP reset, a boot-kernel is loaded into the `seg_ldr` memory segment as defined in `161_ldr.ldf`, which is stored in the DSP tools installation directory (`...\211xx\ldr`).

Boot-Kernel Modification and Loader Issues

The loader reads a DSP executable file (`.DXE`) to produce a boot-loadable file (`.LDR`). However, the loader cannot determine how the DSP's `SYSCON`, `WAIT`, and `SDRAM` registers are to be configured for external memory loading.

Boot-kernel customization is required for some systems. The operation of other tools (such as the C/C++ compiler) is influenced by whether the loader is used.

If you do not specify a boot-kernel file via the **Load** page of the **Project Options** dialog box in VisualDSP++ (or via the loader's `-l` command-line switch), the loader places a default boot-kernel in the loader output file based on the specified boot type.

Rebuilding a Boot-Kernel File

If you modify the boot-kernel source file (`.ASM`) by inserting correct values for your system, you must rebuild the boot-kernel before generating the boot-loadable file. The boot-kernel source file contains default values for `SYSCON`. The `WAIT`, `SDCTL`, and `SDRAM` initialization code are in boot-kernel file's comments.

To Modify a Boot-Kernel Source File

1. Copy the applicable boot-kernel source file (161_link.asm, 161_host.asm, 161_prom.asm, or 161_spi.asm).
2. Apply the appropriate initializations of the SYSCON and WAIT registers.

After modifying the boot-kernel source file, rebuild the boot-kernel (.DXE) file. You can do this from within VisualDSP++ (refer to VisualDSP++ online Help for details). You can also rebuild a boot-kernel file from a command prompt window as described next.

Rebuilding a Boot-Kernel Using Command Lines

Rebuild a boot-kernel using command lines as follows.

EPROM Booting. The default boot-kernel source file for EPROM booting is 161_prom.asm. After copying the default file to my_prom.asm and modifying it to suit your system, use the following command lines to rebuild the boot-kernel.

```
easm21k -proc ADSP-21161 my_prom.asm  
linker -T 161_ldr.ldf my_prom.doj
```

Host Booting. The default boot-kernel source file for host booting is 161_host.asm. After copying the default file to my_host.asm and modifying it to suit your system, use the following command lines to rebuild the boot-kernel:

```
easm21k -proc ADSP-21161 my_host.asm  
linker -T 161_ldr.ldf my_host.doj
```

Loader Guide

Link Booting. The default boot kernel source file for link booting is `161_link.asm`. After copying the default file to `my_link.asm` and modifying it to suit your system, use the following command lines to rebuild the boot-kernel:

```
easm21k -proc ADSP-21161 my_link.asm  
linker -T 161_ldr.ldf my_link.doj
```

SPI Booting. The default boot kernel source file for link booting is `161_SPI.asm`. After copying the default file to `my_SPI.asm` and modifying it to suit your system, use the following command lines to rebuild the boot-kernel:

```
easm21k -proc ADSP-21161 my_SPI.asm  
linker -T 161_ldr.ldf my_SPI.doj
```

Loader File Issues

If you modify the boot-kernel for the EPROM, host, SPI, or link booting modes, ensure that the `seg_ldr` memory segment is defined in the `.LDF` file. Refer to the source of this memory segment in the `.LDF` file located in the `...\211xx\ldr\` directory.

Because the loader uses this address for the first location of the reset vector during the boot-load process, avoid placing code at this address. When using any of the DSP's power-up booting-modes, ensure that this address does not contain a critical instruction, because this address is not executed during the booting sequence. Place a `NOP` or `IDLE` in this location. The loader generates a warning if the vector address `0x40004` does not contain `NOP` or `IDLE`.



When using VisualDSP++ to create the loader file, specify the name of the customized boot-kernel executable in the **Kernel file** box on the **Load** page of the **Project Options** dialog box.

Loader File Header Words

The loader inserts header words at the beginning of data blocks in the loader file. The boot-kernel uses header words to properly place data and instruction blocks into DSP memory. The header format for PROM, host, and link boot loader files is as follows.

```
0x00000000DDDD
0xAAAAAAAAALLL
```

In the above example, D = data block type tag, A = block start address, and L = block word length.

Data block headers for in a loader boot file for the SPI boot type consists of three 32-bit words.

For single-processor systems, the data header has three 32-bit words in SPI boot type, as follows:

0x00000LLL	First word. Data word length or data word count of the data block.
0xAAAAAAAA	Second word. Data block start address.
0x000000DD	Third word. Tag of data block type.

Data structures and packing schemes in each data block for MP systems are the same as those used for single-processor systems.

The loader kernel examines the tag to determine the type of data or instruction being loaded. [Table 6-1](#) lists ADSP-21161N DSP tags.

Table 6-4. ADSP-21161N DSP Loader Tags

Tag Number	Block Type	Tag Number	Block Type
0x0000	final init	0x000E	init pm48
0x0001	zero dm16	0x000F	zero dm64
0x0002	zero dm32	0x0010	init dm64

Table 6-4. ADSP-21161N DSP Loader Tags

Tag Number	Block Type	Tag Number	Block Type
0x0003	zero dm40	0x0011	zero pm64
0x0004	init dm16	0x0012	init pm64
0x0005	init dm32	0x0013	init pm8 ext
0x0006	init dm40	0x0014	init pm16 ext
0x0007	zero pm16	0x0015	init pm32 ext
0x0008	zero pm32	0x0016	init pm48 ext
0x0009	zero pm40	0x0017	zero pm8 ext
0x000A	zero pm48	0x0018	zero pm16 ext
0x000B	init pm16	0x0019	zero pm32 ext
0x000C	init pm32	0x001A	zero pm48 ext
0x000D	init pm40		

Interrupt Vector Table Location

If the ADSP-21161 DSP is booted from an external source (EPROM, host, link port, or SPI), the interrupt vector table is located in internal memory. If the DSP is not booted and executes from external memory (no boot mode), the vector table must be located in external memory.

The `IIVT` bit in the `SYSCON` control register can be used to override the booting mode in determining where the interrupt vector table is located. If the DSP is not booted (no boot mode), setting `IIVT` to 1 selects an internal vector table, and setting `IIVT=0` selects an external vector table. If the DSP is booted from an external source (any boot mode other than no boot), `IIVT` has no effect. The default initialization value of `IIVT` is zero.

Include-Format Files for Booting

Include-format files support booting for ADSP-21161N DSPs. The word width (8-bit, 16-bit, and 32-bit) depends on the specified boot type. Specify word width with the loader's `-HostWidth #` switch. Refer to [Table 6-11 on page 6-31](#) for information on command-line switches.

Similar to Intel hex-32 output, loader output files in include format have four basic parts in the following order:

1. PROM kernel
2. Multiprocessor jump table
3. User application code
4. Saved user code in conflict with the kernel code

The kernel code is the first part. The processor jump table (including the processor IDs and the associated jump addresses) follows. User application code is followed by the saved user code.

Refer to [“Loader Output Files in Include Format \(.LDR\)” on page A-10](#) for information about include-format files.

Boot Types

The following sections describe these boot types:

- [“EPROM Booting”](#)
- [“Host Booting” on page 6-17](#)
- [“Link Port Booting” on page 6-20](#)
- [“SPI Port Booting” on page 6-22](#)
- [“No Boot Mode” on page 6-24](#)

 For multiprocessor booting, refer to [“Multiprocessor Booting” on page 6-24](#).

EPROM Booting

EPROM booting through the external port is selected when the `EBOOT` input is high and the `LB00T` input is low. These settings cause the `BMS` pin to become an output, serving as chip select for the EPROM.

The `DMAC10` control register is initialized for booting packing boot data into 48-bit instructions. EPROM boot mode uses channel 10 of the I/O processor’s DMA controller to transfer the instructions to internal memory. For EPROM booting, the DSP reads data from an 8-bit external EPROM.

After the boot process loads 256 words into memory locations `0x40000` through `0x400FF`, the DSP begins to execute instructions. Because most DSP programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the

application. VisualDSP++ includes loading routines (loader kernels) that can load entire programs. See [“Boot-Kernels” on page 6-5](#) for more information.

-  Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.
-  Be aware that DMA channel differences between the ADSP-21161 DSPs and previous SHARC family DSPs (ADSP-2106x) account for booting differences. Even with these differences, the ADSP-21161 DSP supports the same boot capability and configuration as the ADSP-2106x DSPs. The `DMAC` register default values differ because the ADSP-21161 DSP has additional parameters and different DMA channel assignments. EPROM boot mode uses `EPB0`, DMA channel 10. Similar to ADSP-2106x DSPs, the ADSP-21161 DSP boots from `DATA23-16`.

The DSP determines the booting mode at reset from the `EBOOT`, `LB00T`, and `$\overline{\text{BMS}}$` pin inputs. When `EBOOT=1` and `LB00T=0`, the DSP boots from an EPROM through the external port and uses `$\overline{\text{BMS}}$` as the memory select output. For information on boot mode selection, see the Boot Memory Select pin descriptions in [Table 6-4 on page 6-9](#) and [Table 6-2 on page 6-5](#).

-  When using any of the power-up booting modes, address `0x40004` should not contain a valid instruction since it is not executed during the booting sequence. Place a `NOP` or `IDLE` instruction at this location.

EPROM booting through the external port requires that an 8-bit-wide boot EPROM be connected to the DSP's data bus pins 23-16 (`DATA23-16`). The DSP's lowest address pins should be connected to the EPROM's address lines. The EPROM's chip select should be connected to `$\overline{\text{BMS}}$` , and its output enable should be connected to `$\overline{\text{RD}}$` .

In an MP system, the $\overline{\text{BMS}}$ output is driven by the ADSP-21161N bus master only. This allows the wire-ORing of multiple $\overline{\text{BMS}}$ signals for a single common boot EPROM.

 Systems can boot any number of ADSP-21161N DSPs from a single EPROM using the same code for each processor or differing code for each processor.

During reset, the ACK line is internally pulled high with the equivalent of an internal 20-Kohm resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

When EPROM boot mode is configured, external port DMA channel 10 (DMAC10) becomes active following reset; it is initialized to $0x04A1$. This enables the external port DMA and selects DTYPE for instruction words. The packing mode bits (PMODE) are ignored, BS0 is set in SYSCON , and 8-bit-to-48-bit packing is forced with the least-significant-word first.

The RBWS and RBAM fields of the WAIT register are initialized to perform asynchronous access and generate seven wait states (eight cycles total) for the EPROM access in external memory space. Note that wait states defined for boot memory are applied to $\overline{\text{BMS}}$ -asserted accesses.

Table 6-2 shows how DMA channel 10 parameter registers are initialized at reset. The count register (CEP0) is initialized to $0x0100$ to transfer 256 words to internal memory. The external count register (ECEP0), used when external addresses are generated by the DMA controller, is initialized to $0x0600$ ($0x0100$ words at six bytes per word). The DMAC10 control register is initialized to
`0000 0160`.

The default value sets up external port transfers as follows:

- $\text{DEN} = 1$, external port enabled
- $\text{MSWF} = 0$, LSB first

- PMODE = 100, 8-bit to 48-bit packing
- DTYPE = 1, three-column data

Table 6-5. DMA Channel 10 Parameter Register Initialization for EPROM Booting

Parameter Register	Initialization Value
IIEP0	0x40000
IMEP0	Uninitialized (increment by 1 is automatic)
CEP0	0x100 (256 instruction words)
CPEP0	Uninitialized
GPEP0	Uninitialized
EIEP0	0x800000
EMEP0	Uninitialized (increment by 1 is automatic)
ECEP0	0x600 (256 words x 6 bytes/word)

The following sequence occurs at system start-up, when the DSP's $\overline{\text{RESET}}$ input goes inactive

1. The DSP goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to address 0x40004.
2. The DMA parameter registers for channel 10 are initialized as shown in [Table 6-2](#).
3. $\overline{\text{BMS}}$ becomes the boot EPROM chip select.
4. 8-bit Master Mode DMA transfers from EPROM to the first internal memory address on the external port data bus lines 23-16.

5. The external address lines (ADDR23-0) start at 0x800000 and increment after each access.
6. The $\overline{RD}$ strobe asserts as in a normal memory access with seven wait states (eight cycles).

The DSP's DMA controller reads the 8-bit EPROM words, packs them into 48-bit instruction words, and transfers them to internal memory until 256 words have been loaded. The EPROM is automatically selected by the $\overline{BMS}$ pin; other memory select pins are disabled.

