VisualDSP++™ 3.0

Linker and Utilities Manual for Blackfin™ DSPs

Second Revision, April 2002

Part Number
82-000410-05

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

Revision 2.0

CONTENTS

PREFACE

EXPERT LINKER

PREFACE

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

Purpose of This Manual

The *VisualDSP++ 3.0 Linker & Utilities Manual for Blackfin DSPs* contains information about the linker and utilities programs for Blackfin™ DSPs. These are 16-bit, fixed-point digital signal 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. The manual leads you through using the linker, archiver, and loader to produce DSP programs and provides reference information on the file utility software.

Intended Audience

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

Manual Contents

The manual contains:

- Chapter 1, "Linker"

  This chapter provides an overview of the linker software and command-line switches; shows how to use the linker description file to define your target DSP system for linking.

- Chapter 2, "Expert Linker"

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

- Chapter 3, "Loader"

  This chapter provides an overview of the loader software and command-line switches; describes boot sequence, shows how to use the different boot-kernels for booting various Blackfin DSPs and creating boot-loadable files.

- Chapter 4, "Archiver"

  This chapter provides an overview of the archiver software and command-line switches. An archiver is used for creating libraries of partially linked objects, speeding linking of often used routines

- Appendix A, "File Formats"

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

- Appendix B, "Utilities"

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

What's New in This Manual

This edition of the manual documents support for all Blackfin processors. In addition to documenting all existing linker and archiver features, this manual describes new macros, commands and switches, including syntax and code examples.

Chapter 2, "Expert Linker" provides a description of the Expert Linker—a new interactive tool to set up and map DSP memory. Chapter 3, "Loader" provides descriptions on new booting sequences, boot loadable formats, boot kernels and command-line switches.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

- Visit the DSP Development Tools website at `http://www.analog.com/technology/dsp/development-Tools/index.html`
- Email questions to `dsptools.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your ADI local sales office or authorized distributor
- Send questions by mail to:

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

Supported Processors

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

- ADSP-21532 DSP
- ADSP-21535 DSP

Product Information

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

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

MyAnalog.com

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

Registration:

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

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

DSP Product Information

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

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

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

Related Documents

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

VisualDSP++ 3.0 Getting Started Guide for Blackfin DSPs

VisualDSP++ 3.0 User's Guide for Blackfin DSPs

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

VisualDSP++ 3.0 C/C++ Assembler and Preprocessor Manual for Blackfin DSPs

VisualDSP++ 3.0 Linker and Utilities Manual for Blackfin DSPs

VisualDSP++ 3.0 Product Bulletin

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

Online Technical Documentation

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

A description of each documentation file type is as follows.

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

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

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

From VisualDSP++

- Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.
- Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

From Windows

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

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

Using Windows Explorer

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

Using the Windows Start Button

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

From the Web

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

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

Printed Manuals

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

VisualDSP++ Documentation Set

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

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

Hardware Manuals

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

Datasheets

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

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

Contacting DSP Publications

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

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

Notation Conventions

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

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

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

1 LINKER

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

This chapter contains:

- "Linking Process Overview" on page 1-3 — provides an overview of linking process, introduces the Linker Description File, and gives an overview of linking environment.
- "Getting Started" on page 1-7 — provides an overview of VisualDSP++ environment, describes the memory map, inputs and output code sources, the Linker Description File, and how the LDF enables your code to run in your target environment to produce an executable.
- "Linker Guide" on page 1-30 — describes Linker Description File syntax, LDF commands, macros and operators. and provides an overview of programming techniques.
- "Advanced Linker Features and Commands" on page 1-62 — describes memory overlays and how they are used with ADI DSPs as well as advanced LDF commands.

- "Linker Command-Line Reference" on page 1-95 — lists linker command-line switches and their syntax.
- "LDF Programming Examples" on page 1-109 — provides a series of programming examples for different types of systems.

The VisualDSP++ linker is one of the components of the VisualDSP++ Integrated Development and Debugging Environment (VisualDSP++ IDDE) that provides complete graphical control of DSP project development process.

This chapter provide an overview of how to link executables for single-processor systems, such as Blackfin's ADSP-21535 and ADSP-21532 DSPs. No multiprocessor support information is provided in this manual.

Most code examples in this manual correspond to the use of the ADSP-21535 DSP.

Linking Process Overview

Figure 1-1 illustrates the DSP software development flow. The process of linking can be split into three phases:

1. Input — C (`.C`), C++ (`.CPP`), or assembly (`.ASM`) source files
2. Linking and the Linker Description File (LDF)
3. Output — an executable file (`.DXE`) as well as shared memory (`.SM`) and overlay files (`.OVL`), where applicable

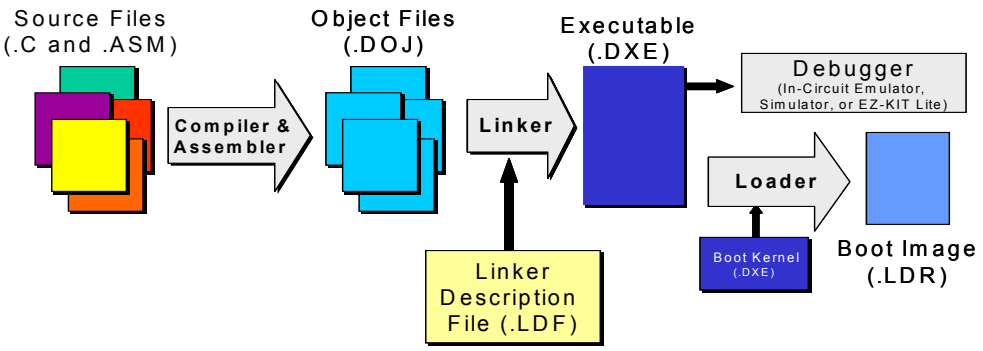

Figure 1-1. Software Development Flow

The process starts with source files, which contain code written in C, C++, or assembly. The compiler, or the developer writing assembly code, will organize each distinct sequence of instructions or data into named sections. These sections will become the main components acted upon by the linker

For more information about input files, see "Inputs — C/C++ and Assembly Sources" on page 1-16.

The next step towards producing an executable is compiling and/or assembling sources into their respective *object files* (`.DOJ`). Each source file produces one object file, comprise of object sections, such that each object section is allocated and labeled according to its respective source file, as shown on Figure 1-2.

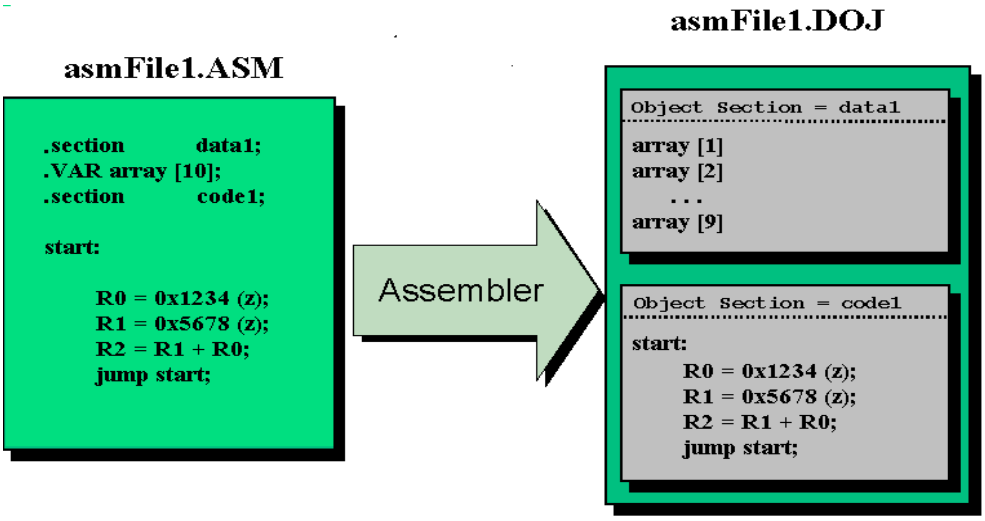

Figure 1-2. Assembly Source and Object Section Names

Whether you want to link a C/C++ function or an assembly routine, the mechanism is the same. Once all of the source files have been converted into objects, the linker combines all of the objects into one integrated executable file (`.DXE`) using directives in the Linker Description File (LDF). This executable may be loaded into a simulator for testing.

Each DSP project must include one LDF file. The LDF specifies the linking process by defining the target memory and the desired mapping of code and data into DSP memory. You can write your own LDF or modify an existing LDF, which is often the easier alternative if you are not dealing

with large changes in your system's hardware or software. VisualDSP++ provides a default LDF to support default mapping for the selected DSP chip.

Just like the object file, the executable consists of different segments, known as *Output Sections*. Input Section names are ***completely independent*** of the output section names. Because they exist in different namespaces, you can have an input section name that is exactly the same as an output section name.

The executable file structure is dictated by the Executable and Linkable Format (ELF) standard, to which the `.DXE` files conform.

Linker operations depend on two types of controls: linker options and linker commands.

Linker options allow you to control how the linker processes your object and library files, specifying such features as search directories, map file output, and dead-code elimination. These options come from linker command-line switches (see "Linker Command-Line Reference" on page 1-95) or, when used within the VisualDSP++ environment, from settings on the **Link** page of the **Project Options** dialog box.

Linker commands, in your project's Linker Description File, define the memory map of your DSP system and the placement of your program's sections within DSP memory. You place the information needed to link your code in the text of these commands.

The VisualDSP++ environment treats the `.LDF` file as a source file in the **Project** window, but this file acts only as command input to the linker.

Using commands in the LDF, the linker:

- Reads the input sections in 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 can be placed into a single memory segment. See "LDF Syntax Overview" on page 1-28 for more information. For a detailed description of the LDF commands, refer to "Linker Guide" on page 1-30.

Getting Started

This section provides an overview of VisualDSP++ environment, describes the memory map, inputs and output code sources, the Linker Description File, and how it enables your code to run in your target environment to produce an executable.

This section contains:

Linking Environment Overview

VisualDSP++ IDDE is intuitive interactive interface for DSP programming. When you open VisualDSP++, you have a work area that contains everything you need to build, manage, and debug your DSP project, including writing an `.LDF` file, mapping code or data to specific memory segments in the `.LDF` file, and generating an executable file.

The linker runs from an operating system command line, which can be issued manually or automatically from the VisualDSP++ environment.

Figure 1-3 shows the VisualDSP++ environment with the **Project** window and an `.LDF` file open in the **Editor** window.

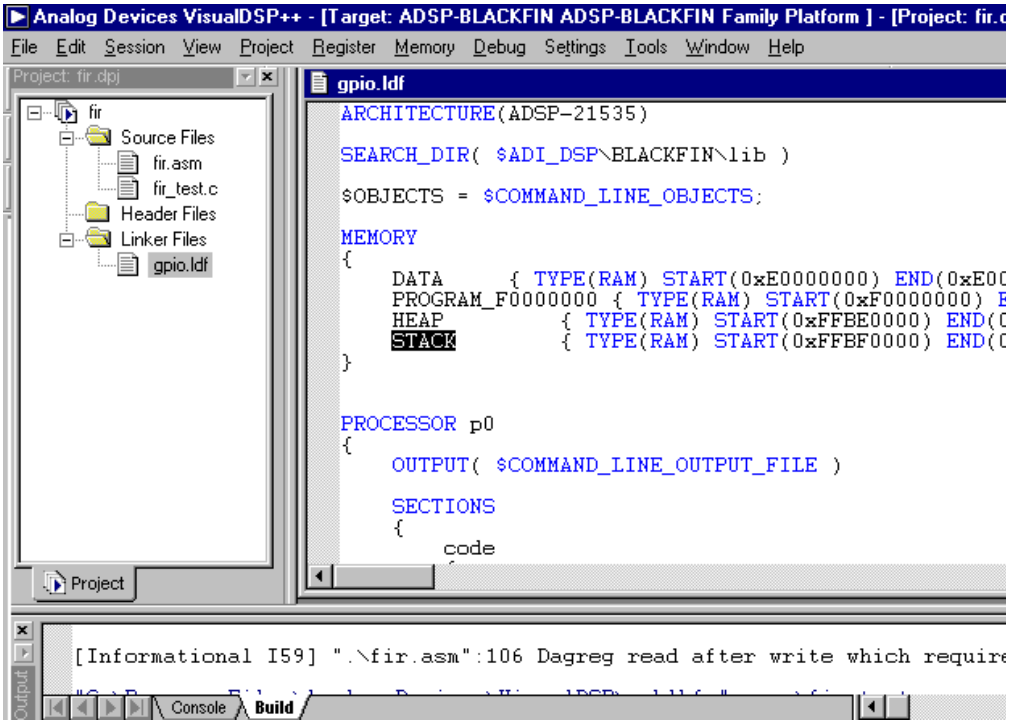

Figure 1-3. VisualDSP++ Environment

When using VisualDSP++, you can specify tool settings for your project builds. You may modify the linker option settings in the VisualDSP++ IDDE. You can do this via the **Link** page of the **Project Project Options** dialog box (selected in the **Project** menu). Figure 1-4 shows the **Link** property page.

Refer to VisualDSP++ online help for more information on VisualDSP++ environment features. The online Help provides a powerful search capability—to get information on a code item, parameter or error, select this item in a window/pane and press **F1** to display appropriate information.