The DMA external count register (ECEP0) decrements after each EPROM transfer. When ECEP0 reaches zero, the following wake-up sequence occurs:

1. DMA transfers stop.
2. External port DMA channel 10 interrupt (EP0I) is activated.
3. $\overline{BMS}$ is deactivated, and normal external memory selects are activated.
4. The DSP vectors to the EP0I interrupt vector at 0x40050.

At this point the DSP has completed its booting mode and is executing instructions normally. The first instruction at the EP0I interrupt vector location, address 0x40050, should be an RTI (return from interrupt). This process returns execution to the reset routine at location 0x40005 where normal program execution can resume. After reaching this point, a program can write a different service routine at the EP0I vector location 0x40050.

Host Booting

The DSP can boot from a host processor through the external port. Host booting is selected when the `EBOOT` and `LBOOT` inputs are low and `BMS` is high. Configured for host booting, the DSP enters the slave mode after reset and waits for the host to download the boot program.

The `DMAC10` control register is initialized for booting, packing boot data into 48-bit instructions. Channel 10 of the I/O processor's DMA controller is used to transfer instructions to internal memory. DSPs accept data 8-, 16-, or 32-bit host microprocessor (or other external devices).

After the boot process loads 256 words into memory locations `0x40000` through `0x400FF`, the DSP begins executing instructions. Because most DSP programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the application. VisualDSP++ includes loading routines (loader kernels) that can load entire programs. These routines come with the development tools. Refer to [“Boot-Kernels” on page 6-5](#) for more information.

-  Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.
-  DMA channel differences between ADSP-21161 DSPs and previous SHARC family DSPs (ADSP-2106x) account for booting differences. Even with these differences, ADSP-21161 DSPs support the same boot capability and configuration as ADSP-2106x DSPs. The `DMAC` register default values differ because the ADSP-21161 DSP has additional parameters and different DMA channel assignments. Host boot mode uses `EPB0`, DMA channel 10. Similar to ADSP-2106x DSPs, the ADSP-21161 DSP boots from `DATA23-16`.

The DSP determines the boot mode at reset from the `EBOOT`, `LBOOT`, and `BMS` pin inputs. When `EBOOT=0`, `LBOOT=0`, and `BMS=1`, the DSP boots from a host through the external port. Refer to [Table 6-1 on page 6-5](#) and [Table 6-2 on page 6-5](#) on boot mode selection.

When using any of the power-up booting modes, address 0x40004 should not contain a valid instruction since it is not executed during the booting sequence. Place a NOP or IDLE instruction at this location.

During reset, the DSP's ACK line is internally pulled high with an equivalent 20-Kohm resistor and is held high with an internal keeper latch. It is not necessary to use an external pull-up resistor on the ACK line during booting or at any other time.

When host booting mode is configured, external port DMA channel 10 (DMAC10) becomes active following reset; it is initialized to 0x04A1. This enables external port DMA and selects DTYPE for instruction words. The packing mode bits (PMODE) are ignored, BSO is set in SYSCON, and 8-bit-to-48-bit packing is forced with least-significant-word first.

Table 6-5 on page 6-15 shows how the DMA channel 10 parameter registers are initialized at reset for host booting. The count register (CEP0) is initialized to 0x0100 to transfer 256 words to internal memory. The external count register (ECEP0), which is used when external addresses are generated by the DMA controller, is initialized to 0x0600 (0x0100 words at six bytes per word). The DMAC10 control register is initialized to 0000 0160.

The default value sets up external port transfers as follows:

- DEN = 1, external port enabled
- MSWF = 0, LSB first
- PMODE = 100, 8- to 48-bit packing
- DTYPE = 1, three-column data

Table 6-6. Host Booting – DMA Channel 10 Parameter Register Initialization

Parameter Register	Initialization Value
IIEP0	0x0004 0000
IMEP0	Uninitialized (increment by 1 is automatic)
CEP0	0x0100 (256 instruction words)
CPEP0	Uninitialized
GPEP0	Uninitialized
EIEP0	0x0080 0000
EMEP0	Uninitialized (increment by 1 is automatic)
ECEP0	0x0600 (256 words x 6 bytes/word)

At system start-up, when the DSP's $\overline{\text{RESET}}$ input goes inactive, the following sequence occurs:

1. The DSP goes into an idle state, identical to that caused by the IDLE instruction. The program counter (PC) is set to address 0x40004.
2. The DMA parameter registers for channel 10 are initialized as shown in [Table 6-5 on page 6-15](#).
3. The host uses HBR and CS to arbitrate for the bus.
4. The host can write to SYSCON to change boot width from default.
5. The host writes boot information to external port buffer 0.

Loader Guide

The DMA external count register (`ECEP0`) decrements after each transfer. When `ECEP0` reaches zero, the following wake-up sequence occurs:

1. The DMA transfers stop.
2. The external port DMA channel 10 interrupt (`EP0I`) is activated.
3. The DSP vectors to the `EP0I` interrupt vector at `0x40050`.

At this point the DSP has completed its booting mode and is executing instructions normally. The first instruction at the `EP0I` interrupt vector location, address `0x40050`, should be an `RTI` (return from interrupt). This process returns execution to the reset routine at location `0x40005` where normal program execution can resume. After reaching this point, a program can write a different service routine at the `EP0I` vector location `0x40050`.

Link Port Booting

Link port booting uses DMA channel 8 of the I/O processor to transfer instructions to internal memory. In this boot mode, the DSP receives 4-bit-wide data in link buffer 0.

After the boot process loads 256 words into memory locations `0x40000` through `0x400FF`, the DSP begins to execute instructions. Because most DSP programs require more than 256 words of instructions and initialization data, the 256 words typically serve as a loading routine for the

application. VisualDSP++ includes loading routines (loader kernels) that load an entire program through the selected port. Refer to [Table 6-3 on page 6-6](#) for more information.

-  Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.
-  DMA channel differences between ADSP-21161 DSPs and previous SHARC family DSPs (ADSP-2106x) account for booting differences. Even with these differences, ADSP-21161 DSPs support the same boot capability and configuration as ADSP-2106x DSPs.

The DSP determines the booting mode at reset from the $\overline{\text{EBOOT}}$, $\overline{\text{LBOOT}}$, and $\overline{\text{BMS}}$ pin inputs. When $\overline{\text{EBOOT}}=0$, $\overline{\text{LBOOT}}=1$, and $\overline{\text{BMS}}=1$, the DSP boots through the link port. For information on boot mode selection, see [Table 6-1 on page 6-5](#) and [Table 6-2 on page 6-5](#).

-  When using any of the power-up booting modes, address 0×40004 should not contain a valid instruction since it is not executed during the booting sequence. Place a NOP or IDLE instruction at this location.

In link port booting, the DSP gets boot data from another DSP's link port or 4-bit-wide external device after system power-up.

The external device must provide a clock signal to the link port assigned to link buffer 0. The clock can be any frequency up to the DSP clock frequency. The clock's falling edges strobe the data into the link port. The most significant 4-bit nibble of the 48-bit instruction must be downloaded first.

[Table 6-5 on page 6-15](#) shows how the DMA channel 8 parameter registers are initialized at reset. The count register (CLB0) is initialized to 0×0100 to transfer 256 words to internal memory. The LCTL register is overridden during link port booting to allow link buffer 0 to receive 48-bit data.

Table 6-7. Link Port Booting – DMA Channel 8 Parameter Register Initialization

Parameter Register	Initialization Value
IILB0	0x0004 0000
IMLB0	Uninitialized (increment by 1 is automatic)
CLB0	0x0100 (256 instruction words)
CPLB0	Uninitialized
GPLB0	Uninitialized

In systems where multiple DSPs are not connected by the parallel external bus, booting can be accomplished from a single source through the link ports. To simultaneously boot all the DSPs, make a parallel common connection to link buffer 0 on each of the processors. If a daisy chain connection exists between the processors' link ports, each DSP can boot the next one in turn. Link buffer 0 must always be used for booting.

SPI Port Booting

Serial Peripheral Interface (SPI) port booting uses DMA channel 8 of the I/O processor to transfer instructions to internal memory. In this boot mode, the DSP receives 8-bit-wide data in the `SPIRX` register.

During the booting process, the program loads 256 words into memory locations `0x40000` through `0x400FF`. The DSP subsequently begins executing instructions. Because most DSP programs require more than 256 words of instructions and initialization data, the 256 words typically serve

as a loading routine for the application. VisualDSP++ includes loading routines (loader kernels) that load an entire program through the selected port. See [“Bootting” on page 6-3](#) for more information.

 Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations. For information about SPI slave booting, refer to EE-177, entitled *SHARC SPI Booting*, located on the Analog Devices DSP Web site.

The DSP determines the booting mode at reset from the $\overline{\text{EB00T}}$, $\overline{\text{LB00T}}$, and $\overline{\text{BMS}}$ pin inputs. When $\overline{\text{EB00T}}=0$, $\overline{\text{LB00T}}=1$, and $\overline{\text{BMS}}=0$, the DSP boots through its SPI port. For information on boot mode selection, see [Table 6-1 on page 6-5](#) and [Table 6-2 on page 6-5](#).

 When using any of the power-up booting modes, address $0x40004$ should not contain a valid instruction because it is not executed during the booting sequence. Place a NOP or IDLE instruction placed at this location.

For SPI port booting, the DSP gets boot data after system power-up from another DSP's SPI port or another SPI-compatible device.

[Table 6-8](#) shows how the DMA channel 8 parameter registers are initialized at reset. The SPI Control Register (SPICTL) is configured to $0x0A001F81$ upon reset during SPI boot.

This configuration sets up the SPIRX register for 32-bit serial transfers. The SPIRX DMA channel 8 parameter registers are configured to DMA in $0x180$ 32-bit words into internal memory normal word address space starting at $0x40000$. Once the 32-bit DMA transfer completes, the data is then accessed as 3-column, 48-bit instructions. The DSP executes a 256-word ($0x100$) loader kernel upon completion of the 32-bit, $0x180$ word DMA.

For 16-bit SPI hosts, two words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs. For 8-bit SPI hosts, four words are shifted into the 32-bit receive shift register before a DMA transfer to internal memory occurs.

Table 6-8. SPI Port Booting – DMA Channel 8 Parameter Register Initialization

Parameter Register	Initialization Value
IISRX	0x0004 0000
IMSRX	Uninitialized (increment by 1 is automatic)
CSRX	0x0180 (256 instruction words)
GPSRX	Uninitialized

No Boot Mode

No boot mode causes the processor to start fetching and executing instructions at address 0x800004 in external memory space. In no boot mode, the processor does not bootload and all DMA control and parameter registers are set to their default initialization values.

Multiprocessor Booting

A multiprocessor (MP) system can be booted from a host processor, external EPROM, through a link port, or through an SPI port.

 Currently, the loader generates single-processor loader files for host, link, and SPI port booting. As a result, the loader supports multiprocessor EPROM booting only. You must modify the application code to properly set up multiprocessor booting in host, link, and SPI port booting modes.

To set up DMA processes for booting a single processor, refer to:

- [“EPROM Booting” on page 6-12](#)
- [“Host Booting” on page 6-17](#)

- “Link Port Booting” on page 6-20
- “SPI Port Booting” on page 6-22



Refer to the *ADSP-21161 SHARC DSP Hardware Reference* for detailed information on DMA and system configurations.

Multiprocessor EPROM Booting

There are two methods by which an MP system can be booted: booting from a single EPROM, and sequential EPROM booting. Regardless of the method, processors perform the following steps.

1. Arbitrate for the bus.
2. Upon becoming bus master, DMA the 256 word boot stream.
3. Release the bus.
4. Execute the loaded instructions.

Bootting From a Single EPROM

The $\overline{\text{BMS}}$ signals from each DSP may be wire-ORed together to drive the EPROM's chip select pin. Each DSP can boot in turn, according to its priority. When the last DSP has finished booting, it must inform the other DSPs (which may be in the idle state) that program execution can begin (if all DSPs are to begin executing instructions simultaneously).

When multiple DSPs boot from a single EPROM, the DSPs can boot identical code or different code from the EPROM. If the processors load differing code, use a jump table (based on processor ID) to select the code for each processor.

Sequential EPROM Booting

Set the $\overline{\text{EB00T}}$ pin of the DSP with $\text{IDx}=1$ high for EPROM booting. The other DSPs should be configured for host booting ($\text{EB00T}=0$, $\text{LB00T}=0$, and $\text{BMS}=1$), leaving them in the idle state at startup and allowing the DSP with $\text{IDx}=1$ to become bus master and boot itself. Connect the $\overline{\text{BMS}}$ pin of DSP #1 only to the EPROM's chip select pin. When DSP #1 has finished booting, it can boot the remaining DSPs by writing to their external port DMA buffer 0 (EPB0) via the multiprocessor memory space.

The loader can produce boot-loadable files that permit SHARC DSPs in an (MP) system to boot from a single EPROM. In such a system, the DSPs' BMS signals are ORed together to drive the EPROM's chip select pin. Each DSP boots in turn, according to its priority. When the last DSP has finished booting, it must inform the other DSPs to begin program execution.

Running the Loader

This section provides reference information on loader options specified from VisualDSP++ or from a command line.

Load Page

When developing a “DSP loader file” project in VisualDSP++, you can modify the loader’s default option settings on the **Load** page of the **Projects Options** dialog box. Refer to [Figure 6-9](#). VisualDSP++ invokes the `elfloader` utility to build the output file.

Dialog box buttons and fields correspond to command-line switches and parameters. Use the **Additional Options** box to enter options that have no dialog box equivalent.

For more information, refer to VisualDSP++ online Help.

Loader Command Line

This section provides reference information on the loader command line. A list of all switches and their descriptions appears in [Table 6-11 on page 6-31](#).

Invoke the `elfloader.exe` loader utility from the command line to generate a bootloader file. This section describes the command line only.

Use the following syntax for the SHARC DSP loader command line.

```
elfloader sourcefile -proc ADSP-21161 -switch [-switch ...]
```

Running the Loader

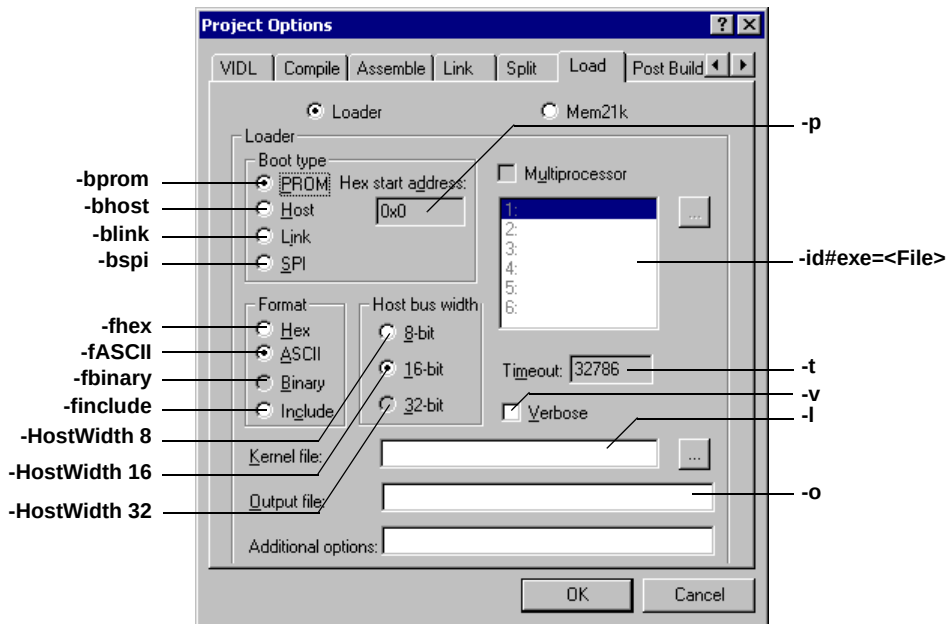


Table 6-9. Project Options Dialog Box – Load Page

where:

- *sourcefile* – Identifies the executable file (.DXE) to be processed for a single-processor bootloadable file. A file name can include the drive, directory, file name and file extension. Enclose long file names within straight-quotes, “long file name”.
- ADSP-21161 – Specifies the processor for which the bootloader file is to be built.
- -switch – Processes the optional switch. The loader has many switches to select operations and modes.



Command-line switches are not case sensitive and may be placed in the command line in any order.

Single-Processor Systems

```
elfloader Input.dxe -bSPI -proc ADSP-21161
```

In the above example for a single-processor system, the command line runs the loader with:

- Input.dxe – Identifies the executable file to process into a bootloadable file. Note that the absence of the -o switch causes the output file name to default to Input.ldr.
- -bSPI – Specifies SPI port booting as the boot-type for the bootloadable file.
- -proc ADSP-21161 – Specifies ADSP-21161 as the DSP target processor.

Multiprocessor Systems

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input1.dxe  
-id2exe=Input2.dxe <switch>
```

Running the Loader

In the above example for an MP system, the command line runs the loader with:

- `-proc ADSP-21161` – Specifies ADSP-21161 as the DSP target processor.
- `-bprom` – Specifies EPROM booting as the boot-type for the boot-loadable file.
- `id1exe=Input1.dxe` – Identifies the executable file to processed for ID#1 as Input1.dxe.
- `id2exe=Input2.dxe` – Identifies the executable file to processed for ID#2 as Input2.dxe.

File Names

Many loader switches take a file name as an optional parameter.

[Figure 6-11](#) lists expected file types, names, and extensions.

File searches are important in the loader's process. The loader supports relative and absolute directory names, and default directories. File searches occur as follows.

- *Specified path* – If you include relative or absolute path information in a file name, the loader searches only in that location for the file.
- *Default directory* – If you do not include path information in the file name, the loader searches for the file in the current working directory.

When providing an input or output file name as a command-line parameter, follow these 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 shown in [Table 6-10](#) for file extensions.

Table 6-10. File Extensions

Extension	File Description
.DXE	Executable files and boot-kernel files. The loader recognizes overlay memory files (.OVL), but does not expect these files on the command line. Place .OVL files in the same directory as the .DXE file that refers to them so the loader can find them when processing the .LDR file. The .OVL files may also be placed in the .OVL file output directory specified in the .LDF file
.LDR	Loader output file for EPROMs

ADSP-21161 Loader Command-Line Switches

A descriptions of each ADSP-21161 loader switch appears in [Table 6-11](#).



Do not place space between the switch and the parameter.
For example, there is no space between `-b` and `PROM`.

Table 6-11. ADSP-21161N Loader Command-Line Items ¹

Switch	Description
<i>sourcefile</i>	A file name without a switch specifies an executable file for a single-processor bootloadable file. For MP system bootloadable files, use the <code>-id#exe=file</code> switch, where # is a number (1 to 6, inclusive). The loader accepts .DXE files only and automatically finds and processes associated .OVL and .SH files. Place .OVL and .SH files in the specified directory and in the current working directory.
<code>-proc ADSP-21161</code> or <code>-dADSP-21161</code>	Specifies the processor. <code>-proc ADSP-21161</code> is the preferred switch. The <code>-d</code> switch (<code>-dADSP-21161</code>) is for legacy support.

Running the Loader

Table 6-11. ADSP-21161N Loader Command-Line Items (Cont'd) ¹

Switch	Description
-bprom -bhost -bspi -blink	Specifies the boot mode. The <code>-b</code> switch directs the loader to prepare a bootloadable file for the specified boot mode. The valid modes (boot types) are PROM, host, SPI, and link. If <code>-b</code> does not appear on the command line, the default is <code>-bprom</code> . To use a custom boot-kernel, the boot-type selected with the <code>-b</code> switch must correspond with the boot-kernel selected with the <code>-l</code> switch. Otherwise, the loader automatically selects a default boot-kernel based on the selected boot-type.
-efilename	Except shared memory. The <code>-e</code> switch omits the listed shared memory (<code>.SM</code>) file from the output loader file. If you prepare a bootloadable file for a multiprocessor system that contains shared memory, each processor's boot image contains the shared memory items. This option omits the shared parts of the executable from multiprocessor boot files Repeat this switch for each <code>.SM</code> file to be omitted.
-fhex -fascii -fbinary -finclude	Species the bootloadable file's format. The <code>-f</code> switch prepares a bootloadable file in the specified <i>format</i> . Available <i>format</i> selections include hex (Intel hex-32), ascii, include, and binary. Available formats depend on the <code>-b</code> switch's boot-type selection. For a PROM boot-type, select a hex, ascii, or include format. For host or link booting, select an ascii, binary, or include format. For SPI booting, select an ASCII or binary format. If the <code>-f</code> switch does not appear on the command line, the default boot-type format is hex for PROM, and ASCII for host, link, or SPI.
-h or -help	Command-line help. This switch outputs the list of command-line switches to standard output and exits. Combining the <code>-h</code> switch with <code>-proc ADSP-21161</code> ; for example, <code>elfloader -proc ADSP-21161 -h</code> , yields the syntax and switches for the elfloader utility for ADSP-21161 DSPs. By default, the <code>-h</code> switch alone provides help for the loader driver.
-HostWidth #	Specifies the width (in bits) of the host bus. This sets up the <code>.LDR</code> file's word width. By default, the word width for PROM and host is 8, for link is 16, and for SPI is 32. The valid word widths for the various boot modes are: PROM (8 for hex or ASCII format, 8 or 16 for include format), host (8 or 16 for ASCII or binary format, 16 for include format), link (16 for ASCII, binary, or include format), SPI (8, 16, or 32 for hex or ASCII format).

Table 6-11. ADSP-21161N Loader Command-Line Items (Cont'd) ¹

Switch	Description
<code>-id#exe=filename</code>	Specifies the multiprocessor ID. The <code>-id</code> switch directs the loader to use the processor ID (#) for the corresponding executable files when producing a bootloadable file for an MP system. This switch is used only to produce a bootloadable file that boots multiple DSPs from a single EPROM. Valid values for # are 1, 2, 3, 4, 5, and 6. Do not use this switch for single-processor systems; instead, use the executable <i>filename</i> as a parameter without a switch. Refer to “Processor ID Numbers” on page 6-34 for details.
<code>-l kernelfile</code>	The <code>-l</code> switch directs the loader to use the specified <i>kernelfile</i> for the bootloading routine in the output bootloadable file. Note that the boot-kernel file selected with this switch must correspond to the boot-type (selected with the <code>-b</code> switch). If the <code>-l</code> switch does not appear on the command line, the loader searches for a default boot-kernel file in the installation directory. Based on the boot-type, the loader searches for the boot-kernel file in the installation directory (...\\21k\\ldr) as follows: For a PROM boot-type (<code>-bprom</code>), the default boot-kernel file is <code>161_prom.dxe</code>. For a host boot-type (<code>-bhost</code>), the default boot-kernel file is <code>161_host.dxe</code>. For a link boot-type (<code>-blink</code>), the default boot-kernel file is <code>161_link.dxe</code>. For an SPI boot-type (<code>-bspi</code>), the default boot-kernel file is <code>161_SPI.dxe</code>.
<code>-o filename</code>	Specifies an output file. The <code>-o</code> switch directs the loader to use the specified <i>filename</i> for the name of the loader's output file. If the <code>-o</code> switch is not specified, the default name is <i>sourcefile.LDR</i>.
<code>-p address</code>	Specifies the PROM start address. The <code>-p</code> switch starts the bootloadable file at the specified address in the EPROM. If the <code>-p</code> switch does not appear on the command line, the loader starts the EPROM file at address 0x0 in the EPROM.

Running the Loader

Table 6-11. ADSP-21161N Loader Command-Line Items (Cont'd) ¹

Switch	Description
<i>-t timeout</i>	(Host boot-type only) Timeout cycles. The <i>-t</i> switch (for example, <i>-t100</i>) limits the number of cycles that the DSP can spend initializing external memory. Valid values range from 3 to 32765 cycles; 32765 is the default value. The <i>timeout</i> value is directly related to the number of cycles the DSP locks the bus for bootloading, instructing the DSP to lock the bus for no more than two times the <i>timeout</i> number of cycles. When working with a fast host that cannot tolerate being locked out of the bus, use a relatively small <i>timeout</i> value.
<i>-v</i>	Verbose loader messages. The <i>-v</i> switch outputs status information as the loader processes files.

¹ Switches are optional. Items in *italics* are user-definable.

Processor ID Numbers

When used with a single-processor system, there is only one input (.DXE) file without any prefix and suffix to the input file name, for example:

```
elfloader -proc ADSP-21161 -bprom Input.dxe <switches>
```

A multiprocessor system requires a distinct processor ID number for each input file in the command line. A processor ID is provided via the *-id#exe=* switch, where *#* is an ID number (1 to 6).

In the following example, the loader processes the input file *Input1.dxe* for the processor with an ID of 1, and it processes the input file *Input2.dxe* for the processor with an ID of 2.