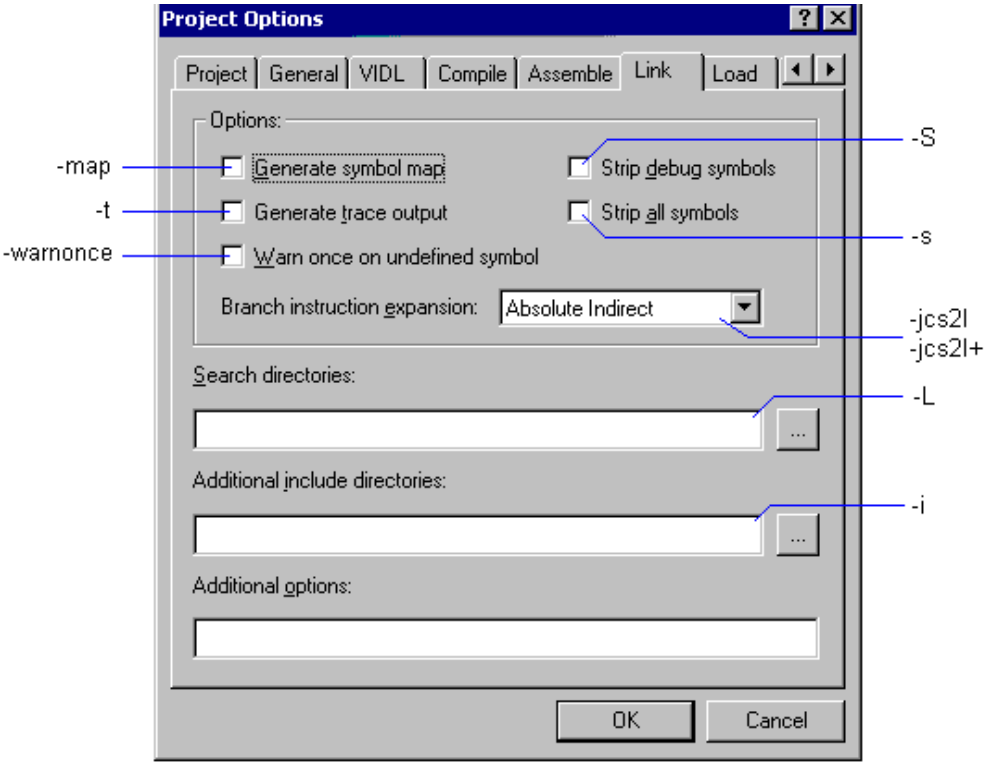

Figure 1-4. Specifying Linker Options

Callouts refer to the corresponding linker command-line switches. The **Additional options** field is used to enter the appropriate file names and options that do not have corresponding controls on the **Link** page but are available as linker switches. See "Linker Command-Line Reference" on page 1-95 for more information).

The VisualDSP++ IDDE also features an *Expert Linker* which is a VisualDSP++ graphical tool that allows you to interactively map code or data to specific memory segments. The Expert Linker takes available project information (object files, LDF macros, libraries and a target memory description) in an `.LDF` file as input and graphically displays it.

You can then use drag-and-drop techniques to arrange the object files in a graphical memory mapping representation. When you are satisfied with the memory layout, you can generate the executable file (.DXE).

Figure 1-5 shows the Expert Linker window with the Input Sections and Memory Map (Output Sections) panes. Refer to Chapter 2 "Expert Linker" for information on the Expert Linker.

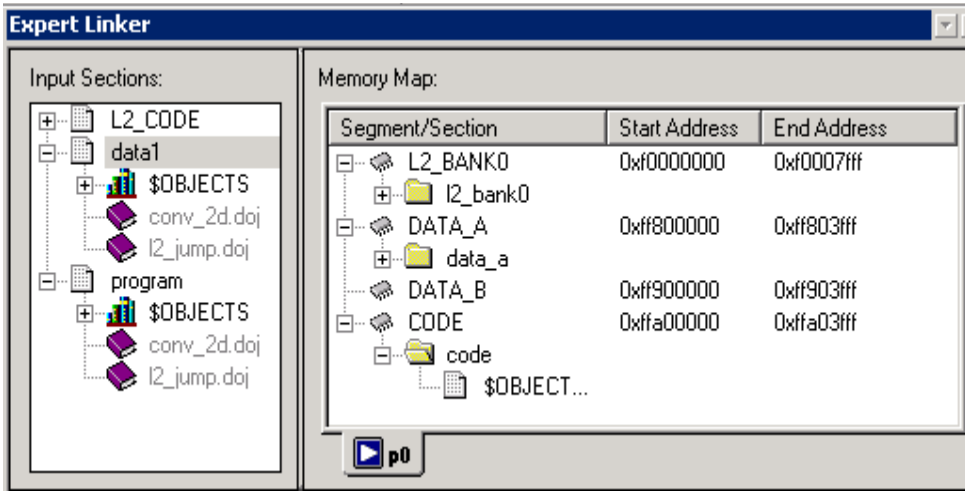

Figure 1-5. Expert Linker Window

Describing the Link Target

Before you define your DSP system's memory and program placement with linker commands, you must analyze the target DSP system and describe it in terms that the linker can process. You then produce an .LDF file for your project, which describes your system attributes:

- DSP system's physical memory map
- Program placement within your DSP system's memory map

If you do not include an LDF, the linker uses a default LDF that matches the `-DPROCESSOR` switch on the linker's command line or VisualDSP++ processor selection option. The examples in this manual are for the ADSP-21535 DSP.

ADSP-21535 DSP Memory Architecture Overview

This section is using the Blackfin ADSP-21535 DSP as an example for describing memory architecture and memory map organization. Other DSPs in the Blackfin DSP family have different memory architectures.

The Blackfin ADSP-21535 DSP includes the L1 memory sub-system, with 16K-byte instruction SRAM/cache, a dedicated 4K-byte data scratchpad, and 32K-byte data SRAM/cache, configured as two independent 16K-byte banks (memories). Each independent bank can be configured as SRAM or cache.

The ADSP-21535 DSP also has an L2 SRAM memory that provides 2M bits (256K bytes) of memory. The L2 memory is unified; that is, it is directly accessible by the instruction and data ports of the ADSP-21535 DSP. The L2 memory is organized as a multi-bank architecture of single-ported SRAMs (there are eight sub-banks in L2), such that multiple accesses can occur in parallel, as long as they are to different banks.

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

Figure 1-6 shows the ADSP-21535 DSP system block diagram.

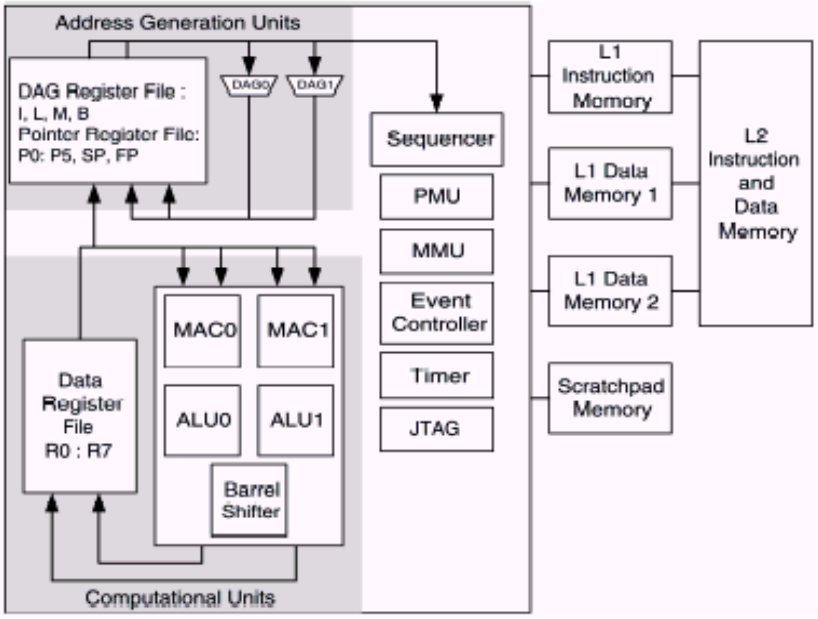

Figure 1-6. ADSP-21535 DSP System Block Diagram

The memory spaces are listed in Table 1-1. The address ranges not listed in the table are reserved.

Table 1-1. ADSP-21535 DSP Memory Map Addresses

Memory Range	Range Description
0xFFE00000 - 0xFFFFFFFF	Core MMR registers (2MB)
0xFFC00000 - 0xFFDFFFFF	System MMR registers (2MB)
0xFFB00000 - 0xFFB00FFF	Scratchpad SRAM (4K)
0xFFA00000 - 0xFFA03FFF	Instruction SRAM (16K)
0xFF900000 - 0xFF903FFF	Data Memory Bank 2 SRAM (16K)
0xFF800000 - 0xFF803FFF	Data Memory Bank 1 SRAM (16K)

Table 1-1. ADSP-21535 DSP Memory Map Addresses

Memory Range	Range Description
0xF0040000 - 0xFF7FFFFF	Reserved RAM
0xF0000000 - 0xF003FFFF	L2 Memory Bank SRAM (256K)
0xEF000400 - 0xEFFFFFFF	Reserved ROM
0xEF000000 - 0xEF0003FF	Boot ROM (1K)
0x00000000 - 0xEEFFFFFF	Unpopulated

The memory sections (defined in Listing 1-1 on page 1-20) assume that only L1 and L2 SRAM are available, and that L1 is unused. See the file `VisualDSP/Blackfin/lib/src/libc/basiccrt.s` for the default startup code, which can be made to initialize the L1 SRAM as cache for L2 (by defining `L1CACHE` while assembling `basiccrt.s`).

For memory architecture of your DSP system, refer to the appropriate Hardware Reference manual.

Representing Memory Architecture

Use the LDF's `MEMORY` command to represent the memory architecture of your DSP system. The linker uses this information to place your executable file in the system's memory. Use the following steps to write a `MEMORY` command:

- List the ways that 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".
- List the types of memory in your system and the address ranges of each type of memory and word width. For Blackfin DSPs, memory can be qualified as `RAM` or `ROM`.

Note: For Blackfin DSPs, word width is always 8 (bits). See "INPUT_SECTION_ALIGN()" on page 1-48 for more information.

- Construct a `MEMORY` command that combines the information in these two lists to declare the memory segments in your system. Use Listing 1-1 on page 1-20 as code example.

Specifying the Memory Map

Your program must conform to the constraints imposed by the processor's data path (bus) widths and addressing capabilities.The steps that follow show a representative LDF associated with a hypothetical project. This LDF specifies several memory segments that support the `SECTIONS` command (see Table 1-2).

The three steps involved in allocating memory for such a project are demonstrated below:

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

   The default LDF handles all the input sections that might be generated by the compiler (the column "Input Section" in Table 1-2). The produced `.DXE` file has appropriate "Output Section" for which material (the corresponding "Input Section") was found in the project's object file. Although not typically used by programmers, the output section labels are only used by downstream tools.

   For example, you can invoke `elfdump.exe` to dump the contents of the `dxe_data1` output section of an executable file. See "ELF File Dumper" on page B-1 for more information on this utility.

Table 1-2 shows Input - Output - Memory correspondences used in the default ADSP-21535's LDF.

Table 1-2. Memory vs. Sections Usage for ADSP-21535 LDF

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

You can modify your LDF to allow for object placement into L1 memories when they are configured as SRAM.

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

   Note: Some portions of the Blackfin DSP memory are reserved. For more information, refer to the Memory chapter in the processor-specific Hardware Reference manual (also see Table 1-1).

3. **Linker MEMORY{} Command** — referring to steps 1 and 2, specify the target's memory with the `MEMORY{}` command in Listing 1-1 on page 1-20.

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 gives object files a `.DOJ` extension.

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

Some input sections also can contain information used to enable source-level debugging and other VisualDSP++ IDDE features. The linker maps each input section (via a corresponding output section in the executable) to a *Memory Segment,* a contiguous range of memory addresses on the target DSP. Each input section in the LDF has a unique name, which you specify in the source code. Depending on whether the source is C, C++ or assembly, there are different conventions for naming an input section (see "LDF Overview" on page 1-18).

Input Section Directives in Assembly Code

The section directive defines a section in assembly source and must precede code or data in an assembly source file. For example,

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

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

Section Directives in C/C++ Source Files

Typically, your code does not specify an input section name, and the compiler uses a default name. The default compiler section names are `program` (for code) and `data1` (for data); additional section names are defined in LDF files for use by the linker.

In a C/C++ source file, you can use the optional `section("name")` C language extension to define "Sections". As the compiler processes the source (as shown in the following example), the compiler stores the code generated from func1 in a separate section of the `.DOJ` file named `ext_code` and the `temp` variable in the section called `ext_data`.

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

Note that the `section("name")` extension is optional as shown in the following example. For more information on LDF sections, refer to "Specifying the Memory Map" on page 1-14.

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

For information on the compiler's default section names, see the *VisualDSP++ C/C++ Compiler & Library Manual for Blackfin DSPs* and Table 1-2 on page 1-15 (column 1).

It is important to identify the difference between Input Section names, Output Section names, and Memory Segment names because these types of names appear in the LDF. Usually, default name conventions are used. However, there may be a situation when you may want to specify non-default names. This happens when various functions or variables (in the same source file) are to be placed into different memory segments.

LDF Overview

The *Linker Description File* (LDF) is used to direct linking operation by mapping code or data to specific memory segments. The linker maps your program code (and data) within the system memory and processor(s), assigning an address to every global symbol, where

`symbol = label/function_name` or `variable name)`

If you neither write nor import an LDF into your project, the VisualDSP++ IDDE uses a *default LDF* to link your code. The default LDF is based on the processor specified in the project options. The default LDF file is packaged with your DSP tool distribution kit in a subdirectory specific to your target processor's family (for example, Blackfin DSPs or TigerSHARC DSPs).

One default LDF is provided for each DSP supported by your VisualDSP++ installation. The default LDF reflects your target processor architecture, as specified by your project's options, and is set according the "*processor*" selection.

You can use your own LDF. However, modifying an existing or a default LDF is often the easier alternative if you are not dealing with large changes in your system's hardware or software.

Figure 1-7 shows how the LDF combines information, directing the linker to place program sections in an executable according to the memory available in the DSP system.

The linker may output warning messages and error messages. Be sure to resolve errors to enable the linker to produce valid output. See "Linker Error and Warning Messages" on page 1-43 for more information.

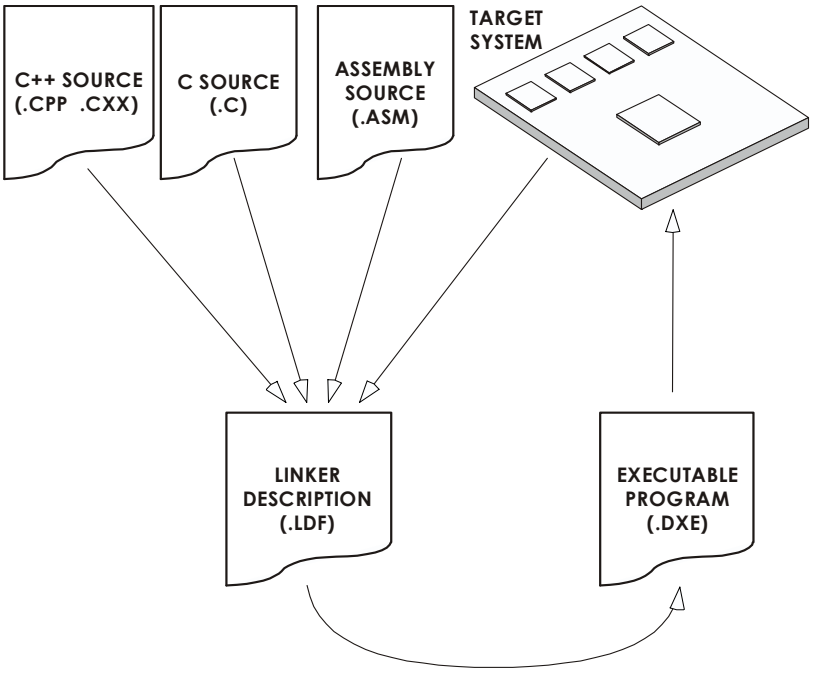

Figure 1-7. LDF File in Linking Process

The linking process is based on the following:

1. Each source file produces one object file.
2. Your source code specifies one or more input sections as destinations for its compiled/assemble object(s).
3. The compiler and assembler produce object code, with labels directing which portion(s) are allocated to which input sections.

4. The linker maps each input section in the object code to an output section, as directed by the LDF.
5. The linker maps each output section to a memory segment, which is a contiguous range of memory on the target, as specified by the LDF.

Each input section may contain multiple code items, but a code item appears in only one input section. More than one input section can be placed in any output section. Each memory segment may have its own (allowed) width. Contiguous addresses on different-width hardware must be in different segments. More than one output section may map to a single memory segment if these output sections fit completely in the memory segment.

Listing 1-1 shows an example of a basic LDF file (formatted for easy reading). Note that this LDF file includes two commands (`MEMORY` and `SECTIONS`) that combine program and system information. For more information refer to "Notes on Basic LDF Example" on page 1-22.

Other LDF examples for assembly and C source files are in "LDF Programming Examples" on page 1-109.

Listing 1-1. Default Sample LDF -- Basic Example

```
ARCHITECTURE(ADSP-21535)
SEARCH_DIR($ADI_DSP\Blackfin\lib)
$OBJECTS = CRT, $COMMAND_LINE_OBJECTS ENDCRT;

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

```
   MEM_ARGV
   { TYPE(RAM) START(0xF003FE00) END(0xF003FFFF) WIDTH(8) }
}

PROCESSOR p0 /* The processor in the system */
{
   OUTPUT($COMMAND_LINE_OUTPUT_FILE)

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

      dxe_L2
      {
         INPUT_SECTION_ALIGN(2)
         /* Align all code sections on 2 byte boundary */
         INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
         INPUT_SECTION_ALIGN(1)
         INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
         INPUT_SECTION_ALIGN(1)
         INPUT_SECTIONS( $OBJECTS(constdata)
                         $LIBRARIES(constdata))
         INPUT_SECTION_ALIGN(1)
         INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
      } >MEM_L2

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

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

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

```
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } >MEM_HEAP

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

    } /* end SECTIONS */

} /* end PROCESSOR p0 */
```

As previously noted, the linker is automatically invoked for builds using the VisualDSP++ IDDE, but can also be run from the command line.

Notes on Basic LDF Example

In the following discussion, commands for connecting your program to the target DSP are `MEMORY` and `SECTIONS`. For information on all LDF commands and their syntax, see "Linker Guide" on page 1-30.

You can define new symbols within the LDF; 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.The `INPUT_SECTIONS` statement specifies the object file that the linker uses to resolve the mapping.

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

- ARCHITECTURE(x) names the target architecture. It thereby specifies possible memory widths and address ranges, the register set, and other structural information for use by the debugger, linker, loader, splitter and utility software. The target architecture (x) must be installed in VisualDSP++.

- SEARCH_DIR specifies path name(s) to search for libraries and object files. This example shows one search directory, the single argument to the SEARCH_DIR command. For more information, see "SEARCH_DIR()" on page 1-55.

  The linker can support a sequence of search directories presented as an argument list (dir1, dir2,...). When searching for an object or library file, the linker follows this sequence and stops at the first match.

- $OBJECTS is an example of a string macro, which are used to make the LDF easier to read; you can substitute short macros for long text strings. While conceptually similar to preprocessor macro support (#defines), also available in the LDF, the string macros are independent. In this listing, $OBJECTS is synonymous with $COMMAND_LINE_OBJECTS, and expands to a comma-delimited list of object files to be linked together.

  Note: In this code example and in the default LDFs accompanying VisualDSP++, $OBJECTS is used in the SECTIONS() command to specify which object files should be searched for specific INPUT SECTIONS.

  For example, $ADI_DSP expands to the home directory for VisualDSP++.

- $COMMAND_LINE_OBJECTS (see more on page 1-41) is an LDF *command-line* macro, which expands to list all of the input object (.DOJ) files in the linker's command line. Remember that all linker

invocations from the VisualDSP++ IDDE have command-line equivalents. When using VisualDSP++, $COMMAND_LINE_OBJECTS includes the .DOJ file from every source file in the VisualDSP++ **Project** window.

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

VisualDSP++ currently uses a linker's command line that lists objects in alphabetical order which carries through to the $OBJECTS macro. One may customize the LDF to link objects in any order desired. Rather than use macros such as $OBJECTS as the defaults do, each INPUT_SECTION command could have one or more explicit object names. The following examples are functionally identical.

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

dxe_program { INPUT_SECTIONS($DOJS(program)) > mem_program

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

- Each PROCESSOR command (see more on page 1-53) encapsulates the linker commands to generate a single executable.
- The MEMORY command (see more on page 1-50) 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 (program, data, stack...). It thereby connects your program to the target system.

Note: Memory segments must have distinct names; however, the memory names occupy different namespaces from input section and output section names. Therefore, a memory segment and an output section may have the same name. In this example, the memory segment and output section are named as `MEM_L2` and `DXE_L2` because the memory holds both program (`program`) and data (`data1`) information.

- The `OUTPUT()` command (see more on page 1-54) directs the linker to produce an executable (`.DXE`) file, specifying the filename. In this listing, the argument to the `OUTPUT` command is the `$COMMAND_LINE_OUTPUT_FILE` macro (see more on page 1-41).

  Therefore, the linker names the executable according to the text following the `-o` switch (which corresponds to the name specified in the **Project Options** tab when the linker is invoked through the VisualDSP++ IDDE).

  ```
  linker ... -o outputFilename
  ```

- `SECTIONS` (see more information on page 1-56) defines the placement of code and data in physical memory. The linker takes sections from object files as inputs, places them in output sections, and maps output sections to the memory segments declared in the `MEMORY` command.

  The `INPUT_SECTIONS` command can be interspersed within an output section with other directives, including location counter information (see more information on page 1-58).

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

```
  dxe_L2
1     INPUT_SECTIONS_ALIGN (2)
2     INPUT_SECTIONS($OBJECTS(program)
                          $LIBRARIES(program))
3     INPUT_SECTIONS_ALIGN (1)
4     INPUT_SECTIONS($OBJECTS(data1)
                         $LIBRARIES(data1))
  }>MEM_L2
```

- The second line directs the linker to place the object code assembled from the source file's "`program`" input section (via the "`.section program`" directive in the assembly source file), place the output object into the "`DXE_L2`" output section, and map it to the "`MEM_L2`" memory segment. The fourth line does the same for the input section "`data1`" and output section "`DXE_L2`", mapping them to the memory segment "`MEM_L2`".
- The two pieces of code follow each other in the `program` memory segment. The `INPUT_SECTIONS()` commands are processed in order, so the program sections appear first, followed by the `data1` sections. The program sections will appear in the order the object files appear in the `$OBJECTS` macro.

Placing Code on the Target

As the simple example above shows, the `SECTIONS()` command defines the mapping of code and data into the physical memory of a processor in a DSP system.

To write a linker `SECTIONS{}` command (per system architecture in Figure 1-6 on page 1-12 and Listing 1-1 on page 1-20):

1. List the input sections defined by your source code.

   When using an assembly file, list each of the assembly code `.SECTION` directives in your DSP program, identifying their mem-

ory types (program or data1) and noting when location is critical to their operation. These .SECTIONS portions include interrupt tables, data buffers, and on-chip code or data.

When using a C/C++ source file, remember the compiler will generate sections with the names program and data1 for the code and data. These sections correspond to your source if you do not specify a section through the section() operator.

2. Compare this list with the segments you defined in the MEMORY command, identifying the memory segment in which each .SECTION must be placed.
3. Combine the information from these two lists to write one or more linker SECTIONS{} command(s). Combining the information from steps 1 and 2, you could specify how to place code for the system with the SECTIONS{} command in Listing 1-1 on page 1-20.

SECTIONS() command always must appear within the context of a PROCESSOR() or SHARED_MEMORY() command.

Passing Arguments for Simulation/Emulation

To support simulation/emulation, the linker should get the start address and buffer length of the argument list from the ARGV section of the .LDF file (see in Listing 1-1).

To set the address, you just have to edit your .LDF file (see changes in the .LDF file in Listing 1-1):

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

If you modify the .LDF file and change the start or end of the MEM_ARGV section within the .LDF file, update the entry in the VisualDSP++ IDDE as well, via the following menu selection:

Settings->Simulator->Command Line Arguments->Command Line Arguments Base Address

Refer to *VisualDSP++ 3.0 User's Manual for Blackfin DSPs* or online Help for more information on simulator and command-line arguments.

Do not try to use command-line arguments for ADSP-21535 linked programs without first modifying the .LDF file to allocate a buffer suitable for your application.

LDF Syntax Overview

The LDF allows you to develop code for any system that contains a DSP. The syntax of the LDF defines your system to the linker and specifies how the linker processes executable code for your system. The LDF uses commands, expression, operators, macros and keywords to control code development and execution. Refer to "Linker Guide" on page 1-30 for more information.

Outputs — DSP Executables

After you have compiled or assembled source files into object files, use the linker to combine all of the object files into one integrated executable file. By default, the development software gives executable files an .DXE extension.

Like object files, the executable is partitioned into *Output Sections* with their own names. These output sections are defined by the ELF (*Executable and Linking Format*) file standard that the development software conforms to for executable files.

The executable's input and output section names occupy different namespaces. Because they are independent of each other, you may have sections with the same names. The linker uses input section names as the labels in order to find the corresponding input sections within object files.

It is important to understand the function of the `.DXE` executable file. It is not loaded into the DSP, nor is it burned into an EPROM. The `.DXE` file contains the raw code and data from the object files, along with additional information which is used by utilities (such as the debugger) to locate code in the target (DSP, simulator, ICE, ...).

The loader is used to process the `.DXE` executable file to generate a boot-loadable file for your target system. Refer to Chapter 3 "Loader" for more loader information.

Getting Started Summary

The "*Getting Started*" section showed how to assign code and data to memory and how to create an executable. It is recommended to use the linker's predefined macros for file searches, input, and output in order to write simple, maintainable linker description files.

For more information, see "Linker Guide" on page 1-30, "LDF Macros" on page 1-40, and "LDF Programming Examples" on page 1-109.

After your `.DXE` file is created, use the simulator or emulator to test it. Refer to the *VisualDSP++ 3.0 User's Guide for Blackfin DSPs* for more information on code testing and debugging.

Linker Guide

The *Linker Description File* (LDF) allows you to develop a code for any system that contains a DSP. The syntax of the LDF lets you define your system to the linker and specify how the linker processes executable code for your system. This reference describes LDF syntax and provides LDF examples for typical systems.

This section contains:

Because the linker runs the preprocessor on the LDF, you can use preprocessor commands (such as `#defines`) within your LDF. For information on preprocessor commands, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for Blackfin DSPs*.

Refer to "LDF Programming Examples" on page 1-109 for LDF examples and discussions supporting several typical system models and source files.

The Blackfin DSP architecture does not support `MULTIPROCESSOR`, `PACKING`, and `SHARED_MEMORY` commands.

LDF Structure

One way to produce a simple, maintainable LDF is to structure the file so that it parallels the structure of your DSP system. Using your system as a model, follow these guidelines for structuring your LDF:

- Split the LDF into a set of `PROCESSOR{}` commands, one for each DSP in your system.
- Put each `MEMORY{}` command in the LDF scope that matches your system, defining memory that is unique to a processor within the scope of the corresponding `PROCESSOR{}` command. Define common memory definitions (shared) are declared in the global LDF scope, before any `PROCESSOR{}` commands.
- Place the `SHARED_MEMORY{}` command in the global LDF scope if they apply to your system. This command represents system resources available as shared resources.

For more information on LDF structure, see "Describing the Link Target" on page 1-10, "Placing Code on the Target" on page 1-26, and "LDF Programming Examples" on page 1-109.

Command Scoping

The two `LDF` scopes are *global* and *command*. A command scope applies to all commands that appear between the braces { } of another command, such as a `PROCESSOR{}` or `PLIT{}` command.

The global scope occurs outside commands. Commands and expressions that appear in the global scope are available globally and visible in all subsequent scopes.

The effects of commands and expressions that appear in the command scopes are limited to those scopes. Note that LDF macros are available globally, regardless of the scope where the macro is defined (see "LDF Macros" on page 1-40).

Figure 1-8 demonstrates some scoping issues.

For example, the `MEMORY{}` command that appears in the global LDF scope is available in all the command scopes, but the `MEMORY{}` commands that appear in the command scopes are restricted to those scopes.