```
elfloader -proc ADSP-21161 -bprom -id1exe=Input1.dxe  
-id2exe=Input2.dxe <switch>
```

7 SPLITTER

Load the linker output into VisualDSP++ for simulation. Once you have fully debugged your program, use the splitter utility (`elfspl21k.exe`) to generate non-bootable PROM image files, which execute from DSP external memory. These files are often used with ADSP-21065L DSP systems, which have limited internal memory.

In most instances, developers working with SHARC DSPs should use the loader instead of the splitter – the exception is when a SHARC system must execute instructions from external memory.

This chapter provides the following information:

- “[Splitter Guide](#)” on [page 7-2](#) provides usage information
- “[Splitter Command-Line Reference](#)” on [page 7-4](#) provides reference information on splitter commands

For information about the DSP program generation process, refer to “[Software Development Flow](#)” on [page 1-2](#).

Splitter Guide

The splitter (`elfspl21k.exe`) processes executable files, producing one or more non-bootable PROM image files. Splitter operation relies on splitter options, which control the processing of the executable files into output files.

There are two ways to run the splitter:

- Command-line invocation. Refer to [“Splitter Command-Line Reference” on page 7-4](#).
- VisualDSP++ environment. Generate non-bootable PROM image files by selecting **DSP splitter file** as the project’s output. Specify splitter options from the **Split** page of the **Project Options** dialog box. Refer to online Help for details.

Split Page

When developing a DSP project, you can modify splitter default option settings directly from within the VisualDSP++ environment.

Splitter options are available from the **Split** page (also called the Split property page) of the **Project Options** dialog box. Settings correspond to `elfspl21k` command-line switches. Refer to [Figure 7-3](#).

For more information, refer to VisualDSP++ online Help.

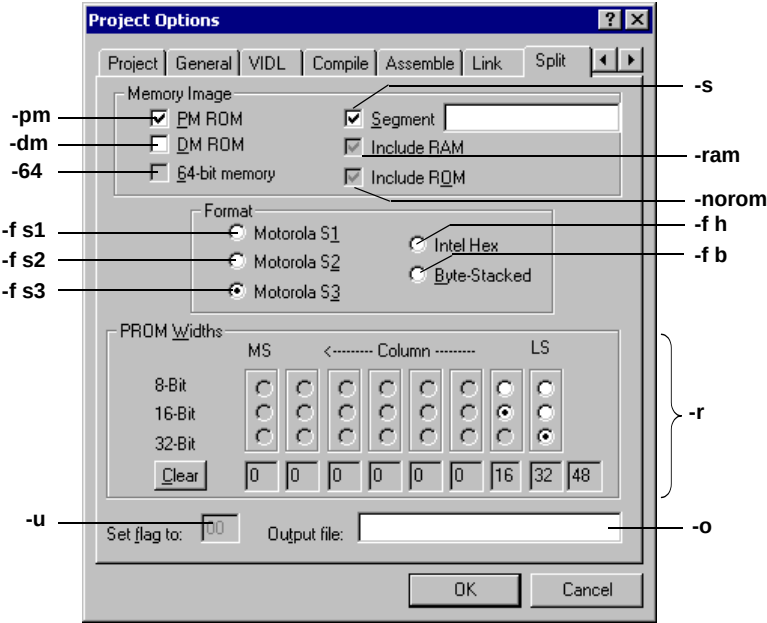


Table 7-1. Split Page

Splitter Command-Line Reference

The splitter (`elfspl21k.exe`) generates non-bootable PROM image files for SHARC DSPs by processing executable (`.DXX`) files. Splitter output is a set of PROM files with `.S_#`, `.H_#`, or `.STK` file extensions.

 When using the splitter from the VisualDSP++ environment, **Split** page settings in the **Project Options** dialog box correspond to splitter command-line switches. For more information, refer to [“Split Page” on page 7-2](#) and VisualDSP++ online Help.

elfspl21k Command-Line Syntax

The splitter command line requires the following syntax:

```
elfspl21k [-switch ...] -pm &| -dm &| -64 &| -s segment executable
```

where:

- `[-switch ...]` – Specifies the name of one or more optional switches to be processed. These switches select the operations and modes for the splitter.
- `-pm &| -dm &| -64 &| -s segment` – The “&|” symbol between these three switch indicates AND/OR. The splitter command line must include one or more of the `-pm`, `-dm`, `-64`, and `-s` switches. The `-s` switch must accompany the `-pm`, `-dm`, or `-64` switch.
- *executable* – Specifies the name of the executable file (`.DXX`) to be processed into a boot-loadable file for a single-processor system.

Items shown between bracket characters [] are optional. Items in *italic font* are user-definable and are described with each switch.

[Table 7-2](#) lists file types, file names, and extensions that the splitter expects on the `elfspl21k` command line.

Splitter command-line items and their descriptions appears in [Table 7-3 on page 7-8](#).

Command-Lines Rules

Follow these rules when using command-line invocation:

- Most items in the splitter's command line are not case sensitive; for example, `-pm` and `-PM` are interchangeable. However, the names of memory segment names must be identical, including case, to the names used in the executable.
- Switches may be used in any order.
- The name of the executable must appear at the end of the command. The name can include the drive, directory, file name, and file extension. Enclose long file names within straight-quotes; for example, "long file name".

Example Splitter Command Lines