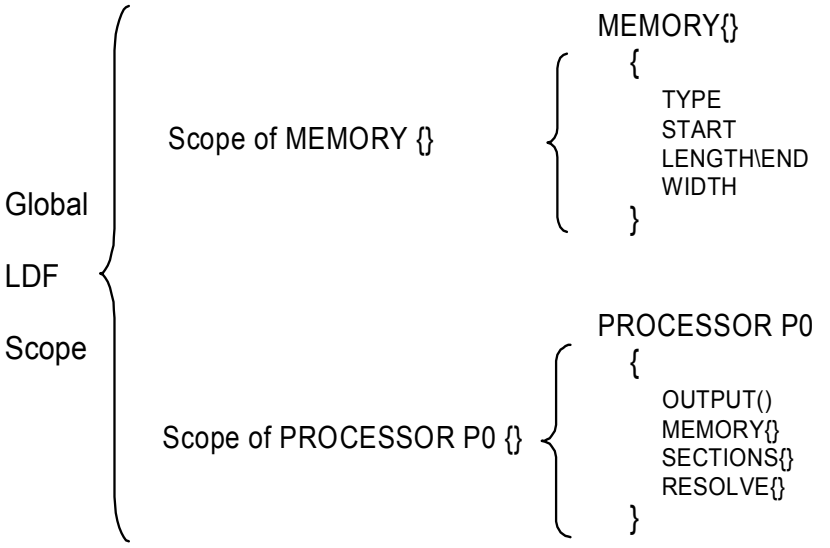

Figure 1-8. LDF Command Scoping Example

LDF Expressions and Conventions

Table 1-3 lists the linker's non-keyword operators and conventions.

Table 1-3. Linker Non-Keyword Operators and Conventions

Convention	Description
.	A dot in an address expression refers to the current location counter (described on page 1-39).
`0x`*`number`*	A "`0x`" prefix indicates a hexadecimal number

Table 1-3. Linker Non-Keyword Operators and Conventions (Cont'd)

Convention	Description
number	A number without a prefix is a decimal number
*number*k (or K)	A decimal number multiplied by 1024
/* comment */	C style comments: can cross newline boundaries until */ is encountered.
// comment	A "//" string precedes single-line C++ style comments

Linker commands may contain arithmetic expressions. These expressions follow the same syntax rules as C/C++ language expressions. The linker handles expressions as follows.

- The linker evaluates all expressions as type `unsigned long` and treats all constants as type `unsigned long`.
- The linker supports all C/C++ language arithmetic operators.
- The linker lets you define and refer to symbolic constants in the linker description file.
- The linker lets you refer to global variables in the program being linked.
- The linker recognizes labels conforming to the following constraints:
    - Must start with a letter, underscore, or point
    - May contain any letters, underscores, digits, and points
    - Are white space delimited
    - Do not conflict with any keywords, and are unique

Linker Keywords, Commands and Operators

Descriptions of linker keywords from Table 1-4 appear in the following sections.

- "Miscellaneous LDF Keywords" on page 1-35
- "LDF Operators" on page 1-35
- "LDF Macros" on page 1-40
- "LDF Command Summary" on page 1-44

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

Table 1-4. Linker Keywords Summary

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BOOT	COMAP
DEFINED	DYNAMIC	
ELIMINATE	ELIMINATE_SECTIONS	END
FALSE	FILL	FIRST_FIT
INCLUDE	INPUT_SECTION_ALIGN	INPUT_SECTIONS
KEEP	LENGTH	LINK_AGAINST
MAP	MEMORY	MEMORY_SIZEOF
NUMBER_OF_OVERLAYS	OUTPUT	
OVERLAY_GROUP	OVERLAY_ID	OVERLAY_INPUT
OVERLAY_OUTPUT	PACKING	PLIT
PLIT_DATA_ OVERLAY_IDS	PLIT_SYMBOL_ ADDRESS	PLIT_SYMBOL_ OVERLAYID
PROCESSOR	RAM	

Table 1-4. Linker Keywords Summary (Cont'd)

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

Miscellaneous LDF Keywords

The miscellaneous linker keywords are not operators, macros, or commands. They are:

- BOOT—Boot memory from which a Blackfin DSP can be booted.
- FALSE—A constant with a value of `0`.
- TRUE—A constant with a value of `1`.
- XREF—A cross-reference option setting.

For more information about other linker keywords, see "LDF Operators" on page 1-35, "LDF Macros" on page 1-40 and "LDF Command Summary" on page 1-44.

LDF Operators

LDF operators in expressions support memory address operations. Expressions containing these operators terminate with a semicolon, except when you use the operator as a variable for an address. The linker responds to several LDF operators: `ABSOLUTE`, `ADDR`, `DEFINED`, `MEMORY_SIZEOF`, `SIZEOF`, and . (location counter).

ABSOLUTE () Operator

Syntax:

```
ABSOLUTE(expression)
```

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

- A symbolic expression, in parentheses. Note that

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

  gives `ldf_start_expr` the value corresponding to the address of the symbol start, plus 8, as in

  ```
  Ldf_start_expr = start + 8;
  ```

- An integer in one of the following forms: hexadecimal, decimal, or decimal followed by "K" for kilo (x 1024) or "M" for Mega (x 1024 x 1024).
- The period, indicating the current location (see "Location Counter (.)" on page 1-39).
- The statement defining the bottom of stack space in the LDF

  ```
  ldf_stack_space = .;
  ```

  could also be written as

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

- A symbol name.

ADDR(*section name*) Operator

Syntax:

```
ADDR(section_name)
```

This operator returns the start address of the named section — an "*output section*" defined in the LDF. The operator is useful for assigning a section's absolute address to a symbol. For example, if an `.LDF` file defines some output sections as:

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

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

then the LDF may contain the command

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

The linker generates the constant `ldf_start_ctor`, assigning the start address of the `ctor` output section.

DEFINED Operator

Syntax:

```
DEFINED(symbol)
```

The linker returns the value `1` if the symbol appears in the linker's symbol table and the value `0` if the symbol is not defined. This operator is useful for assigning default values to symbols.

For example, if an assembly object linked by the LDF defines the global symbol `test`, then the statement

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

sets the constant `test_present` to 1; otherwise the constant will have the value 0.

MEMORY_SIZEOF Operator

Syntax:

```
MEMORY_SIZEOF(segment_name)
```

This operator returns the size, in words, of the memory segment *segment_name*. This operator is useful when knowing a segment's size helps with moving the current location counter to an appropriate location.

The following code example (from a default LDF) demonstrates use of the location counter and the `MEMORY_SIZEOF` operator to set linker-generated constants.

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

This code example defines the `sec_stack` section to consume the entire `mem_stack` segment.

SIZEOF Operator

Syntax:

```
SIZEOF(section_name)
```

This operator returns the size, in bytes, of the section *section_name*.

The operator is useful when knowing a section's size helps with moving the current location counter to an appropriate memory location.

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

```
Data1
{
    INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
    _sizeofdata1 = SIZEOF(data1);
} > MEM_DATA1
```

Location Counter (.)

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

When manipulating the . operator:

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

For an example of location counter usage, see the sample code in Listing 1-1 on page 1-20.

LDF Macros

Macros are names of text strings. They may be assigned values, textual or procedural, or simply declared to exist. The linker supports three way of treating macros in LDFs:

- Substituting the string value for the name. Normally the string value is longer than the name, so the macro "expands" to its textual length.
- Performing actions conditional on the existence or value of the macro.
- Assigning a value to the macro, possibly as the result of a procedure, then use that value in further processing.

Some macros are built-ins, with predefined procedures or values, which may be system-specific. These are called *linker* (or LDF) macros, and are described in this section. Others, called *user* macros, are user-defined.

Macros are identified with leading dollar signs (`$`).

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 on these topics, see "Command Scoping" on page 1-31 and "LDF Macros and Command-Line Interaction" on page 1-42.

LDF macros are independent of preprocessor macro support which is also available in the LDF. Preprocessor macros (or other preprocessor commands) are placed by the preprocessor into source files. The preprocessor macros are useful for repeating instruction sequences in your source code or defining symbolic constants. These macros facilitate text replacement, file inclusion, and condi-

tional assembly and compilation. For example, the assembler's preprocessor use the `#define` command to define macros and symbolic constants.

Refer to *VisualDSP++ 3.0 C/C++ Compiler & Library Manual for Blackfin DSPs* and *VisualDSP++ 3.0 Assembler and Preprocessor Manual for Blackfin DSPs* for more information on preprocessor functions.

LDF Macro List

The linker provides the following LDF macros:

- `$COMMAND_LINE_OBJECTS`

  The linker 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`

  The linker expands to the list of executable (`.DXE` or `.SM`) files that are 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 when resolving external symbols.

- `$COMMAND_LINE_OUTPUT_FILE`

  The linker expands to the output executable file name, which is set with the linker's `-o` switch. This file name corresponds to the

<projectname.dxe> set via VisualDSP++ **Project** settings. Use this macro only once in your LDF for file name substitution within an OUTPUT() command.

- $ADI_DSP

  The linker expands this macro into the path to the installation directory. Use this macro to control how the linker searches for files.

- $macro = *file1, file2, file3, ...* ;

  The linker supports user-defined macros for file lists. Use this syntax to define $*macro* as a comma-delimited *file1, file2,* etc. After you define a $*macro*, the linker substitutes *files* for the $*macro* where it subsequently appears in the LDF. Terminate a $*macro* declaration with a semicolon. The linker processes the files in the order that they appear.

LDF Macros and Command-Line Interaction

Whether you run the linker automatically from the VisualDSP++ IDDE or explicitly from a command line, the linker gets its commands through a command-line interface. Many linker operations, such as input, output, and link-against files can be controlled through the command line entries. Using LDF macros, you can apply these command-line inputs throughout your LDF.

Whether you should use the command-line inputs in the LDF or control the linker with LDF code depends on these two criteria:

- Writing an .LDF file that uses command-line inputs can produce a more generic LDF that you can use for multiple projects. Because you can only specify a single output from the command line, an .LDF file that relies on command-line input should be written to produce one output file for a single-processor system.

- Writing an .LDF file that *does not use* command-line inputs can produce a more specific LDF that you can use with more complex linker features.

Linker Error and Warning Messages

The linker writes link warnings and errors to the VisualDSP++ **Output** window (or standard output in the command-line version of the linker). Linker warning and error messages describe problems that the linker encountered when processing the Linker Description File.

A linker *warning* message indicates a processing error which does not keep the linker from producing a valid output file. For example, the presence of an unused symbol in your code will produce a warning.

The linker issues an *error* message when it encounters an error that prevents it from producing a valid output file. Typically, these messages include the LDF name, line number containing the error, and a brief description of the error condition.

For 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)
```

When developing within the VisualDSP++ environment, the **Output** window's **Build** page displays project build status and error messages. In most cases, you can double-click on a message or error number to display the line in the source file that contains the error. You can access all linker's error messages and their descriptions through **Error Messages** in the VisualDSP++ online Help.

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

For example, if an input section from the object file is not placed by the LDF, there will be a cross-reference error in every object that refers to labels in the missing section. You can solve this problem by reviewing the LDF and correcting it by specifying all sections that need placement.

For more information, see the *VisualDSP++ 3.0 User's Manual for Blackfin DSPs*

LDF Command Summary

Commands in the LDF define the target system and specify the order in which the linker processes output for that system. Linker 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 1-31.

This linker supports the following LDF commands:

- "ALIGN()" on page 1-45
- "ARCHITECTURE()" on page 1-45
- "ELIMINATE()" on page 1-47
- "ELIMINATE_SECTIONS()" on page 1-47
- "DYNAMIC()" on page 1-46
- "INCLUDE()" on page 1-47
- "INPUT_SECTION_ALIGN()" on page 1-48
- "KEEP()" on page 1-49
- "LINK_AGAINST()" on page 1-49
- "MAP(filename)" on page 1-50

- "MEMORY{}" on page 1-50
- "OVERLAY_GROUP{} Command" on page 1-82
- "PLIT{} Command" on page 1-86
- "PROCESSOR{}" on page 1-53
- "RESOLVE()" on page 1-55
- "SEARCH_DIR()" on page 1-55
- "SECTIONS{}" on page 1-56

The Blackfin DSP architecture currently does not support `MULTIPROCESSOR`, `PACKING`, and `SHARED_MEMORY` commands.

ALIGN()

The linker uses the `ALIGN(`*`address_boundary_expression`*`)` command to align the address of the current location counter to the next address that is a multiple (power of 2) of *`address_boundary_expression`*. The *`address_boundary_expression`* is a word boundary (address), which depends on the word size of the segment where the `ALIGN()` is taking place.

ARCHITECTURE()

The `ARCHITECTURE()` command specifies the processor in your target system. Your LDF may contain only one `ARCHITECTURE()` command. The command must appear with global LDF scope, applying to the entire linker description file. The syntax for this command is:

```
ARCHITECTURE(processor)
```

The `ARCHITECTURE()` command is case sensitive. Hence, `ADSP-21535` is a legal value but `adsp-21535` is not.

If you do not specify the target processor with the ARCHITECTURE() command, it must be in the command line (linker -D<architecture> ...). Otherwise, the linker cannot link your program. If the processor-specific MEMORY{} commands in the LDF conflict with the processor type, the linker issues an error message and halts.

To test whether your VisualDSP++ installation accommodates a particular processor, type:

```
linker -D<your target architecture>
```

at a command line. If the architecture is not installed, the linker prints out a message to that effect.

DYNAMIC()

The DYNAMIC() command allows a creation of a Dynamic Linking Object (DLO) to be located at runtime by a run-time operating system with a dynamic linker component. The names of the overlay output files (.OVL) are written out in the .DYNAMIC section. The syntax for this command is:

```
DYNAMIC ([resolve_locally], outputFileName, sections, [plit]);
```

The arguments are defined as follows:

- resolve_locally — A boolean variable indicating whether the linker should generate PLIT entries for function calls that can be resolved within the DLO. A value of TRUE (the default) instructs the linker *not* to generate PLIT entries for function calls that can be resolved within the DLO; a value of FALSE instructs the linker to generate PLIT entries for each and every function call.
- *outputFileName* — The name of the linker output file for this processor. The specified name *must* be a DLO file.

- `sections` — The output section for this processor. See "SECTIONS{}" on page 1-56 for more information about the sections command.
- `plit` — A processor-specific PLIT description. See "PLIT{} Command" on page 1-86 for more information about PLITs.

ELIMINATE()

The linker uses the `ELIMINATE()` command to turn on object elimination, removing symbols from the executable if they are not called. If the `VERBOSE` keyword is added (for example, `ELIMINATE(VERBOSE)`), the linker reports on objects as they are eliminated. This command is performs the same function as the `-e` command-line switch.

ELIMINATE_SECTIONS()

The linker uses the `ELIMINATE_SECTIONS(sectionList)` command to turn on section elimination, removing symbols ONLY from the listed sections of the executable if they are not called. The sectionList is a comma-delimited list of sections. Verbose elimination can also be obtained by specifying `ELIMINATE(VERBOSE)`. This command performs the same function as the `-es` command-line switch.

INCLUDE()

This command specifies an additional LDF that the linker processes before processing the remainder of the current LDF. You may specify any number of additional files. Supply one filename per `INCLUDE()` command.

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

INPUT_SECTION_ALIGN()

The INPUT_SECTION_ALIGN() command instructs the linker to align each input section (instruction or data) placed in an output section to an address satisfying the *address_boundary_expression*. The address boundary expression (a power of 2) is a word boundary (address). Legal values for this expression depend on the word size of the segment that receive the output section being aligned.

The linker fills any "holes" created by the INPUT_SECTION_ALIGN() instructions with zeroes (by default), or with the value specified with the preceding FILL command valid for the current scope. For more information, see FILL on page 1-59.

The INPUT_SECTION_ALIGN(*address_boundary_expression*) command is valid only within the scope of an output section. For more information, see "Command Scoping" on page 1-31. For more information on output sections, see the syntax description for "SECTIONS{}" on page 1-56.)

In the following example, the input sections from a.doj, b.doj, and c.doj *will* be aligned on even addresses. However, the input sections from d.doj and e.doj will *not* be aligned, because the INPUT_SECTION_ALIGN(1) command indicates the subsequent sections are not subject to the input section alignment.

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

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

KEEP()

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

LINK_AGAINST()

The `LINK_AGAINST()` command directs the linker to check specific executables to resolve variables and labels that have not been resolved locally.

To link multiprocessor programs, you must use the `LINK_AGAINST()` command in your LDF. The Blackfin DSP architecture currently does not support multiprocessor systems.

This command is an optional part of the `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` — the processor name (typically `0`, `1`, ...). For example, `p0` or `01`.
- *`executable_file_names`* — a list of one or more executable (`.DXE`) or shared memory (`.SM`) files. If a list of file names is given, the names are separated by white space.

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

You can override the search order for a specific variable or label by using the `RESOLVE()` command (see on page 1-55), which directs the linker to ignore `LINK_AGAINST()` for a specific symbol. `LINK_AGAINST()` for other symbols still applies. Example LDFs containing the `LINK_AGAINST()` and `RESOLVE()` commands are useful for seeing how this process works. For more information, see Listing 1-10 on page 1-112.

MAP(filename)

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

This command corresponds to and is overridden by the `-Map <`*`filename`*`>` command line switch. If the VisualDSP++ project's options include generating a symbol map (via the **Link** page of the **Project Options** dialog box), the linker runs with `-Map <`*`projectname`*`>.map` asserted, and your LDF's `MAP()` command generates a warning.

MEMORY{}

The linker's `MEMORY{}` command specifies the memory map of your target system. After you declare memory segment names with this command, you can use the memory segment names for placing program `SECTION`s through the `SECTIONS{}` command.

The LDF may contain a `MEMORY{}` command that applies to each processor's scope, and must contain a `MEMORY{}` command for any global memory on your target system. There is no limit to the number of segments you can declare within each `MEMORY{}` command. For more information, see "Command Scoping" on page 1-31.

In each scope scenario, follow the MEMORY{} command with a SECTIONS{} command. Use the memory segment names for placing program SECTIONs. Only memory segment declarations may appear within the MEMORY{} command. There is no limit on 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 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 1-9, followed by definitions for the command's component.

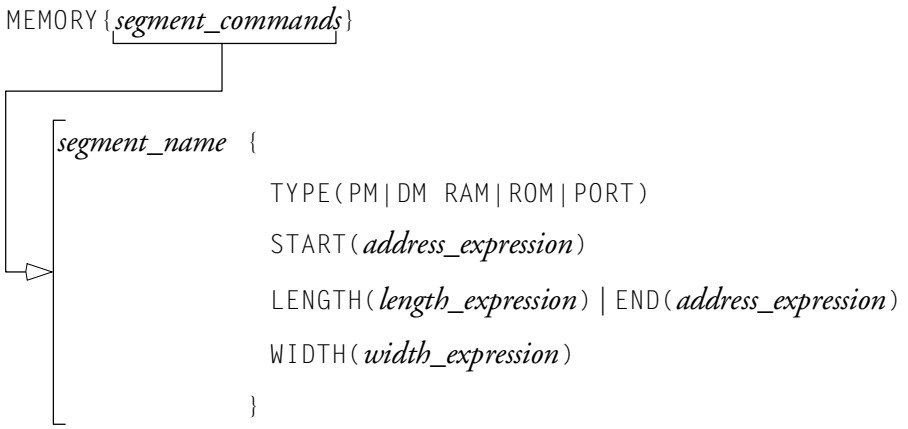

Figure 1-9. Syntax Tree of the MEMORY{} Command

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

- *segment_commands*

  Declares your target processor's memory segments. Although your LDF may contain only one MEMORY{} command that applies to each

scope of the LDF, there is no limit to the number of segments that you can declare within each `MEMORY{}` command.

Each segment declaration must contain a *`segment_name`*, a `TYPE`() command, a `START`() command, a `LENGTH`() or `END`() command, and a `WIDTH`() command.

- a *`segment_name`* command

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

- a segment `TYPE` command: `TYPE(string)`

  Identifies the architecture-specific type of memory within the segment. The linker stores this information in the executable for use by other development tools. The `TYPE` command identifies the memory's functional or hardware location: `RAM` or `ROM`.

- a `START(`*`address_expression`*`)` command

  Identifies the segment's start address. The *`address_expression`* must be an absolute address or an expression that evaluates to an absolute address.

- a `LENGTH(`*`length_expression`*`)` or `END(`*`address_expression`*`)` command

  Identifies the segment length in bytes or sets the segment's end address. When stating the length, *`length_expression`* must be the number of addressable words within the region or an expression that evaluates to the number of words. When stating the end

address, *address_expression* must be an absolute address or an expression that evaluates to an absolute address, such as START_1 + LENGTH_1 = END_1.

- a WIDTH(*width_expression*) command

  Identifies the bit width of the addressable memory words within the segment. The *width_expression* must be a number or an expression that evaluates to that number.

PROCESSOR{}

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

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

The syntax for the PROCESSOR{} command appears in Figure 1-10.

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

Figure 1-10. Syntax of the PROCESSOR{} Command

The PROCESSOR{} command syntax is defined as follows:

- *processor_name*

  Assigns a *processor_name* to the processor. Processor names follow the same rules as any linker label. For more information, see "LDF Expressions and Conventions" on page 1-32.

- OUTPUT(*file_name*.DXE)

  Selects the output file name for the executable (.DXE). Note that an OUTPUT() command in an LDF scope must appear before a SECTIONS{} command in that scope.

- MEMORY{*segment_commands*}

  Defines memory segments that apply only to this processor. Use LDF command scoping to define these segments outside the PROCESSOR{} command. For more information, see "Command Scoping" on page 1-31 and "MEMORY{}" on page 1-50.

- PLIT{*plit_commands*}

  Defines Procedure Linkage Table (PLIT) commands that apply only to this processor. For more information, see "PLIT{} Command" on page 1-86.

- SECTIONS{*section_commands*}

  Defines sections for placement within the executable (.DXE). For more information, see "SECTIONS{}" on page 1-56.

RESOLVE()

The RESOLVE(*symbol_name*, *resolver*) command directs the linker to resolve a particular symbol (variable or label) to an address using the resolver. The resolver is an absolute address or a file (.DXE or .SM) containing the definition of the symbol. If a linker does not find the symbol in the designated file, it issues an error.

When you resolve a C/C++ variable, prefix it with an underscore in the RESOLVE() statement (for example, _symbol_name).

Use the RESOLVE()command, which directs the linker to ignore a LINK_AGAINST() for a specific symbol, to override the search order for a specific variable or label. See Listing 1-10 on page 1-112 for more information.

SEARCH_DIR()

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

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

For example,

```
ARCHITECTURE(ADSP-21535)
MAP(SINGLE-PROCESSOR.MAP)          // Generate a MAP file

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

SECTIONS{}

The SECTIONS{} command specifies the placement of your program's SECTIONs in memory, using segments defined with the MEMORY{} command. The syntax for the SECTIONS{} command is shown in Figure 1-11.

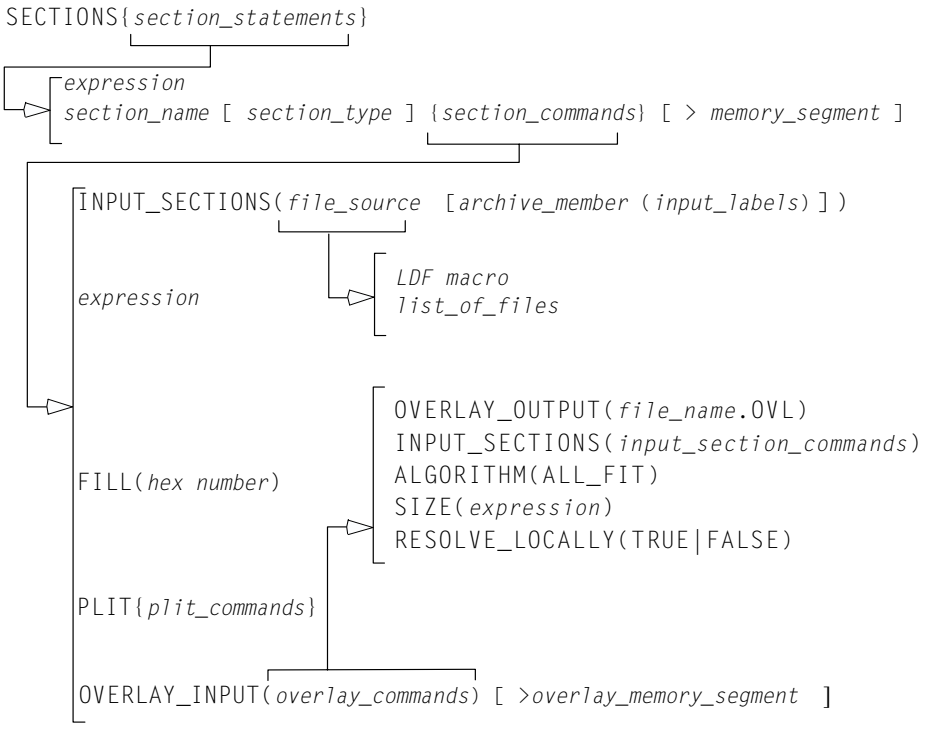

Figure 1-11. Syntax Tree of the SECTIONS{} Command

The LDF may contain a SECTIONS{} command within each PROCESSOR{} command. The SECTIONS{} command must be preceded by a MEMORY{} command, defining the memory segments in which the linker places the sections.

The following sections describe definitions for the parts of the SECTIONS{} command's syntax.

section_commands or expressions

This argument defines *expressions* or output sections (*section_name*). Use *expressions* to manipulate symbols or position the current location counter. Use output section commands to declare your program's sections. While your LDF may contain only one SECTIONS{} command within each LDF scope, there is no limit to the number of output sections that you can declare within each SECTIONS{} command. For more information, see "Command Scoping" on page 1-31.

section_name

The output section, *section_name* declaration, has the following syntax rules:

- A *section_name* starts with a letter, underscore, or period and may include any letters, underscores, digits, and points. A *section_name* must not conflict with any linker keywords. The special *section_name*.plit, indicates the Procedure Linkage Table (PLIT) section that the linker generates when resolving symbols in overlay memory. You must place this section in non-overlay memory to manage references to items in overlay memory.
- *section_type*

  The *section_type* is optional and assigns an ELF section type. The only legal section type keyword is SHT_NOBITS. 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 how to use SHT_NOBITS, see Listing 1-10 on page 1-112.

- *section_commands*

  The *section_commands* may contain any combination of the following commands: an INPUT_SECTIONS() command, an *expression*, a FILL() command, a PLIT{} command, or an OVERLAY_INPUT() command.

- *memory_segment*

  The *memory_segment* at the end of a section definition declares that the section is placed in the specified memory segment.

  The *memory_segment* is optional. Some sections, such as those for debugging, do not need to be included in the memory image of the executable, but are needed for other development tools that read the executable file. By omitting a memory segment assignment for a section, you direct the linker to generate the section in the executable, but prevent section content from appearing in the memory image of the executable.

INPUT_SECTIONS()

This part of the syntax in an *output_section_command* identifies the parts of your program to place in the executable with *input_section_commands*. When placing an input section, specify the *file_source*, *archive_member* (if the *file_source* is an archive), and *input_labels* of the sections.

An INPUT_SECTIONS() command has the following syntax rules:

- *file_source* may be a list of files or any LDF macro that expands into a file list, such as $COMMAND_LINE_OBJECTS. The list may contain object or archive files. Use commas to delimit files within the list.

- *archive_member* names the source-object file within an archive. The *archive_member* parameter and the left/right brackets, [ ], are only required if the *file_source* of the *input_label* is an archive.
- *input_labels* come from the run-time .SECTION names in your assembly program. Use commas to delimit .SECTION names within the list.

expression

In a *section_command*, the *expression* manipulates symbols or positions the current location counter, specified by a period (see "Location Counter (.)" on page 1-39). It is an assembly directive.

FILL(hex number)

In a *section_command*, the FILL is used to fill with hexadecimal numbers any gaps that you create by aligning or advancing the current location counter (see "Location Counter (.)" on page 1-39).

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

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

PLIT{plit_commands}

In a *section_command*, the PLIT command declares a locally[1] scoped Procedure Linkage Table (PLIT). It contain its own labels and expressions. For more information, see "PLIT{} Command" on page 1-86.

[1] In that section only.

OVERLAY_INPUT(overlay_commands)

In a *section_command*, the OVERLAY_INPUT() command identifies parts of the program to place in an overlay executable with *overlay_commands*. For more information on overlays, see "Memory Overlays and Overlay Memory Manager" on page 1-62 and "Linking for Overlay Memory Example" on page 1-123.

The *overlay_commands* part of the syntax must contain at least one of the following commands: INPUT_SECTIONS() command, OVERLAY_ID() command, NUMBER_OF_OVERLAYS() command, OVERLAY_OUTPUT() command, ALGORITHM() command, RESOLVE_LOCALLY() command, or SIZE() command.

The *overlay_memory_segment* determines whether the section is placed in an overlay segment and is optional. Some overlay sections, such as those loaded from a host, do not need to be included in the overlay memory image of the executable, but are needed for other development tools that read the executable file.

By omitting an overlay memory segment assignment for a section, you direct the linker to keep the section in the executable, but mark the section for exclusion from the overlay memory image of the executable.

An OVERLAY_INPUT() command supports the following syntax rules.

- The OVERLAY_OUTPUT() command directs the linker to output a overlay file (.OVL) for the overlay with the specified name. Note that a OVERLAY_OUTPUT() command in an OVERLAY_INPUT() command must appear before any INPUT_SECTIONS() for that overlay.
- The INPUT_SECTIONS() command has the same syntax within an OVERLAY_INPUT() command as when it appears within a *output_section_command*, except that you may not place the .PLIT section in overlay memory. For more information, see "INPUT_SECTIONS()" on page 1-58.

- The OVERLAY_ID() command directs the linker to return the overlay ID of the resolved symbol.
- The ALGORITHM() command directs the linker to use the specified overlay linking algorithm. The linker supports the ALL_FIT algorithm only. Therefore, 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
- The RESOLVE_LOCALLY() command, when applied to an overlay, controls whether the linker generates PLIT entries for function calls that are resolved within the overlay.

  For RESOLVE_LOCALLY(TRUE), the linker does not generate PLIT entries for locally resolved functions within the overlay. For RESOLVE_LOCALLY(FALSE), the linker generates PLIT entries for all functions, whether or not they are locally resolved within the overlay. The default is TRUE.
- The SIZE() command directs the linker to set an upper limit on the size of the memory that may be occupied by an overlay.

Advanced Linker Features and Commands

The Blackfin DSP linker's advanced features support linking executables for systems with overlay memory. This section discusses the concept of memory overlays and how they are used with Analog Devices DSPs, as well as the `OVERLAY_GROUP` and `PLIT` commands.

The VisualDSP++ 3.0 linker's advanced features also support linking executables for systems with multiprocessor and shared memory using `MULTIPROCESSOR`, `PACKING` and `SHARED_MEMORY` commands. However, currently these features are not supported by Blackfin DSP linker.

Memory Overlays and Overlay Memory Manager

In order to reduce DSP system costs, many applications use DSPs with smaller amounts of on-chip memory — placing much of the program code and data off chip. In order to run the applications efficiently, memory overlays are used.

This section discusses the concept of memory overlays and how they are used with Analog Devices DSPs, including the following topics and examples:

- "Memory Overlay Support" on page 1-66
- "Overlay Manager Example" on page 1-70
- "Reducing Overlay Manager Overhead" on page 1-78

All of the code segments used in the following discussion are parts of the two example programs that appear at the end of this section. Refer to "Linking for Overlay Memory Example" on page 1-123 for a complete code example of an LDF with overlay.

The Concept of Memory Overlays

Memory overlays provide support for applications whose entire program instructions do not fit in the internal memory of the processor. In such a case, program instructions are partitioned and stored in external memory until they are required for program execution. The partitions are referred to as *memory overlays* and the routines that call and execute them *overlay managers*.

Overlays are a "many to one" memory mapping system. Several overlays "live" (are stored) in unique locations in external memory, but "run" (or execute) in a common location in internal memory. Throughout this section, the storage location of overlays are referred to as the "live" location, and the internal location where instructions are executed are referred to as the "run" (run-time) space.

The overlay functions are written to *overlay files* (`*.OVL`) which can be used as one of linker's executable output files and can be read by the loader to generate a `.LDR` file.

Figure 1-12 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 segments reserved for execution of overlay program instructions.

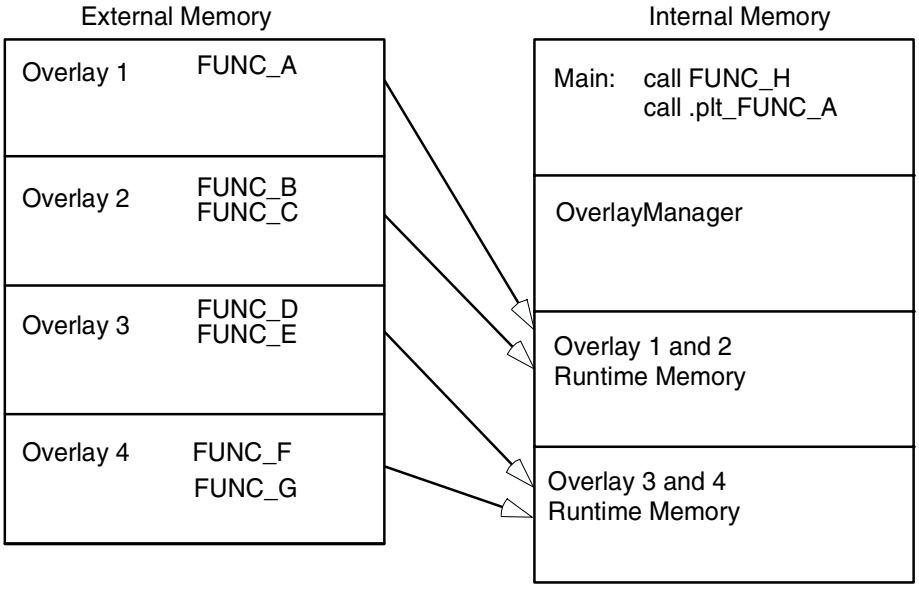

Figure 1-12. Memory Overlays

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

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

The Concept of Overlay Manager

The overlay manager is a user-defined routine that is responsible for loading a referenced overlay function or data buffer into internal memory (run time space). This is done with the aid of the linker-generated constants and the PLIT commands. The linker-generated constants tell the overlay manager the addresses of the live overlay, where the overlay resides for execution, and the number of words in the overlay. The PLIT commands tell the overlay manager such information as which overlay is required and the run time address of the referenced symbol.

The main objective of overlay managers is to transfer overlays to their run-time location when required. However, overlay managers may also be required to:

- Set up a stack to store register values.
- Check if 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 can begin immediately.
- Load an overlay while executing a function from a second overlay (or a non overlay function).

You may need your 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 the following:

- Specification of the live and run location of each overlay
- The generation of constants
- The redirection of overlay function calls to a jump table

The overlay support is partially designed by the user in the LDF. The user can specify which overlays share run-time memory and which memory segments establish the live and run space.

Listing 1-2 shows the section of an LDF defining two overlays. The overlay declaration configures two overlays to share a common run-time memory space. The `OVERLAY_INPUT` command syntax is described on page 1-60.

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

Listing 1-2. Overlay Declaration in LDF

```
.program
  {  OVERLAY_INPUT
      OVERLAY_OUTPUT(OVLY_one.ovl)
      INPUT_SECTIONS(FUNC_A.doj(program))
  }>ovl_code
  {  OVERLAY_INPUT
      OVERLAY_OUTPUT(OVLY_two.ovl)
      INPUT_SECTIONS(FUNC_B.doj(program) FUNC_C.doj(program))
  } >ovl_code
} >MEM_PROGRAM
```

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

The LDF tells the linker how to configure the overlays as well as provides the information necessary for the overlay manager routine to load the overlays. The information provided by the linker includes the following linker-generated overlay constants (where `N = Overlay ID`).

```
_ov_startaddress_N
_ov_endaddress_N
_ov_size_N
_ov_word_size_run_N
_ov_word_size_live_N
_ov_runtimestartaddress_N
```

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. Exception is the `_ov_size_N`, that specifies the total size in bytes.

The overlay live and run word sizes are different if the internal and external memory widths are different. A system containing 16-bit wide external memory may require data packing (depending upon the processor's external bus hardware support) to store an overlay containing instructions in an architecture using, for example, 24-bit instructions. The overlay live word size (number of words in the overlay) is based on the number of 16-bit words required to pack all of the 24-bit instructions.

The Blackfin DSP architecture supports byte addressing—it uses 16- or 32-bit opcodes. Therefore, no data packing is required.

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

The PLIT code is defined within the linker description file (LDF) by the programmer. This code prepares the overlay manager to handle the overlay containing the referenced symbol. The code generally 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 = [I0];
R1 = R0 * R2;
CALL FUNC_A;        /* Call to function in overlay */
[I3] = R1;
```

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

```
R0 = [I0];
R1 = R0 * R2;
CALL .plt_FUNC_A; / * Call to PLIT entry */
[I3] = R1;
```

The `.plt_FUNC_A` is the entry in the PLIT containing your 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.

Listing 1-3 is an example PLIT definition from an LDF, where the 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. (`PLIT_SYMBOL_OVERLAY_ID` and `PLIT_SYMBOL_ADDRESS` are linker keywords.). 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 overlay function called.

Listing 1-3. PLIT Definition in LDF

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

The linker expands the PLIT definition into individual entries in a table. An entry is created for each overlay symbol as shown in Listing 1-4. The redirect 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).

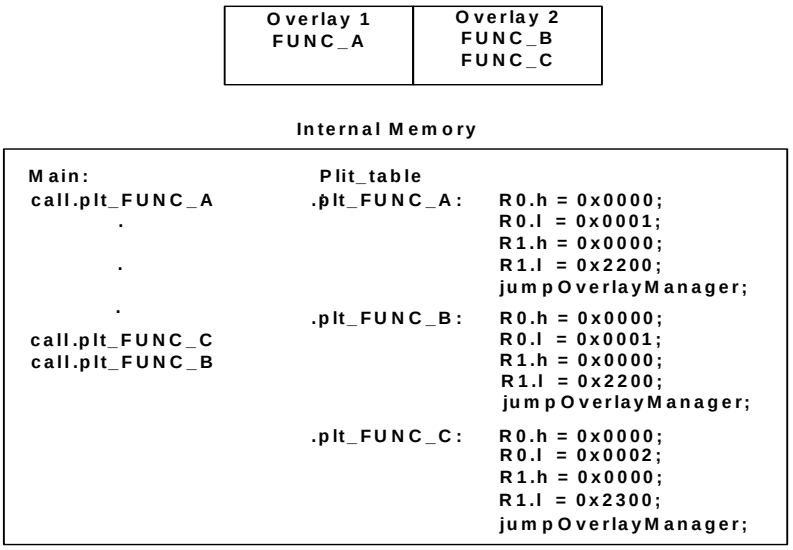

Figure 1-13. Expanded PLIT Table

For example, the first entry in the PLIT shown in Listing 1-4 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 subroutine 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 subroutine uses the overlay ID and run-time address to load and execute the referenced function.

Overlay Manager Example

The example in this section has two overlays, each of which contain 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 the sample overlay manager source code, see the examples that come with the development software. In the following example, the overlay manager:

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

Several code segments for the LDF and the Overlay Manager are displayed and explained in the text.

Listing 1-4. FFT Overlay Example 1

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

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

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

To verify whether an overlay is already in internal memory, place the overlay ID of each overlay already loaded and the overlay to be loaded in registers, for example `P0` and `P1`, and compare:

```
CC = p0 == p1; /* Is overlay already in internal memory? */
if CC jump skipped_DMA_setup;
                /* If so, bypass transferring it in.*/
```

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

The following code segment calls the four FFT functions.

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

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

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

Register `R0` contains the overlay ID that contains the referenced symbol and register `R1` contains the run-time address of the referenced symbol. The final instruction causes the program counter (PC) to jump to the starting address of the overlay manager function. The overlay manager routine uses the overlay ID in conjunction with the overlay constants gen-

erated 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 on R1.

The linker generates these constants used by the overlay manager:

```
.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_startaddress_1;
.EXTERN _ov_startaddress_2;
.EXTERN _ov_runtimestartaddress_1;
.EXTERN _ov_runtimestartaddress_2;
```

These constants provide the following information to the overlay manager.

- The overlays' sizes, in both run time word sizes and live word sizes
- the starting address of the live space
- the starting address of the run space.

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

```
.VAR liveAddresses[2] = _ov_startaddress_1,
                        _ov_startaddress_2;
.VAR runAddresses[2]  = _ov_runtimestartaddress_1,
                        _ov_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 any other registers consumed by the overlay manager.

The overlay manager also stores the ID of an overlay 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.