```
elfspl21k -pm -o pm_stuff my_proj.dxe
elfspl21k -dm -o dm_stuff my_proj.dxe
elfspl21k -64 -o dm_stuff my_proj.dxe
```

In the above examples, each command line runs the splitter. The first command produces a PROM file for program memory. The second command produces a PROM file for data memory. The third command produces a PROM files for DATA64 memory.

Splitter Command-Line Reference

The switches on these command lines are as follows.

-pm
-dm
-64

Selects program memory (-pm), data memory (-dm), or DATA64 memory (-64) as sources in the executable for extraction and placement into the image. DATA64 memory does not apply to ADSP-2106x DSPs.

Because these are the only switches used to identify the memory source, the specified sources are PM, DM, or DATA64 memory segments. Because no other content switches appear on these command lines, the output file's format defaults to a Motorola 32-bit format, and the output's PROM word width defaults to 8 bits for all PROMs.

-o pm_stuff
-o dm_stuff

Specifies names for the output files. Use different names so the output of a run does not overwrite the output of a previous run. The output names are pm_stuff.s_# and dm_stuff.s_#.

my_proj.dxe

Specifies the name of the input file (.DXE) to be processed into non-bootable PROM-image files.

File Searches

Some switches take a file name as an optional parameter. [“Output File Extensions”](#) lists the expected file types, names, and extensions.

File searches are important in the splitter's process. The splitter 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 splitter searches only in that location for the file.
- *Default directory* – If you do not include path information in the file name, the splitter 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.

Output File Extensions

The splitter follows the conventions shown in [Table 7-2](#) for output file extensions.

Table 7-2. Output File Extensions

Extension	File Description
.S_#	Motorola S-record format file. The # indicates the position (0 = least significant, 1 = next-to-least significant, and so on). For info about Motorola S-record file format, refer to “Splitter Output Files in Motorola S-Record Format (.S_#)” on page A-11.
.H_#	Intel hex-32 format file. The # indicates the position (0 = least significant, 1 = next-to-least significant, and so on). For information about Intel hex-32 file format, refer to “Splitter Output Files in Intel Hex-32 Format (.H_#)” on page A-13.
.STK	Byte-stacked format file. These files are intended for host transfer of data, not for PROMs. For more information about byte-stacked file format, format files, , refer to “Splitter Output Files in Byte-Stacked Format (.STK)” on page A-13.

Command-Line Items

This section provides reference information about valid items used in the elfspl21k command line. A list of all command-line items and their descriptions appears in [Table 7-3](#).

Table 7-3. elfspl21k Utility Command-Line Items

Item	Description
<i>executable</i>	Specifies the executable file (.DxE) to be processed into PROM files. The file name must appear at the end of the command line.
-64	The -64 (include DATA64 memory) switch directs the splitter to extract all segments declared as 64-bit memory segments from the .DxE file. This switch influences the operation of the -ram and -norom switches, adding 64-bit data memory as their target.
-dm	The -dm (include data memory) switch directs the splitter to extract memory segments declared as data memory ROM from the .DxE file. The -dm switch influences the operation of the -ram and -norom switches, adding data memory as their target.
-f h -f s1 -f s2 -f s3 -f b	The -f (PROM file format) switch directs the splitter to generate a non-bootable PROM image file in the specified <i>format</i> . Available selections for <i>format</i> include: h = Intel hex-32 format s1 = Motorola EXORciser format s2 = Motorola EXORMAX format s3 = Motorola 32-bit format b = byte-stacked format If the -f switch does not appear on the command line, the default format for the PROM file is Motorola 32-bit (s3). For information on file formats, see “Build Files” on page A-6 .
-o <i>imagefile</i>	The -o (output file) switch directs the splitter to use <i>imagefile</i> as the name of the splitter’s output file(s). If not specified, the default name for the splitter’s output file is “ <i>executable.ext</i> ”, where <i>ext</i> depends on the output format.

Table 7-3. elfspl21k Utility Command-Line Items

Item	Description						
-norom	The -norom (no ROM in PROM) switch directs the splitter to ignore ROM memory segments in <i>executable</i> when extracting information for the output image. The -dm and -pm switches select data memory or program memory. The operation of the -s switch is not influenced by the -norom switch.						
-pm	The -pm (include program memory) switch directs the splitter to extract memory segments declared program memory ROM from the executable. The -pm switch influences the operation of the -ram and -norom switches, adding program memory as the target.						
-r # [# ...]	The -r (PROM widths) switch specifies the number of PROM files and their width (in bits). The splitter can create PROM files for 8-, 16-, and 32-bit wide PROMs. The default width is 8-bits. Each # parameter specifies the width of one PROM file. Place # parameters in order from most significant to least significant. The sum of the # parameters must equal the bit width of the destination memory (40 bits for DM, 48 bits for PM, or 64 bits for 64-bit memory). Example: <code>elfspl21k -dm -r 16 16 8 myfile.dxe</code> This command extracts data memory ROM from <code>myfile.dxe</code> and creates the following three output PROM files. <table> <tr> <td><code>myfile.s_0</code></td> <td>8-bits wide, contains bits 7-0</td> </tr> <tr> <td><code>myfile.s_1</code></td> <td>16-bits wide, contains bits 23-8</td> </tr> <tr> <td><code>myfile.s_2</code></td> <td>16-bits wide, contains bits 39-24</td> </tr> </table> The width of the three output files is 40 bits.	<code>myfile.s_0</code>	8-bits wide, contains bits 7-0	<code>myfile.s_1</code>	16-bits wide, contains bits 23-8	<code>myfile.s_2</code>	16-bits wide, contains bits 39-24
<code>myfile.s_0</code>	8-bits wide, contains bits 7-0						
<code>myfile.s_1</code>	16-bits wide, contains bits 23-8						
<code>myfile.s_2</code>	16-bits wide, contains bits 39-24						
-ram	The -ram (include RAM in PROM) switch directs the splitter to extract RAM segments from <i>executable</i> . The -dm, -pm, and -64 switches select the memory. The -s switch is not influenced by the -ram switch.						

Splitter Command-Line Reference

Table 7-3. elfspl21k Utility Command-Line Items

Item	Description
<code>-s segmentname</code>	The <code>-s</code> (include memory segment) switch directs the splitter to extract the contents of the specified memory segment (<i>segmentname</i>). Use the <code>-s segmentname</code> switch as many times as needed. Each instance of the <code>-s</code> switch can specify only one <i>segmentname</i> . To use this switch, the command line must also include one of the following switches (<code>-pm</code> , <code>-dm</code> , or <code>-64</code>).
<code>-u number</code>	(Byte-stacked format files only) The <code>-u</code> (user flags) switch, which may be used only in combination with the <code>-f b</code> switch, directs the splitter to use the <i>number</i> in the user-flags field of a byte-stacked format file. If the <code>-u</code> switch is not used, the default value for <i>number</i> is 0. By default, <i>number</i> is decimal. If <i>number</i> is prefixed with “0x”, the splitter interprets the <i>number</i> as hexadecimal. For more information, see “Splitter Output Files in Byte-Stacked Format (.STK)” on page A-13 .

A FILE FORMATS

The VisualDSP++ development tools support many file formats, in some cases several for each development tool. This appendix describes file formats that are prepared as input for the tools and points out the features of files produced by the tools.

This appendix describes three types of file formats:

- [“Source Files” on page A-2](#)
- [“Build Files” on page A-6](#)
- [“Debugger Files” on page A-15](#)

Most of the development tools use industry-standard file formats. Sources that describe these formats appear in [“Format References” on page A-16](#).

Source Files

This section describes these input file formats:

- “C/C++ Source Files” on page A-2
- “Assembly Source Files (.ASM)” on page A-3
- “Assembly Initialization Data Files (.DAT)” on page A-3
- “Header Files (.H)” on page A-5
- “Linker Description Files (.LDF)” on page A-5
- “Linker Command-Line Files (.TXT)” on page A-6

C/C++ Source Files

These are text files (with extensions such as .C, .CPP, .CXX, and so on) containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and (typically) preprocessor commands.

Several “dialects” of C code are supported: pure (portable) ANSI C, and at least two subtypes^{*} 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 and Library Manual for SHARC DSPs*.

^{*} With and without built-in function support; a minimal differentiator. There are others.

Assembly Source Files (.ASM)

Assembly source files are text files containing assembly instructions, assembler directives, and (optionally) preprocessor commands. For information on assembly instructions, see your DSP's *Programming Reference*.

The DSP's instruction set is supplemented with assembler directives. Preprocessor commands control macro processing and conditional assembly or compilation.

For information on the assembler and preprocessor, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs*.

Assembly Initialization Data Files (.DAT)

Assembly initialization data (.DAT) files are text files that contain fixed- or floating-point data. These files provide the initialization data for an assembler `.VAR` directive or serve in other tool operations.

When a `.VAR` directive uses a `.DAT` file for data initialization, the assembler reads the data file and initializes the buffer in the output object file (.DOJ). Data files have one data value per line and may have any number of lines.

The `.DAT` extension is explanatory or mnemonic. A directive to `#include <file>` can take any file name (or extension) as an argument.

Floating-point values in data files consist of a decimal mantissa value, a decimal point, and a decimal integer exponent. Separate the exponent from the mantissa with an uppercase or lowercase "e". The mantissa and exponent can be signed. The following examples show valid floating-point values.

1.2345e12

567.8e-3

-7.1234

Source Files

The assembler converts decimal representations to floating-point format (standard IEEE 32-bit single-precision or 40-bit extended-precision). The source code file containing a `.VAR` initialization specifies the format (32-bit or 40-bit) for floating-point numbers using the `.PRECISION` directive. If a floating-point number in an initialization file does not fit the specified format, the assembler rounds the number.

Rounding mode is also specified in the source code file with a `.ROUND` directive. For more information, refer to the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs*.

The assembler supports binary, decimal, and hexadecimal formats (bases) within expressions and assembly instructions. [Table A-1](#) describes the notation conventions used by the assembler to distinguish between numeric formats.

Table A-1. Numeric Formats

Convention	Description
<code>0xnumber</code> <code>H#number</code> <code>h#number</code>	Hexadecimal number.
<code>number</code> <code>D#number</code> <code>d#number</code>	Decimal number.
<code>B#number</code> <code>b#number</code>	Binary number.
<code>O#number</code> <code>o#number</code>	Octal number.

Rules

For all numeric bases, the assembler uses 32-, 40-, or 48-bit words for data storage. 48-bit data is for PM only. The largest word in the buffer determines the size for all words in the buffer. If you have some 32-bit data in a 40-bit wide buffer, the assembler loads the 32-bit value into the most sig-

nificant 32 bits of the 40-bit memory location and zero-fills the lower eight bits. Similarly, the assembler loads 32- or 40-bit data into the most significant bits in a 48-bit wide buffer. [Table A-2](#) shows examples of data storage.

Table A-2. Data Storage Examples

Decimal Data in .DAT File	Hexadecimal Data in .DAT File or 32-bit DM	Hexadecimal Data in .DAT File or 40-bit DM	Hexadecimal Data in .DAT File or 48-bit DM
7	0x00000007	0x0000000700	0x000000070000
255	0x000000FF	0x000000FF00	0x000000FF0000
32768	0x00008000	0x0000800000	0x000080000000
10855845	0x00A5A5A5	0x00A5A5A500	0x00A5A5A50000
305419896	0x12345678	0x1234567800	0x123456780000
Not expressible	Not expressible	0x1122334455	0x112233445500
Not expressible	Not expressible	0x0012345678	0x001234567800
Not expressible	Not expressible	Not expressible	0x000012345678

Header Files (.H)

Header files are ASCII text files that contain macros or other preprocessor commands that the preprocessor substitutes into source files. For information on macros or other preprocessor commands, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs*.

Linker Description Files (.LDF)

.LDF files are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language, see [“Linker Command-Line Reference” on page 1-28](#).

Linker Command-Line Files (.TXT)

Linker command-line files (.TXT) are ASCII text files that contain command-line input for the linker. For more information on the linker command line, see [“Linker Command-Line Reference”](#) on page 1-28.

Build Files

Build files are produced by the VisualDSP++ development tools when building a project. This section describes these build file formats:

- [“Assembler Object Files \(.DOJ\)”](#) on page A-7
- [“Library Files \(.DLB\)”](#) on page A-7
- [“Linker Output Files \(.DXE, .SM, and .OVL\)”](#) on page A-7
- [“Memory Map Files \(.MAP\)”](#) on page A-8
- [“Loader Output Files in Intel Hex-32 Format \(.LDR\)”](#) on page A-8
- [“Loader Output Files in ASCII Format \(.LDR\)”](#) on page A-10
- [“Loader Output Files in Include Format \(.LDR\)”](#) on page A-10
- [“Loader Output Files in Binary Format \(.LDR\)”](#) on page A-11
- [“Splitter Output Files in Motorola S-Record Format \(.S_#\)”](#) on page A-11
- [“Splitter Output Files in Intel Hex-32 Format \(.H_#\)”](#) on page A-13
- [“Splitter Output Files in Byte-Stacked Format \(.STK\)”](#) on page A-13

Assembler Object Files (.DOJ)

Assembler output object files (.DOJ) are binary, executable and linkable file (ELF) format. Object files contain relocatable code and debugging information for a DSP program's memory segments. The linker processes object files into an executable file (.DXE). For information on the object file's ELF format, see the [“Format References” on page A-16](#).

Library Files (.DLB)

Library files, the archiver's output, are in binary, executable and linkable file (ELF) format. Library files (called archive files in previous software releases) contain one or more object files (archive elements).

The linker searches through library files for library members used by the code. For information on the ELF format used for executable files, refer to [“Format References” on page A-16](#).

Linker Output Files (.DXE, .SM, and .OVL)

The linker's output files are binary, executable and linkable file (ELF) format. These executable files contain program code and debugging information. The linker may fully or partially resolve addresses in executable files, depending on the options given on the linker's command line and in the .LDF file. For information on the ELF format used for executable files, see the TIS Committee texts cited in [“Format References” on page A-16](#).



The archiver automatically converts legacy input objects from COFF to ELF format.

Memory Map Files (.MAP)

The linker can output memory map files (.MAP), which are ASCII text files that contain memory and symbol information for your executable file(s). The map contains a summary of memory defined with `MEMORY{ }` commands in the .LDF file, and provides a list of the absolute addresses of all symbols.

Loader Output Files in Intel Hex-32 Format (.LDR)

The loader can output Intel hex-32 format (.LDR) files. These ASCII files support 8-bit-wide PROMs. The files are used with an industry-standard PROM programmer to program memory devices for a hardware system. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel hex-32 format appears in the loader's output file. Each line in the Intel hex-32 file contains an extended linear address record, a data record, or the end-of-file record.

:020000040010E9	Extended linear address record
:0A0004003C40343434261422260850	Data record
:00000401FB	End-of-file record

Extended linear address records are used because data records have a 4-character (16-bit) address field, but SHARC processors require up to 32 bits to address data memory (using DAG1 and DAG2) and 24 bits to execute code from external memory (program sequencer). Extended linear address records specify bits 16-31 for the data records that follow.

[Table A-3](#) shows the organization of an example data record.

[Table A-4](#) shows an example of an extended linear address record.

[Table A-5](#) shows an end-of-file record.

Table A-3. Example – Data Record

Field	Purpose
:0A0004003C40343434261422260850	Example record
:	Start character
0A	Byte count of this record
0004	Address of first byte
00	Record type
3C	First data byte
08	Last data byte
50	Checksum

Table A-4. Example – Extended Linear Address Record

Field	Purpose
:02000000340850	Example record
:	Start character
02	Byte count (always 02)
0000	Address (always 0000)
00	Record type
3408	Offset address
50	Checksum

Table A-5. End-of-File Record

Field	Purpose
:00000001FF	End-of-file record
:	Start character
00	Byte count (zero for this record)
0000	Address of first byte

Table A-5. End-of-File Record (Cont'd)

Field	Purpose
01	Record type
FF	Checksum

Loader Output Files in ASCII Format (.LDR)

The loader can output ASCII-format files (.LDR), which are similar to assembler initialization data files (.DAT). ASCII-format files are used in the same manner as initialization data files. For information on .DAT files, see [“Assembly Initialization Data Files \(.DAT\)” on page A-3](#).

The data order is one 16-bit hexadecimal value per line, providing the lower-, middle-, and upper 16 bits of each 48-bit instruction.

Loader Output Files in Include Format (.LDR)

The loader can output include-format files (.LDR). These files permit the inclusion of the loader file in a C program.

Files in include format are ASCII text files that consist of 48-bit instructions, one per line. Each instruction is presented as three 16-bit hexadecimal numbers. For each 48-bit instruction, the data order is lower-, middle, and then upper-16 bits. The following are example lines from an include-format file.

```
0x005c, 0x0620, 0x0620,  
0x0045, 0x1103, 0x1103,  
0x00c2, 0x06be, 0x06be
```

The following example shows how to include this file in a C program.

```
const unsigned loader_file[] =  
{  
#include "foo.ldr"
```



```
};
const unsigned loader_file_count = sizeof loader_file
                                / sizeof loader_file;
```

`loader_file_count` reflects the actual number of elements in the array and cannot be used to process the data.

Loader Output Files in Binary Format (.LDR)

The loader can output binary-format files (.LDR) to support a variety of PROM and microcontroller storage applications.

Binary-format files use less space than the other loader file formats. Binary files contains 48-bit instructions in big-endian format (most significant bit first).

Splitter Output Files in Motorola S-Record Format (.S_#)

The splitter can output Motorola S-record format files, which conform to the Intel standard. The three file formats supported by the PROM splitter differ only in the width of the address field: S1 (16 bits), S2 (24 bits), or S3 (32 bits).

An S-record file begins with a header record and ends with a termination record. Between these two records are data records, one per line.

Example

S00600004844521B	Header record
S10D00043C4034343426142226084C	Data record (S1)
S903000DEF	Termination record (S1)

[Table A-6](#) shows the organization of an example header record.

Table A-6. Example – Header Record

Field	Purpose
S00600004844521B	Example record
S0	Start character
06	Byte count of this record
0000	Address of first data byte
484452	Identifies records that follow
1B	Checksum

[Table A-7](#) shows the organization of an S1 data record.

Table A-7. Example – S1 Data Record

Field	Purpose
S10D00043C4034343426142226084C	Address of first byte
S1	Record type
0D	First data byte
0004	Last data byte
3C	Checksum
08	Last data byte
4C	Checksum

The S2 data record has the same format, except that the start character is S2 and the address field is six characters wide. The S3 data record is the same as the S1 data record except that the start character is S3 and the address field is eight characters wide.

Termination records have an address field that is 16-, 24-, or 32-bits wide, whichever matches the format of the preceding records. [Table A-8](#) shows the organization of an S1 termination record.

Table A-8. Example – S1 Termination Record

Field	Purpose
S903000DEF	Example record
S9	Start character
03	Byte count of this record
000D	Address
EF	Checksum

The S2 termination record has the same format, except that the start character is S8 and the address field is six characters wide.

The S3 termination record is the same as the S1 format, except the start character is S7 and the address field is eight characters wide.

Splitter Output Files in Intel Hex-32 Format (.H_#)

The splitter can output Intel hex-32 format (.H_#) files. These ASCII files support a variety of PROM devices. For an example of how the Intel hex-32 format appears for an 8-bit-wide PROM, see [“Source Files” on page A-2](#).

The splitter prepares a set of PROM files. Each PROM holds a portion of each instruction or data. This configuration differs from the loader’s output.

Splitter Output Files in Byte-Stacked Format (.STK)

The splitter can output files in byte-stacked (.STK) format. These files are not intended for PROMs, but are ideal for microcontroller data transfers.

A file in byte-stacked format comprises a series of one-line headers, each followed by a block (one or more lines) of data. The last line in the file is a header that signals the end of the file.

Build Files

Lines consist of ASCII text that represents hexadecimal digits. Two characters represent one byte. For example, F3 represents a byte whose decimal value is 243.

Table A-9 shows an example of a header record in byte-stacked format.

Table A-9. Example – Header Record in Byte-Stacked Format

Field	Purpose
200080000000000080000001E	Example record
20	Width of address and length fields (in bits)
00	Reserved
80	PROM splitter flags (90 = PM, 00 = DM)
00	User-defined flags (loaded with -u switch)
00000008	Start address of data block
0000001E	Number of bytes that follow

In the above example, the start address and block length fields are 32 bits wide. The file contains program memory data (the MSB is the only flag currently used in the PROM splitter flags field). No user flags are set. The address of the first location in the block is 0x08. The block contains 30 (1E) bytes (5 program memory code words). The number of bytes that follow (until next header record or termination record) must be nonzero.

A block of data records follows its header record, five bytes per line for data memory, and six byte per line for program memory. For example:

Program Memory Segment (Code or Data)