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 LDF.
   Set up DMA transfer to internal memory. Store values of
   registers used by the overlay manager on the stack. */
[sp++]=i1;
[sp++]=p0;
[sp+]=m3;
[sp++]=R2;

/* Use the overlay id as an index (must subtract one) */
R0=R0-1;   /* Overlay ID -1                          */
m3=R0;    /* Offset into the arrays containing linker
                defined overlay constants. */
p0.l = ov_id_loaded;
p0.h = ov_id_loaded;

R2=[p0];
CC=R0 ==R2;
if CC jump continue;
[p0]=m3;
```

```
R0=i0;        [sp++]=R0;
R0=m0;        [sp++]=R0;
R0=l0;        [sp++]=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.

```
// program sDMA (aka read channel) to read from the external
// memory; sDMA fills the FIFO;
// program pDMA (aka write channel) to write to internal
// memory, drains the FIFO.
// create the descriptor for the read:Live_DMA_Hdr.
P3.l = liveAddresses;
P3.h = liveAddresses;
P4.l = runAddresses;
P4.h = runAddresses;
//simplifying assumption: the transfer can be completed in
//one work unit.
P1.L = Live_DMA_Desc;   // address of descriptor for block to
                        // read
P1.H = Live_DMA_Desc;
P2.l = Run_DMA_Desc;    // address of descriptor for block to
                        // write
P2.H = Run_DMA_Desc;    // fill the descriptors
r1 = [P3];
[P1 + 4]=r1;            // start address for read
r1 = [P4];
[P2 + 4]=r1;            // start address for write
P3.l = liveWordSize;
P4.l = runWordSize;
r1 = W[P3];
W[P1+ 2]=r1;            // live mem size for read count
r1 = W[P4];
W[P2+ 2]=r1;           // run mem size for write count
                     // read channnel clear fifo & disable DMA
P3.L = MDR_DCFG & 0XFFFF;
P3.H = (MDR_DCFG >> 16) & 0XFFFF;
R1 = W[P3];
```

```
BITSET(R1,DMA_BUFCLR);
BITCLR(R1,DMA_EN);
W[P3] = R1;

                    // Write channel clear fifo and disable DMA
P3.L = MDW_DCFG & 0XFFFF;
R1 = W[P3];
BITSET(R1,DMA_BUFCLR);
BITCLR(R1,DMA_EN);
W[P3] = R1;
//kick off DMA
R1 = P1;
P3.L = MDR_DND & 0XFFFF;
W[P3] = R1.L;             // Write 16 LSBs of read header
                          // descriptor block address to DMA
                         // current descriptor pointer register
P3.L = DB_NDBP & 0xFFFF;
W[P3] = R1.H;             // Write 16 MSBs of read header
                          // descriptor block address to DMA
                           // next descriptor base pointer
register
R1 = P2;
P3.L = MDW_DND & 0XFFFF;
W[P3] = R1.L;             // Write 16 LSBs of write header
                          // descriptor block address to DMA
                         // current descriptor pointer register
P3.L = MDR_DCFG & 0XFFFF;
R1 = W[P3];
BITSET(R1,DMA_EN);
W[P3] = R1;                // Enable read DMA

P3.L = MDW_DCFG & 0XFFFF;
R1 = W[P3];
BITSET(R1,DMA_EN);
W[P3] = R1;                 // Enable write DMA

Wait_for_DONE:
R1 = W[P2];                // Read config word of the write
                            // block
```

```
cc = bittst(R1,15);       // Poll ownership bit 15 of DMA
                          // Config word to see if DMA is done
IF cc JUMP Wait_for_DONE;
cc = bittst(R1,14);        // Check completion status
IF cc JUMP OvlMgr_ERROR;  // Go to my error rtn if it did
                           // not work
```

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

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

In summary, the overlay manager routine does the following:

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

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

Reducing Overlay Manager Overhead

The example in this section incorporates the ability to transfer one overlay to internal memory while the core executes a function from another overlay. Instead of the core sitting idle while the overlay DMA transfer occurs, the core enables the DMA, 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 software-provided example contains deterministic overlay function calls. The time of overlay function execution requests are known as are the number of cycles required to transfer an overlay. Therefore, an overlay function load request can be placed such that the transfer is complete by the time the execution request is made. The next overlay transfer (from a load request) can be enabled by the core and the core can execute the instructions leading up to the function execution request.

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

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

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

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

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

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

The next set of instructions represents a load function call.

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

The implementation executes the first function and transfers the second function and so on. In this implementation, each function resides in a unique overlay and requires reserving two run-time locations; while one overlay is loading into one run-time location, a second overlay function is executing in another run-time location.

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

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

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

```
INPUT_SECTIONS(ovly_mgr.doj(program))

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

The first and third overlays share one run-time location and the second and fourth overlays share the second run-time location. By placing an input section between overlay declarations, multiple run-time locations are allocated.

Additional instructions are included to determine if the function call is a load or an execution call. If the function call is a load, the overlay manager initiates the DMA transfer, then jumps the PC back to the location where the call was made. If the call is an execution call the overlay manager determines if 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 to see if the requested overlay is currently in internal memory. If not, it sets up the DMA transfer. It then checks to see if 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. These modifications may allow for the elimination of some instructions. For instance, if your application allows for the free use of registers, you may not need a run-time stack.

OVERLAY_GROUP{} Command

Memory overlays provide support for applications whose entire 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. See the `OVERLAY_INPUT` command description on page 1-60. Refer to "Memory Overlays and Overlay Memory Manager" on page 1-62 for detailed description of overlay functionality.

The `OVERLAY_GROUP` command allows you to group overlays, so that 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 1-14 demonstrates grouped overlays concept.

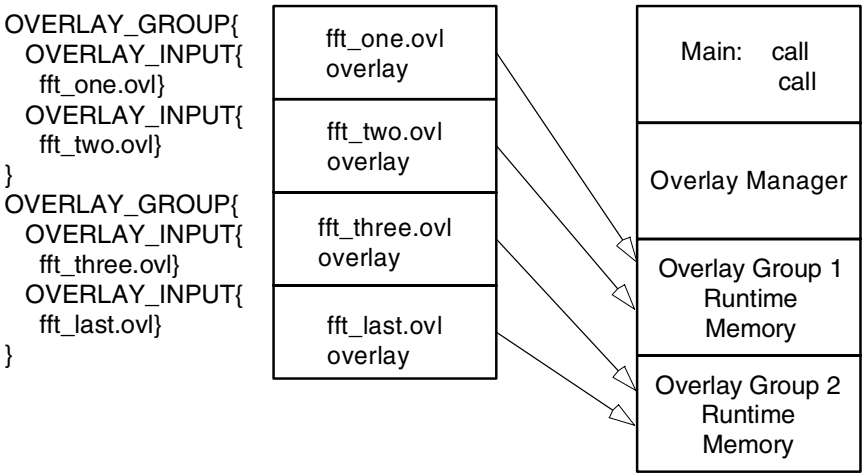

Figure 1-14. Overlays, Grouped

In the simplified examples in Listing 1-5 and Listing 1-6, the functions are written to *overlay files* (`*.OVL`). It does not matter (except to a DMA transfer that brings them in) whether these are disk files or memory segments. The overlays are active only when they are executed in run-time memory, all of which is located in segment `program`.

Ungrouped Overlay Execution

In Listing 1-5, as the FFT progresses and the overlay functions are called in turn, they are brought into run-time memory in sequence: four function transfers.

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

Listing 1-5. LDF Overlays, Not Grouped

```
// This listing is part of the SECTIONS command for processor P0.
// Declare which functions reside in which overlay. The over-
// lays have been split up either into different segments in
// one file, or into different files.

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

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

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

OVERLAY_INPUT {
   ALGORITHM(ALL_FIT)
   OVERLAY_OUTPUT(fft_last.ovl)
   INPUT_SECTIONS(  Fft_4th.doj(program) ) } >ovl_code
   INPUT_SECTIONS(  Fft_last.doj(program) ) } >ovl_code
```

Grouped Overlay Execution

Listing 1-6 shows a different implementation of the same algorithm. The overlaid functions are grouped in pairs. Since each pair of the four routines is resident simultaneously, the processor can execute both routines before paging.

Listing 1-6. LDF Overlays, Grouped

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

PLIT{} Command

The linker resolves function calls and variable accesses, both direct and indirect, across overlays. This support requires that the linker generate extra code in order 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 linker's `PLIT{}` (*Procedure Linkage Table)* commands in your LDF allow you to insert assembly instructions that handle calls to functions in overlays.The assembly commands are specific to an overlay and are executed each time a call to a function in that overlay is detected.

The `PLIT{}` commands provide a template from which the linker generates assembly code whenever a symbol resolves to a function in overlay memory. These instructions typically handle a call to a function in overlay memory by calling an overlay memory manager. Refer to "Memory Overlays and Overlay Memory Manager" on page 1-62 for 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 1-91.

PLIT Syntax

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.

Figure 1-15 shows the general syntax of the `PLIT{}` command indicating how the linker handles a symbol (`symbol`) local to an overlay function.

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

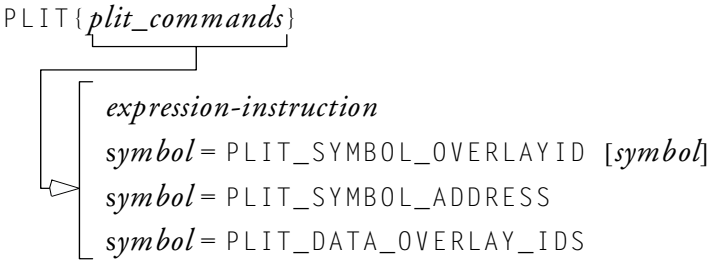

Figure 1-15. Syntax Tree of the PLIT{} Command

After evaluation, the linker puts the returned bytes in the `.PLIT` output section. It also manages addressing in that output section.

plit_commands include *instructions* that are assembly instructions or expressions. There may be none, one, or more *instructions*. They may occur in any reasonable order in the command structure, may precede or may follow the symbols discussed in the next few paragraphs.

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

The following two `PLIT_SYMBOL` constants contain information about *symbol* and the overlay where it occurs. You must supply instructions to handle that information.

- The `PLIT_SYMBOL_OVERLAYID` command directs the linker to return the overlay ID of the resolved symbol.
  symbol_1 is a register name.
- The `PLIT_SYMBOL_ADDRESS` command directs the linker to return the absolute address of the resolved symbol in run-time memory.
  symbol_2 is a register name.

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

The PLIT_DATA_OVERLAY_ID command directs the linker to return the address of an array containing the IDs of overlays that hold data used by the resolved symbol's function. The array terminates with the null ID "0". symbol_n is typically a register name or memory location, which is loaded with that (start) address.

After the setup and variable identification are completed, the overlay itself must be brought (via DMA transfer) into run-time memory. That happens under the control of a piece of assembly code called the overlay manager.

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

Allocating Space for PLITs

The LDF must allocate space in memory to hold any PLITs your linker builds. Typically, that memory resides in the program code (program[1]) Memory section. A typical LDF declaration for that purpose appears below.

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

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 is allocating space for linker-generated routines, not containing any of your (input) data objects.

[1] Whatever you name your program code Memory Segment.

- 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 LPLIT command implementation.

- Simple PLIT - no state saved
- A PLIT that saves register contents

Simple PLIT – No State Saved
A simple PLIT merely copies the symbol's address and overlay ID into registers, and jumps to the overlay manager, as shown in Listing 1-7. This fragment was extracted from the global scope (just after the `MEMORY{}` command) of sample `fft_group.ldf`.

Listing 1-7. A Simple PLIT{} Command

```
/* 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 runtime address of the
symbol called. Be sure the registers used in the plit do not con-
tain values which cannot be overwritten. */

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

In this case, you are responsible for verifying the contents of `R0` and `R1` are either safe or irrelevant.

A PLIT that Saves Register Contents

The `PLIT{}` command in the code fragment below saves the contents of `R0` and `R1` in data memory before being overwritten.

```
PLIT{
p0.l = save_r0;         // p0 points to memory mapped
p0.h = save_r0;         // save_r0 variable
[p0] = r0;              // save r0 to "save_r0" variable
p0.l = save_r1;         // p0 points to memory mapped
p0.h = save_r1;         // save_r1 variable
[p0] = r1;              // save r1 to "save_r1" variable
r0 = PLIT_SYMBOL_OVERLAYID;
r1 = PLIT_SYMBOL_ADDRESS;
jump _OverlayManager;
}
```

Note that the `p0` register gets trashed in this example, so it would need to be saved before calling the overlay function.

As a general case, you want to minimize the overlay transfer traffic. Designing your code so overlay functions are imported and used with minimal (perhaps zero) reloading has a performance payoff.

What PLIT Does – Summary

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

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

The overlay manager can also:

- Helped by user's 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 implemented this feature within the PLIT section of your LDF.

  Note: Your program may not need this information saved.

- 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 the overlay function is already in internal memory, and avoid reloading it.

Using PLIT and Overlay Manager

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

Refer to "PLIT{} Command" on page 1-86 for basic syntax information. Refer to "Memory Overlays and Overlay Memory Manager" on page 1-62 for detailed information on overlays.

Figure 1-16 shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker generates some special symbols for overlays. These overlay symbols are:

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

The # in these symbols indicates the overlay number.

Note that the overlay number starts from 1, not 0. This is done to avoid confusion when placing these elements into an array or buffer which will be used by an overlay manager.

The two functions, in Figure 1-16, are on different overlays. By default, the linker generates PLIT code only when an unresolved function reference is resolved to a function definition in overlay memory.

Code in Non-Overlay Memory

```
main()
  {
   int (*pf)() = X;
   Y();
  }
/* PLIT & software-manager handle calls,
using the PLIT for resolving calls and
loading overlays as needed */

.plt_X:          // setup OM information
      call OM
.plt_Y:          // setup OM information
      call OM

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

Overlay 1

```
X() {...}      // function X defined
```

Overlay 2

```
Y() { ... }    // function Y defined
```

Runtime Overlay Memory

```
               // currently loaded overlay
```

Figure 1-16. PLITs & Overlay Memory; main() Calls to Overlays

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

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

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

PLITs allows you to resolve interoverlay calls, as shown in Figure 1-17. You should structure your LDF in such a way the PLIT code that the linker generates for inter-overlay function references is part of the `.plit` section for main(), which is stored in non-overlay memory.

The `.plit` section should always be stored in non-overlay memory.

The linker resolves all references to variables in overlays, and the PLIT lets an overlay manager handle the overhead of loading and unloading overlays. One way to optimize overlays is to not put global variables in overlays. This avoids the difficulty of making sure the proper overlay is loaded before a global gets called.

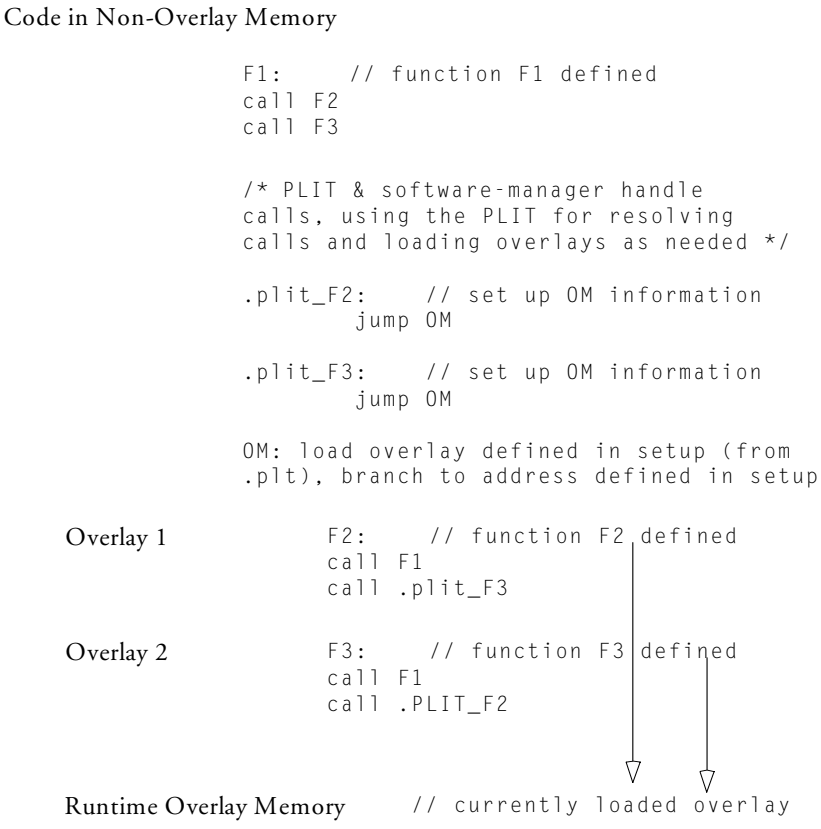

Figure 1-17. PLITs and Overlay Memory; Inter-Overlay Calls

Linker Command-Line Reference

This section provides reference information on the linker command line and linking, including:

- "Command-Line Syntax"
- "Linker Command-Line Switch Summary" on page 1-100
- "Command-Line Switch Descriptions" on page 1-102

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

When you use the linker via VisualDSP++, the settings in the **Link** property page correspond to linker's command-line switches. The VisualDSP++ calls the linker with those switches when you link your code. For more information, use see the *VisualDSP++ 3.0 User's Manual for Blackfin DSP* and VisualDSP++ online Help.

Command-Line Syntax

You can run the linker using one of the following normalized formats for the linker command line.

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

The linker (the command itself) and either `-Darchitecture` or `-T<ldf name>` must be provided for the link to proceed. The LDF specified following the `-T` switch must contain an `-Darchitecture` command if the command line does not have `-Darchitecture`.

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

For example,