```
3C4034343426
142226083C15
```

Data Memory Segment

```
3C40343434
2614222608
```

The bytes are ordered left to right, most significant to least.

The termination record has the same format as the header record, except for the right-most field (number of records), which is all zeros.

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 assembler's data file format (.DAT) and the loader's loadable file formats (.LDR).

The standard hexadecimal format for a SPORT data file is one integer value per line. Hexadecimal numbers do not require a 0x prefix. A value can have any number of digits, but is read into the SPORT register as follows:

- The hexadecimal number is converted to binary.
- The number of binary bits read in matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zero values shorter than the word size or conversely truncates bits beyond the word size on the MSB end.

Example

In this example, a SPORT register is set for 20-bit words and the data file contains hexadecimal numbers. The simulator converts the hex numbers to binary and then fills/truncates to match the SPORT word size. In [Table A-10](#), the A5A5 is filled and 123456 is truncated.

Format References

Table A-10. 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
11111	0001 0001 0001 0001 0001	0001 0001 0001 0001 0001
123456	0001 0010 0011 0100 0101 0110	0010 0011 0100 0101 0110

Format References

The following texts define industry-standard file formats supported by VisualDSP++.

- Gircys, G.R. (1988) *Understanding and Using COFF* by O'Reilly & Associates, Newton, MA
- (1993) *Executable and Linkable Format (ELF) V1.1* from the Portable Formats Specification V1.1, Tools Interface Standards (TIS) Committee.

Go to: <http://developer.intel.com/vtune/tis.htm>.

- (1993) *Debugging Information Format (DWARF) V1.1* from the Portable Formats Specification V1.1, UNIX International, Inc.

Go to: <http://developer.intel.com/vtune/tis.htm>.

B UTILITIES

The VisualDSP++ development software includes several utilities, some of which run from a command line only. Some utilities support legacy code, and others are intended for users who prefer to use the command-line version of the tools instead of accessing them from within the VisualDSP++ environment.

This appendix describes the ELF file dumper utility and mem21k memory initializer utility.

This appendix describes the following utilities:

- “[elfdump – ELF File Dumper](#)”
- “[mem21k – Memory Initializer](#)” on page B-6

elfdump – ELF File Dumper

The ELF file dumper (`elfdump.exe`) utility extracts data from ELF-format executable files (`.DXXE`) yielding a text showing the ELF file’s contents.

`elfdump` is often used with the archiver (`elfar.exe`). Refer to “[Disassembling a Library Member](#)” on page B-4 for details.

Syntax: `elfdump [switches] [objectfile]`

[Table B-1](#) shows switches used with the `elfdump` command.

Table B-1. ELF File Dumper Command-Line Switches

Switch	Description
-c	Stabs to mdebug conversion
-fh	Prints the file header
-arsym	Prints the library symbol table
-arall	Prints every library member
-help	Prints the list of elfdump switches to stdout
-ph	Prints the program header table
-sh	Prints the section header table. This is the default when no options are specified
-notes	Prints note segment(s)
-n <i>name</i>	Prints contents of the named section(s). The name may be a simple 'glob'-style pattern, using "?" and "*" as wildcard characters. Each section's name and type determines its output format, unless overridden by a modifier (see below).
-i x0[-x1]	Prints contents of sections numbered x0 through x1, where x0 and x1 are decimal integers, and x1 defaults to x0 if omitted. Formatting rules as are for the -n switch.
-all	Prints everything. Same as -fh -ph -sh -notes -n '*'.
-ost	Omits string table sections

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

Switch	Description
-v	Prints version information
<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. - hN Dumps contents in hexlet, N bytes per group. - HN Dumps contents in hexlet (MSB first order), N bytes per group. - i Prints contents as list of disassembled machine instructions.

Disassembling a Library Member

The `elfar` and `elfdump` utilities are more effective when their capabilities are combined. One application of these utilities is for disassembling a library member and converting it to source code. Use this technique when the source of a particularly useful routine is missing and is available only as a library routine.

For information about `elfar`, refer to [“Archiver” on page 4-1](#).

The following procedure lists the objects in a library, extracts an object, and converts the object to a listing file. The first archiver command line lists the objects in the library and writes the output to a text file.

```
elfar -p libc.dlb > libc.txt
```



This command assumes the current directory is:

C:\Program Files\Analog Devices\VisualDSP\21xxx\lib

Open the text file, scroll through it, and locate the object file you need. Then, use the following archiver command to extract the object from the library.

```
elfar -e libc.dlb fir.doj
```

To convert the object file to an assembly listing file with labels, use the following `elfdump` command line.

```
elfdump -ns * fir.doj > fir.asm
```

The output file is practically source code. Just remove the line numbers and opcodes.

Disassembly yields a listing file with symbols. Assembly source with symbols can be useful if you are familiar with the code and hopefully have some documentation on what the code does. If the symbols were stripped during linking, the dumped file contains no symbols.



Disassembling a third-party's library may violate the license for the third-party software. Ensure there are no copyright or license issues with the code's owner before using this disassembly technique.

Dumping Overlay Library Files

Use the `elfar` and `elfdump` utilities to extract and view the contents of overlay library files (`.OVL`).

For example, the following command lists (prints) the contents (library members) of the `CLONE2.OVL` library file.

```
elfar -p CLONE2.OVL
```

The following command allows you to view one of the library members (`CLONE2.ELF`).

```
elfdump -all CLONE2.OVL(CLONE2.elf)
```

The following commands extract `CLONE2.ELF` and print its contents.

```
elfar -e CLONE2.ovl  
elfdump -all CLONE2.elf
```



Switches for these commands are case sensitive.

mem21k – Memory_INITIALIZER

The memory initializer (`mem21k.exe`) operates on the executable file (`.DXE`) produced by the linker. It does not operate on associated `.OVL` files or `.SM` files. When run by the compiler driver from the command line (or when selected via the **Mem21K** button on the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment), the linker creates an executable file that becomes the input to the memory initializer. To use memory initialization, use the compiler's `-mem` switch.

The memory initializer transfers all RAM memory initializations to the `seg_init` PM ROM memory segment. This has two effects. First, all RAM is initialized to its proper value before the call to `main()`. This is true for embedded code programmed into ROM and for programs downloaded into RAM in a SHARC DSP system.

Besides memory initialization, the utility reduces an executable's overall size by combining contiguous, identical initializations into a single block. A large array of identically initialized data (for example, zeros) are compressed to a single element in the executable after being processed by the initializer. When the `PACKING()` command is used in the `.LDF` file, `mem21k` processes packed words in a `.DXE` file the same as unpacked words.

The C run-time header reads the `seg_init` memory segment generated by the initializer to determine which memory locations should be initialized to what values. This process occurs during the `__lib_setup_processor` routine called for the run-time header.

The memory initializer does not attempt to compress three segments (even when they are defined in the `.LDF` file as RAM): the initialization segment (`seg_init`), the code segment (`seg_pmco`), and the run-time header segment (`seg_rth`). These memory segments contain initialization routines and data, so they cannot be compressed.

Long Words (64 Bits)

For long words, mem21k breaks each 64-bit word into two 31-bit words and then extends each 32-bit word to a 40- or 48-bit word, depending on whether they are declared as DM or PM. Then mem21k places them into seg_init. Instead of initializing a single 64-bit word, the C run-time header initializes two 32-bit words to achieve the initialization of DATA64 memory.

Short Words (16 Bits)

In a .DxE file, 16-bit words with a DM qualifier are extended to 40-bit words and 16-bit words with a PM qualifier are extended to 48-bit words.

Command Syntax

The memory initializer is invoked as follows:

```
mem21k [-h -v] -o outputfile inputfile
```

The command options have the following meanings:

- -h – Displays usage
- -v – Verbose
- -o *outputfile* – Specifies the output file name
- *inputfile* – Specifies the input file name (.DxE)

C LINKER LEGACY SUPPORT

The linker and utilities provide several types of support for code developed with previous development software tools. For more information on legacy support, see [“Utilities” on page B-1](#).

This release of the linker has been updated to provide support for VisualDSP++ 3.0. This includes the following changes:

- The addition of the `-M` and `-MM` linker control switches (documented in [“Linker Command-Line Switches” on page 1-33](#)) to control whether dependencies are built and generated to `stdout`
- Support for the ADSP-21161 processor
- Other underlying changes to improve error-checking and error-reporting

I INDEX

Symbols

- `$COMMAND_LINE_LINK_AGAI`
 - NST LDF macro [2-19](#)
- `$COMMAND_LINE_OBJECTS`
 - LDF macro [2-18](#)
- `$COMMAND_LINE_OUTPUT_DIRECTORY`
 - LDF macro [2-19](#)
- `$COMMAND_LINE_OUTPUT_FILE`
 - LE LDF macro [2-7](#), [2-19](#)
- `$macro` LDF macro [2-19](#)
- `$OBJECTS` LDF macro [2-5](#)
- `.ASM` files
 - assembler [A-3](#)
- `.DAT` files
 - initialization data [A-3](#)
 - numeric formats [A-4](#)
- `.DLB` files [1-30](#), [A-7](#)
 - description of [A-7](#)
 - symbol name encryption [4-11](#)
- `.DOJ` files [1-30](#)
 - about [A-7](#)
- `.DXE` files [1-6](#), [1-30](#), [A-7](#)
 - data extraction [B-1](#)
 - linker [A-7](#)
- `.H` files [A-5](#)
- `.H_` files [A-13](#)
- `.LDF` files [1-30](#), [A-5](#)
 - commands in [1-12](#), [2-20](#)
 - comments in [2-8](#)
 - creating in Expert Linker [3-4](#)
 - memory segments [1-13](#)
 - operators [2-13](#)
 - output sections [1-13](#)
 - purpose [1-14](#)
- `.LDR` files [A-11](#)
 - ASCII-format [A-10](#)
 - hex-format [A-8](#)
 - include-format [A-10](#)
 - specifying host bus width [6-32](#)
- `.MAP` files [A-8](#)
 - linker [A-8](#)
- `.OVL` files [1-6](#), [1-30](#), [2-57](#), [A-7](#)
 - dumping [B-5](#)
 - extracting content from [B-5](#)
 - linker [A-7](#)
 - viewing contents [B-5](#)
- `.S_` files [A-11](#)
- `.SECTION` directive [1-4](#)
- `.SM` files [1-6](#), [1-30](#), [A-7](#)
 - linker [A-7](#)
 - omitting [5-30](#)
- `.STK` files [A-13](#)
- `.TXT` files [A-6](#)
 - linker [A-6](#)

INDEX

@filename linker command-line switch
1-36
_ov_endaddress_# 1-45, 1-63
_ov_runtimestartaddress_# 1-46, 1-63
_ov_size_# 1-45, 1-63
_ov_startaddress_ 1-63
_ov_startaddress_# 1-45
_ov_word_size_live_# 1-46, 1-63
_ov_word_size_run_# 1-46, 1-63

A

ABSOLUTE() LDF operator 2-13

adding

- input sections 3-10
- LDF macros 3-10
- library files 3-10
- object files 3-10

ADDR() LDF operator 2-14

ADDR23-0 lines 6-16

ADSP-21065L control registers

- DMAC8 5-20

ADSP-2106x control registers

- DMAC0 5-20
- DMAC6 5-23

ADSP-2106x/21160 control registers

- DMAC6 5-20
- DMAC8 5-20

ADSP-21160 control registers

- DMAC10 5-20
- DMAC8 5-23

ADSP-21161 control registers

- DMAC10 6-14, 6-18
- DMAC8 6-21, 6-23

ADSP-21161N loader

header words 6-9

-HostWidth # switch 6-32

include output file 6-11

include-format files 6-11

ADSP-21xxx DSPs

- overlays 2-72

ALGORITHM() LDF command 2-57

ALIGN() LDF command 2-21

alignment

- specifying properties 3-58

ALL_FIT LDF identifier 2-57, 3-61

ARCHITECTURE() LDF command
2-22

archive files

- see library files A-7

archive members A-7

archive routines

- creating entry points 4-4

archiver

- about 4-1

- command-line reference 4-7

- constraints 4-9

- file searches 4-6

- running 4-7

- symbol name encryption 4-11

- use in disassembly B-4

assembler

- initialization data files (.DAT) A-3

- numeric formats A-4

- object files (.DOJ) A-7

- source files (.ASM) A-3

B

BEST_FIT LDF identifier 2-57

- binary-format files [A-11](#)
 - boot loading process [5-8](#)
 - boot memory formats [6-9](#)
 - boot mode pins [5-5](#), [6-4](#)
 - BMS [5-6](#), [6-5](#)
 - EBOOT [5-6](#), [6-5](#)
 - LBOOT [5-6](#), [6-5](#)
 - boot modes [5-2](#), [5-3](#), [6-2](#), [6-3](#)
 - boot sequence [5-5](#), [6-4](#)
 - data packing [5-18](#)
 - EPROM [5-2](#), [5-3](#), [5-15](#), [6-2](#), [6-3](#), [6-12](#), [6-14](#)
 - host [5-2](#), [5-3](#), [5-19](#), [6-2](#), [6-3](#)
 - host (ADSP-21161) [6-17](#)
 - IIVT bit [5-14](#), [6-10](#)
 - kernel loading [5-5](#), [6-4](#)
 - link [5-2](#), [5-3](#), [5-23](#), [6-2](#), [6-3](#)
 - link (ADSP-21161) [6-20](#)
 - no boot [5-3](#), [5-24](#), [6-3](#), [6-10](#), [6-24](#)
 - reset vector address [5-5](#), [6-4](#)
 - SPI [6-22](#)
 - boot types
 - working with [5-2](#), [6-2](#)
 - booting
 - about [5-3](#)
 - DATA23-16 bits [6-13](#), [6-17](#)
 - EPROM mode selection [5-15](#), [6-13](#)
 - host mode selection [5-19](#), [6-17](#)
 - include-format files [6-11](#)
 - interrupt vector table [5-14](#), [6-10](#)
 - link mode selection [5-23](#), [6-20](#), [6-21](#)
 - link port [5-23](#)
 - multiprocessor system [5-24](#)
 - NOP or IDLE instruction [6-13](#), [6-18](#)
 - SPI mode selection [6-23](#)
 - wake-up sequence [6-16](#), [6-20](#)
 - boot-kernels
 - changes and software issues [5-10](#), [6-6](#)
 - default [6-6](#)
 - features & boot-loading process [5-13](#), [6-5](#)
 - modifying [5-11](#), [6-6](#)
 - rebuilding [5-11](#), [6-6](#)
 - branch instruction [2-46](#)
 - broadcast space [2-71](#)
 - broadcast writes [2-71](#)
 - build files
 - description of [A-6](#)
 - build options
 - loader [5-26](#), [6-27](#)
 - splitter [7-2](#)
 - built-in LDF macros [2-18](#)
 - bus lock [2-71](#)
 - multiprocessor systems [2-71](#)
 - byte-stacked files [A-13](#)
 - used with splitter [7-10](#)
- ## C
- C/C++
 - source files [A-2](#)
 - calls
 - inter-overlay [1-64](#)
 - inter-processor [1-66](#)
 - color selection
 - Expert Linker [3-13](#)
 - command line
 - ADSP-21xxx loader [5-26](#), [6-27](#)
 - loader [5-26](#), [6-27](#)

INDEX

- splitter [7-8](#)
- commands
 - LDF [2-20](#)
 - linker [1-28](#)
- comments
 - .LDF file [2-8](#)
- conventions, manual [-xxv](#)
- converting
 - library members to source code [B-4](#)
- Create LDF wizard [3-4](#)
- customer support [-xvii](#)

D

- Darchitecture (target architecture)
 - linker command-line switch [1-36](#)
- data blocks
 - header words [6-9](#)
- debugger
 - files [A-15](#)
- DEFINED() LDF operator [2-15](#)
- development tools
 - file formats [A-1](#)
- dialog boxes
 - Legend [3-13](#)
- directories
 - supported by linker [1-31](#)
- disassembling
 - library members [B-4](#)
- disassembly
 - using archiver [B-4](#)
 - using dumper [B-4](#)
- dm (include data memory) splitter
 - command-line switch [7-8](#)

- dm (include Data Memory) splitter
 - switch [7-8](#)
- DMA channel 10 (ADSP-21161) [6-13](#), [6-17](#)
- DMAC0 control register
 - host booting [5-20](#)
- DMAC10 control register [6-12](#)
 - EPROM booting [6-14](#)
 - host booting [5-20](#), [6-17](#), [6-18](#)
 - initialization [6-14](#), [6-18](#)
- DMAC6 control register
 - EPROM booting [5-16](#)
 - link port booting [5-23](#)
- DMAC8 control register
 - EPROM booting [5-16](#)
 - host booting [5-20](#)
 - link port booting [5-23](#), [6-21](#)
 - SPI port booting [6-23](#)
- DSPs
 - development software [1-2](#)
- dumper
 - use in disassembly [B-4](#)
- DWARF
 - references [A-16](#)

E

- e (eliminate unused symbols) linker
 - command-line switch [1-38](#)
- ELF file dumper
 - about [B-1](#)
 - command-line switches [B-1](#)
 - extracting data [B-1](#)
 - overlay library files [B-5](#)
 - references [A-16](#)

- elfar.exe
 - about [4-1](#)
 - command-line reference [4-7](#)
- elfdump.exe
 - about [B-1](#)
 - used by Expert Linker [3-37](#)
- elfloader.exe [5-26, 6-27](#)
- ELIMINATE() LDF command [2-22](#)
- ELIMINATE_SECTIONS() LDF
 - command [2-23](#)
- elimination
 - specifying properties [3-47](#)
- encryption
 - symbol names in libraries [4-11](#)
- END() LDF identifier [2-28](#)
- EPROM booting
 - ADDR 31-0 [5-18](#)
 - ADSP-21161N loader [6-12](#)
 - BMS pin [5-19](#)
 - boot-loading sequence [5-17](#)
 - data packing [5-17](#)
 - DMA settings [5-16, 6-12](#)
 - external port data bus lines [5-18](#)
 - mode selections [5-15, 6-12](#)
 - multiprocessor systems [5-24, 6-25](#)
 - program counter settings [5-17](#)
 - single-processor systems [5-19](#)
 - transfer settings [6-14](#)
 - WAIT register [5-19](#)
- errors
 - linker [1-19](#)
- es (eliminate listed sections) linker
 - command-line switch [1-38](#)
- ev (eliminate unused symbols,
 - verbose) linker command-line
 - switch [1-38](#)
- executable files [A-7](#)
- EXORciser format files [7-8](#)
- EXORMAX format files [7-8](#)
- Expert Linker
 - about [3-1](#)
 - color selection [3-13](#)
 - launching [3-3](#)
 - mapping sections in [3-12](#)
 - Memory Map pane [3-16](#)
 - object properties [3-42](#)
 - overview [1-18, 3-1](#)
 - resize cursor [3-26](#)
 - window [3-10](#)
- extracting data
 - ELF executable files [B-1](#)
- F**
- f (PROM file format) splitter
 - command-line switch [7-8](#)
- file extensions
 - ADSP-21161 loader [6-29, 6-31](#)
 - ADSP-21xxx loader [5-28](#)
 - splitter [7-7](#)
- files
 - .ASM [A-3](#)
 - .DAT [A-3](#)
 - .DLB [A-7](#)
 - .DOJ [A-7](#)
 - .DXE [A-7](#)
 - .H [A-5](#)
 - .H_ [A-13](#)

INDEX

- .LDR (ASCII-format) [A-10](#)
- .LDR (binary format) [A-11](#)
- .LDR (hex format) [A-8](#)
- .LDR (include-format) [A-10](#)
- .MAP [A-8](#)
- .OVL [A-7](#)
- .S_ [A-11](#)
- .SM [A-7](#)
- .STK [A-13](#)
- .TXT [A-6](#)
- assembler [A-7](#)
- build [A-6](#)
- byte-stacked [A-13](#)
- C/C++ [A-2](#)
- debugger [A-15](#)
- dumping contents of [B-1](#)
- executable [A-7](#)
- EXORciser format [7-8](#)
- EXORMAX format [7-8](#)
- format references [A-16](#)
- formats [A-1](#)
- header [A-5](#)
- include format [6-11](#), [A-10](#)
- input [A-2](#)
- Intel hex-32 format [A-13](#)
- library [A-7](#)
- linker command-line [1-30](#), [1-32](#)
- linker command-line (.TXT) [A-6](#)
- loader include-format (.LDR) [A-10](#)
- Motorola S-record format [A-11](#)
- output [1-6](#)
- FILL() LDF command [2-23](#), [2-56](#)
- FIRST_FIT LDF identifier [2-57](#)

G

- gaps
 - inserting [3-34](#)
- generating
 - PROM files [7-1](#)
- global properties
 - setting [3-43](#)
- graphical memory map
 - unavailable memory regions [3-28](#)

H

- h (command-line help) linker
 - command-line switch [1-38](#)
- header files [A-5](#)
- header words
 - data blocks [6-9](#)
- heap
 - graphic representation [3-62](#)
 - managing in memory [3-62](#)
- help (command-line help) linker
 - command-line switch [1-38](#)
- hex-format files
 - .H_ [A-13](#)
 - .LDR [A-8](#)
- host
 - setting word width [6-11](#)
- host booting [5-19](#), [6-17](#)
 - BMS pin [5-19](#), [6-17](#)
 - boot-loading sequence [5-21](#)
 - BUSLCK bit [5-22](#)
 - DMA interrupts [5-22](#)
 - IMASK register [5-21](#), [5-22](#)
 - IMDW register [5-22](#)
 - NOP or IDLE instruction [5-22](#)

- program counter settings 5-20
- transfer settings 6-18
- host bus width
 - specifying 6-32

I

- i (include search directory) linker
 - command-line switch 1-39

- icons

- Expert Linker 3-13
 - unmapped icon 3-12

- id#exe loader switch 5-25
 - processor ID 5-33, 6-34

- include format

- booting 6-11

- INCLUDE() LDF command 2-23

- include-format files

- PROM booting 6-11

- initialization type tags 6-9

- input sections

- adding 3-10

- directives 1-4

- names 1-21

- source code 1-3

- Input Sections pane 3-10

- menu selections 3-10

- INPUT_SECTION_ALIGN() LDF
 - command 2-23

- INPUT_SECTIONS() LDF identifier
 - 2-7, 2-55

- inserting

- gaps 3-34

- Intel hex-32 (.H_) files A-13

- inter-overlay calls 1-64

- inter-processor calls 1-66

- interrupt vector table 5-14, 6-10

K

- keep (keep unused symbols) linker
 - command-line switch 1-39

- KEEP() LDF command 2-25

L

- L path (libraries and objects) linker
 - command-line switch 1-37

- LDF

- see .LDF files 1-6

- LDF commands

- about 1-12, 2-20

- ALIGN() 2-21

- ARCHITECTURE() 2-22

- ELIMINATE() 2-22

- ELIMINATE_SECTIONS() 2-23

- FILL() 2-56

- INCLUDE() 2-23

- INPUT_SECTION_ALIGN() 2-23

- KEEP() 2-25

- LINK_AGAINST() 2-25

- MAP() 2-26

- MEMORY{} 2-26

- MPMEMORY{} 2-29

- OVERLAY_GROUP{} 2-31

- OVERLAY_INPUT{} 2-56

- PLIT{} 2-44, 2-56

- PROCESSOR{} 2-50

- RESOLVE() 2-52

- SEARCH_DIR() 2-52

- SECTIONS{} 2-53

INDEX

- LDF macros
 - about [2-17](#)
 - adding [3-10](#)
 - list of [2-18](#)
- LDF operators
 - about [2-16](#)
 - ABSOLUTE() [2-13](#)
 - ADDR() [2-14](#)
 - DEFINED() [2-15](#)
 - MEMORY_SIZEOF() [2-15](#)
 - SIZEOF() [2-16](#)
- Legend dialog box [3-13](#)
- legends
 - Expert Linker [3-11](#)
- LENGTH() LDF identifier [2-28](#)
- librarian
 - VisualDSP++ [4-1](#)
- libraries
 - members [4-1](#)
 - symbol name encryption [4-11](#)
- library files
 - about [A-7](#)
 - adding [3-10](#)
 - creating [4-3](#)
 - searching [4-2](#)
- library members [4-1](#)
 - converting to source code [B-4](#)
 - disassembling [B-4](#)
- library routines
 - using [4-5](#)
- link booting [5-23](#), [6-20](#)
 - ADSP-21161N [6-20](#)
 - BMS pin [5-23](#), [6-21](#)
 - data packing [5-23](#)
 - DMA channel interrupts [5-23](#)
 - external clock signal [5-23](#)
 - IMASK register [5-23](#)
- link buffers
 - buffer 0 [6-20](#)
 - buffer 4 [5-23](#)
- Link page
 - options [1-17](#)
- LINK_AGAINST() LDF command [2-25](#)
- linker
 - about [1-10](#)
 - command-line files (.TXT) [A-6](#)
 - command-line switches [1-33](#)
 - command-line syntax [1-28](#)
 - commands [1-28](#)
 - describing the target [1-20](#)
 - error messages [1-19](#)
 - executable files [A-7](#)
 - file name conventions [1-31](#)
 - memory map files (.MAP) [A-8](#)
 - objects [1-32](#)
 - outputs [1-6](#)
 - overlay constants generated by [1-45](#)
 - selecting a target processor [1-39](#)
 - switches [1-12](#)
 - warning messages [1-19](#)
- linker command-line switches
 - Darchitecture [1-36](#)
 - e [1-38](#)
 - es secName [1-38](#)
 - ev [1-38](#)
 - help [1-38](#)
 - i path [1-39](#)

- keep symbolName 1-39
- L path 1-37
- M 1-37
- Map file 1-37
- MDmacro 1-37
- MM 1-37
- o filename 1-39
- pp 1-39
- proc processor 1-39
- S 1-38
- s 1-40
- sp 1-40
- t 1-40
- T filename 1-38
- v 1-40
- version 1-40
- warnonce 1-40
- xref filename 1-40
- Linker Description Files
 - see .LDF files A-5
- linker.exe 1-2
- linking
 - about 1-10
 - controlling 1-12
 - files with large uninitialized variables 2-64
 - multiprocessor systems 2-65
 - single-processor system 2-62
- loader
 - ASCII-format files (.LDR) A-10
 - guide 6-2
 - hex-format files A-8
 - setting options 5-26, 6-27
 - using 5-26, 6-27
- loader kernels
 - see "boot-kernels" 6-3
- loader routines 6-3
- location counter 2-16
 - definition of 2-16
- M
- M (dependency check and output)
 - linker command-line switch 1-37
- macros
 - LDF 2-17
- Map (file) linker command-line switch 1-37
- MAP() LDF command 2-26
- mapping
 - input sections to output sections 3-12
- MDmacro (macro value) linker
 - command-line switch 1-37
- mem21k.exe 5-11, B-6
- memory
 - allocation 1-21
 - architecture representation 1-20
 - managing heap/stack 3-62
 - overlays 1-41
 - partitions 3-16
 - segment declaration 1-21
 - segments 3-16
 - types 1-21
 - unavailable in SHARC DSPs 3-28
- memory initializer B-6
- memory map files A-8
- Memory Map pane 3-16, 3-17
 - overlays 3-35
 - unavailable regions 3-28

INDEX

- memory maps
 - graphical view [3-22](#)
 - highlighted objects in [3-26](#)
 - specifying [1-21](#)
 - tree view [3-21](#)
 - viewing [3-16](#)
- memory overlays [1-42](#)
- memory segments
 - about [1-3](#)
 - changing size of [3-25](#)
 - gap [3-34](#)
 - MEMORY{} command [3-16](#)
 - rules [1-13](#)
 - size [3-21](#)
 - specifying properties [3-53](#)
 - start address [3-21](#)
- memory spaces [1-42](#)
- MEMORY_SIZEOF() LDF operator [2-15](#)
- MEMORY{} LDF command [1-21](#), [2-6](#), [2-26](#)
- MM (dependency check, output and build) linker command-line switch [1-37](#)
- modifying
 - boot-kernel source files [6-6](#)
 - boot-kernels [5-11](#), [6-6](#)
- Motorola S-record (.S_) files [A-11](#)
- MP systems
 - processor IDs [5-33](#), [6-34](#)
 - semaphores [2-71](#)
- MPMEMORY{} LDF command [2-29](#)
- multiple overlays [3-35](#)
- multiprocessor booting [5-24](#), [6-24](#)
 - from EPROM [5-24](#), [6-26](#)
 - from single EPROM [5-25](#), [6-25](#)
- multiprocessor systems
 - bus lock [2-71](#)
 - semaphores [2-71](#)
- N
 - no boot mode [5-3](#), [5-24](#), [6-3](#), [6-10](#), [6-24](#)
 - no-mem (disable memory initialization) compiler switch [5-11](#)
 - norom (no ROM in PROM) splitter command-line switch [7-9](#)
- O
 - o (output file) linker command-line switch [1-39](#)
 - o (output file) splitter command-line switch [7-8](#)
 - o (output file) splitter switch [7-8](#)
 - object files
 - adding [3-10](#)
 - object properties
 - managing with Expert Linker [3-42](#)
 - objects
 - deleting [3-11](#)
 - sorting [3-15](#)
 - operators
 - LDF [2-13](#)
 - output directory
 - specifying [1-39](#)
 - output sections
 - about [1-11](#)

- dumping 1-22
- rules 1-13
- specifying properties 3-55
- OUTPUT() LDF command 2-7, 3-16
- ov_id_loaded buffer 1-55
- overlay algorithm
 - ALL_FIT 3-61
- overlay library files
 - viewing
 - archive files B-5
- overlay manager
 - about 1-41, 1-43
 - assembly code 2-46
 - constants 1-52
 - major functions 1-44
 - placing constants 1-53
 - PLIT table 1-49
 - routine steps 1-57
- OVERLAY_GROUP{} LDF
 - command 2-31
- OVERLAY_ID LDF identifier 2-57
- OVERLAY_INPUT{} LDF command 2-56
- overlays
 - address 1-46
 - ADSP-21xxx DSPs 2-72
 - constants 1-45, 1-52
 - dumping library files B-5
 - grouped 2-31
 - grouping 2-31
 - in Memory Map pane 3-35
 - live space 3-35
 - loading instructions with PLIT 2-48
 - managing properties 3-60

- memory 1-41, 1-42
- multiple 3-35
- reducing overhead 1-58
- run space 3-36
- symbols 1-63
- ungrouped 2-31
- word size 1-46

P

- packing
 - specifying properties 3-57
- packing mode (PMODE) bits 6-14, 6-18
- pinning
 - to output section 3-19
- PLIT
 - about 1-46
 - allocating space for 2-46
 - constants 2-46
 - executing user-defined code 1-46
 - overlay management 1-43
 - resolving inter-overlay calls 1-65
 - saving register contents 2-48
 - specifying properties 3-46
 - summary 2-48
 - syntax 2-44
- PLIT commands
 - PLIT_SYMBOL_ADDRESS 2-45
 - PLIT_SYMBOL_OVERLAYID 2-45
- PLIT_SYMBOL_ADDRESS 2-45
- PLIT_SYMBOL_OVERLAYID 2-45
- PLIT{} LDF command 2-48
 - about 2-44

INDEX

- in SECTIONS{} [2-56](#)
- pm (include program memory) splitter
 - command-line switch [7-9](#)
- pm (include Program Memory)
 - splitter switch [7-9](#)
- porting
 - from previous SHARCs [6-13](#), [6-17](#), [6-21](#)
- pp (end after preprocessing) linker
 - command-line switch [1-39](#)
- preprocessor
 - running from linker [1-39](#)
- proc (processor)
 - linker command-line switch [1-39](#)
- procedure linkage table (PLIT)
 - about [1-46](#)
 - PLIT{} [2-44](#)
 - using [1-63](#)
- processor
 - selection [1-36](#)
- processor IDs [5-33](#), [6-34](#)
- PROCESSOR{} LDF command [2-50](#)
- processors
 - specifying properties [3-45](#)
- program counter [1-52](#)
- project builds
 - linker [1-15](#)
- Project Options dialog box
 - Link page [1-17](#)
- PROM booting
 - using include-format files [6-11](#)
- PROM files
 - generating [7-1](#)
- PROMs

- setting word width [6-11](#)

R

- r (ROM widths) splitter
 - command-line switch [7-9](#)
- ram (include RAM in PROM) splitter
 - command-line switch [7-9](#)
- rebuilding
 - boot-kernels [5-11](#), [6-6](#)
- references
 - file formats [A-16](#)
- reflective semaphores [2-71](#)
- reset vector address [5-17](#)
- resize cursor [3-26](#)
- RESOLVE() LDF command [2-26](#), [2-52](#)
- RESOLVE_LOCALLY() LDF
 - command [2-58](#)

S

- s (include memory segment) splitter
 - command-line switch [7-10](#)
- s (strip all symbols) linker
 - command-line switch [1-40](#)
- S (strip debug symbols) linker
 - command-line switch [1-38](#)
- SEARCH_DIR() LDF command [2-52](#)
- SECTIONS{} LDF command [1-25](#), [2-7](#), [2-53](#)
- selecting
 - target processor [1-39](#)
- semaphores
 - reflective [2-71](#)
- SHARC DSPs

- broadcast space [2-71](#)
 - overlays [2-72](#)
 - unavailable memory regions [3-28](#)
- shared memory system [2-65](#)
- SHT_NOBITS
 - keyword [2-64](#)
 - section qualifier [2-64](#)
- SIZE() LDF command [2-58](#)
- SIZEOF() LDF operator [2-16](#)
- software
 - development [1-2](#)
- sorting
 - objects [3-15](#)
- source code
 - in input sections [1-3](#)
- source files
 - assembly instructions [A-3](#)
 - C/C++ [A-2](#)
 - fixed-point data [A-3](#)
- sp (skip preprocessing) linker
 - command-line switch [1-40](#)
- specifying
 - loader options [5-26](#), [6-27](#)
 - splitter options [7-2](#)
- SPI control register (SPICTL) [6-23](#)
- SPI port booting [6-22](#)
- SPIRX register [6-3](#), [6-22](#), [6-23](#)
- splitter
 - byte-stacked (.STK) files [A-13](#)
 - byte-stacked files [7-10](#)
 - command-line parameters [7-8](#)
 - command-line reference [7-4](#)
 - command-line switches [7-5](#)
 - command-line syntax [7-4](#)
- file extensions [7-7](#)
 - guide [7-2](#)
 - hex format files
 - .H_ [A-13](#)
 - Motorola S-record files [A-11](#)
 - producing PROM files [7-5](#)
 - reference [7-4](#)
 - using [7-2](#)
- SPORT data files [A-15](#)
- stacks
 - graphic representation [3-62](#)
 - managing in memory [3-62](#)
- symbol declaration [2-5](#)
- symbols
 - adding [3-50](#)
 - deleting [3-52](#)
 - encryption of names [4-11](#)
 - managing properties [3-49](#)
 - removal [1-40](#)
 - viewing [3-41](#)
- SYSCON register [5-10](#), [5-14](#), [5-18](#), [5-20](#), [5-21](#), [6-6](#), [6-10](#)
- T
 - t (trace) linker command-line switch [1-40](#)
 - T file (executable program placement)
 - linker command-line switch [1-38](#)
- target processor
 - specifying [1-36](#)
- tree view
 - memory map [3-21](#)

INDEX

U

- u (user flags) splitter switch [7-10](#)
- unavailable memory regions
 - SHARC DSPs [3-28](#)
- uninitialized variables [2-64](#)
- unmapped object icon [3-12](#)
- utilities
 - archiver [4-1](#)
 - elfar.exe [4-1](#)
 - elfdump.exe [B-1](#)
 - elfloader.exe [5-26](#), [6-27](#)
 - file dumper [B-1](#)
 - linker.exe [1-2](#)
 - loader [5-26](#), [6-27](#)
 - mem21k [5-11](#), [B-6](#)
 - memory initializer [B-6](#)

V

- v (verbose) linker command-line switch [1-40](#)
- version (linker version) linker command-line switch [1-40](#)
- viewing

- .OVL file content [B-5](#)

- VisualDSP++
 - archiver [4-1](#)
 - creating library files [4-3](#)
 - Expert Linker [3-2](#)
 - librarian [4-1](#)
 - loader [5-26](#), [6-27](#)
 - splitter [7-2](#)

W

- warnings
 - linker [1-19](#)
- warnonce (single symbol warning)
 - linker command-line switch [1-40](#)
- wizards
 - Create LDF [3-4](#)
- word width
 - setting for loader output file [6-32](#)

X

- xref (external reference file) linker command-line switch [1-40](#)