```
linker -DADSP-21535 p0.doj p1.doj -T target.ldf -t -o program.dxe
```

When using the linker's command line, make sure you are familiar with the following topics:

- "Object Files in the Linker Command Line"
- "Switch Format in the Linker Command Line" on page 1-97
- "File Names on the Linker Command Line" on page 1-98

Analog Devices suggest that you use `-proc processorID` instead of `-Darchitecture` on the command line to make the target processor selection. See Table 1-6 on page 1-100 for more information.

Object Files in the Linker Command Line

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

- The standard object file is produced by the assembler and has a `.DOJ` extension.
- The command line may list archives (libraries) each of one or more files, with a `.DLB` extension. Examples include C run-time and math libraries delivered with VisualDSP++. Developers may create archives of common or specialized objects. Special libraries may be obtained from DSP algorithm vendors.
- It may also be an executable (`.DXE`) file *to be linked against*[1].

Object file names are not case-sensitive. But linker switches are case sensitive. For example, `linker -t` is not the same as `linker -T`.

[1] "Link Against" is described on page 1-41, under $COMMAND_LINE_LINK_AGAINST

An `object` filename has the following characteristics:

- It can include the drive, directory path, file name, and file extension
- Its path may be absolute or relative to the directory where the linker is invoked
- It should enclose long file names within straight-quotes.

If the file exists before the link starts, the linker opens it and verifies its type before processing the file. If the file is created during the link, the linker uses the file's extension to determine the type of file to create.

Table 1-2 on page 1-15 lists the valid extensions and matching linker operations.

Switch Format in the Linker Command Line

The linker has many optional switches that can be used to select the operations and modes for the compiler and other tools. The standard linker switch syntax is as follows:

`-switch [`*`argument`*`]` — name of the switch to be processed, plus its parameters (if any). Different switches require (or prohibit) white space between the switch and its parameter.

As noted above, the linker command line (except for filenames) is case sensitive. For example, the command line:

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

calls the linker as described below.

Note the difference between the `-T` and `-t` switches:

- `p0.doj, p1.doj` and `p2.doj` — Links object files together into an executable.
- `(-T target.ldf)` — Uses the LDF listed 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` — Names the linked, executable output file.

File Names on the Linker Command Line

Many linker switches take a file name as an optional parameter. Table 1-5 on page 1-99 lists the types of files, names, and extensions that the linker expects on filename arguments.

The linker supports relative and absolute directory names, default directories, and user-selected directories for file search paths. File searches occur as follows:

- Specified path — If you include relative or absolute path information on the command line, the linker searches in that location for the file.
- *User -selected directories* — If you do not include path information on the command line and the file is not in a default directory, the linker searches for the file in search directories that you select with the `-L` (path) command-line switch and `SEARCH_DIR` commands in the LDF. The linker searches these directories in the order that they appear on the command line or in the LDF.
- *Default directory* — If you do not include path information in the LDF named in the `-T` switch, the linker searches for the LDF named in the current working directory. If you use a default LDF

by omitting any LDF information in the command line and instead specifying -D*architecture*, the linker searches in the processor-specific `LDF` directory; for example, `...\$ADI_DSP\Blackfin\ldf`.

For more information on file searches, see "LDF Macros" on page 1-40.

When you provide an input or output file name as a command line parameter, use the following guidelines:

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

The linker follows the conventions for file name extensions that appear in Table 1-5

Table 1-5. File Name Extension Conventions

Extension	File Description
`.dlb`	Library (archive) file
`.doj`	Object file
`.dxe`	Executable file
`.ldf`	Linker description file
`.ovl`	Overlay file

Linker Command-Line Switch Summary

This section describes the linker command-line switches. A list of all linker's command-line switches appears in Table 1-6. Refer to "Command-Line Switch Descriptions" on page 1-102 for full description of each switch.

A brief description of each switch includes information on case sensitivity, equivalent switches, switches overridden/contradicted by the one described, and naming and spacing constraints on parameters:

- Switches may be used in any order on the command line. Items shown in `[ ]` are optional; items in italics are user-defined 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.

Table 1-6. Linker Command-Line Switches

Switch	Description
object(s) on page 1-102	Specifies object files involved in the linking; process files that are not parameters to a switch.
<null> on page 1-103	Displays a summary of command line options and exit.
@ *file* on page 1-103	Directs the linker to use the specified file as input on the command line.
-Darchitecture on page 1-103	Specifies the target architecture (processor).
-L *path*on page 1-103	Adds the path name to search libraries for objects.
-M on page 1-104	Produces dependencies.
-MM on page 1-104	Builds and produces dependencies.

Table 1-6. Linker Command-Line Switches (Cont'd)

Switch	Description
`-Map file` on page 1-104	Outputs a map of link symbol information to a file.
`-MDmacro[=def]` on page 1-104	Defines and assigns value `def` to a preprocessor macro.
`-S` on page 1-104	Omits debugging symbol information from the output file.
`-T filename` on page 1-104	Names the LDF.
`-e` on page 1-105	Directs the linker to eliminate unused symbols from the executable.
`-es secName` on page 1-105	Names sections (`secName` list) to which elimination algorithm is being applied.
`-ev` on page 1-105	Eliminates unused symbols verbosely.
`-h|help` on page 1-105	Outputs the list of command-line switches and exits.
`-i path` on page 1-105	Includes search directory for preprocessor `include` files.
`-ip` on page 1-106	Fills in fragmented memory with individual data objects that fit. Also requires objects to have been assembled using the assembler's `-ip` switch.
`-jcs2l` on page 1-106	Directs the linker to convert out-of-range short calls and jumps to the longer form.
`-jcs2l+` on page 1-106	Enables `-jcs2l` and allows the linker to convert out-of-range branches to indirect calls/jumps sequences.
`-keep symName` on page 1-107	Retains unused symbols.
`-o filename` on page 1-107	Outputs the named executable file.
`-pp` on page 1-107	Stops after preprocessing.
`-proc ProcessorID` on page 1-107	Directs the linker to select a target processor.
`-s` on page 1-107	Strips symbol information from the output file.
`-sp` on page 1-108	Skips preprocessing.

Table 1-6. Linker Command-Line Switches (Cont'd)

Switch	Description
-t on page 1-108	Directs the linker to output the names of link objects.
-v on page 1-108	Verbose: directs the linker to output status information.
-versionon page 1-108	Directs the linker to output its version and exit.
-warnonce on page 1-108	Warns only once for each undefined symbol.
-xref *filename* on page 1-108	Outputs a list of all cross-referenced symbols.

Command-Line Switch Descriptions

objects

When naming or specifying the files (or *objects*) that are not parameters to a switch, the linker uses a file's type to determine how to handle it. The linker gets a file's type as follows:

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

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 linker macro (see on page 1-41). For information on executables, see the $COMMAND_LINE_LINK_AGAINST linker macro (see on page 1-41).

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

<null>

Displays a summary of command line options and exit. Same as `linker -help`.

@ file

Uses *`file`* as input to the linker command line. This switch allows you circumvent environmental command line length restrictions. The *`file`* may not start with "`linker`" (it can't be a linker command line). Any white space in *`file`* serves to separate tokens, including "newline".

-Darchitecture

Specifies target architecture/processor. For example, `-DADSP-21535` or `-DBlackfin`. No white space is permitted between `-D` and *`architecture`*. The "architecture" entry is case-sensitive, and must be available in your VisualDSP++ installation. This option must be used if no LDF is specified on the command line (see `-T` option). It also must be used if the specified LDF does not specify `ARCHITECTURE()`. Architectural inconsistency between this option and LDF causes an error.

It is preferable to use `-proc` *`processorID`* instead of `-D`*`architecture`* on the command line to make the target processor selection.

-L *path*

Adds path name to search libraries and objects. Not case-sensitive; spacing is unimportant. The *`path`* parameter enables searching for any file, including the LDF itself. May be repeated to add multiple search paths. Paths named in this command are searched before the 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 also to perform the build.

-Map *file*

Outputs a map of link symbol information to a *`file`*, which can have any name. The *`file`* parameter is obligatory and has a `.MAP` extension, provided by the Linker. The white space is obligatory before *`file`*. Otherwise, the link fails.

-MD*macro*[=*def*]

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

-S

Omits debugging symbol information (*not* all symbol information) from the output file. Compare with `-s` switch, on page 1-107.

-T *file*

Uses *`file`* to name an LDF.
The LDF specified following the `-T` switch must contain an `ARCHITECTURE()` command if the command line does not have `-Darchitecture`.

The linker "requires" the `-T` switch when linking for a processor for which no IDDE support has been installed (e.g., the processor ID does not show up in the **Target processor** field of the **Project Options** dialog box.)

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

-e

Eliminates unused symbols from the executable.

-es *secName*

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

-ev

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

-h | -help

Displays a summary of command-line options and exit.

-i *path*

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

-ip

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

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

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

The `-noip` option (in assembler) turns off the individual placement option. See the *VDSP++ 3.0 Assembler and Preprocessor Manual for Blackfin DSPs*.

-jcs2l

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

-jcs2l+

Enables the `-jcs2l` switch and allows the linker to convert out-of-range branches (`-0x800000` to `0x7FFFFF`) to indirect calls/jumps sequences using the `p1` register. This option can be useful for automatically expending jumps from L1 to L2 memory (or vice versa) on Blackfin DSP architecture. Refer to **Branch expansion instruction** on the **Link** property page.

-keep *symName*

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

-o *filename*

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

-pp

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

-proc *ProcessorID*

Directs the linker to select a target processor.

- To select ADSP-21535 DSP, enter `-proc` *`ADSP-21535`*
- To select ADSP-21532 DSP, enter `-proc` *`ADSP-21532`*

-s

`S`trips all symbols—directs the linker to omit all symbol information from the output file.

Some debugger functionality, including "run to main", all `stdio` functions, and the ability to stop at the end of program execution all rely on the debugger being able to find certain symbols in the executable. These symbols are also removed when using this switch.

-sp

Skips preprocessing—links without preprocessing the LDF.

-t

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

-v

Verbose — outputs status information while linking.

-version

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

-warnonce

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

-xref *filename*

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

LDF Programming Examples

This section shows several typical LDFs. As you modify these examples, refer to the syntax descriptions in "LDF Command Summary" on page 1-44.

This section provides the following examples:

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

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

A variety of per-processor default LDF files come with the development software, providing an example LDF for each processor's internal memory architecture. These default LDFs are in the directory:

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

Linking for Single-Processor 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 1-8 shows a single-processor LDF. Note the following commands in this LDF:

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

Listing 1-8. Single-Processor System LDF Example

```
ARCHITECTURE(ADSP-21535)

SEARCH_DIR( $ADI_DSP\Blackfin\lib )

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

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

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

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

$OBJECTS = $COMMAND_LINE_OBJECTS;

//    A linker project to generate a DXE file

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

   MEMORY            // Processor specific memory command
   { INCLUDE("21535_memory.ldf") }

   SECTIONS          // Specify the Output Sections
   {
       INCLUDE( "21535_sections.ldf" )
   }                       // end P0 sections
}                          // end P0 processor
```

Linking Large Uninitialized Variables

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

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers this can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur large download times when using with the emulator. This also may occur when booting from a loader file (because of the increased file size). Listing 1-10 shows an example using the `SHT_NOBITS` section to avoid initialization of a segment.

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

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

Listing 1-9. Large Uninitialized Variables: Assembly Source

```
.Section   extram_area;   /* 1Mx8 EXTRAM */
.byte huge_buffer[0x006000];
```

Listing 1-10. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-Blackfin)
$OBJECTS = $COMMAND_LINE_OBJECTS; // Libraries & objects from
                                  // the command line
MEMORY {
   mem_extram {
      TYPE(RAM) START(0x10000) END(0x15fff) WIDTH(8)
      } // end segment
   } // end memory

PROCESSOR P0 {
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
         // SHT_NOBITS section isn't written to the output file
   SECTION {
     extram_ouput SHT_NOBITS {
       INPUT_SECTIONS( $OBJECTS ( extram_area ) )} >mem_extram;
      }  //end section
   }     // end processor P0
```

Linking for Assembly Source File

Listing 1-12 shows an example LDF (for ADSP-21535 DSP) that describes a simple memory placement of an assembly source file (Listing 1-11) which contains code and data that is to reside in, and execute from, L2 SRAM. This example assumes that the code and data declared in L2 memory is cacheable within L1 code and data memories. The LDF file includes two commands, `MEMORY` and `SECTIONS`, which are used to describe specific memory and system information. Refer to Notes for Listing 1-1 on page 1-20 for information on items in this basic example.

Listing 1-11. MyFile.ASM

```
.section program;
.global main;
main:

    p0.l = (myArray & 0xffff);
    p0.h = (myArray >> 16);
    r0 = [p0++];
    …

.section data1;
.global myArray;
var myArray[256] = "myArray.dat";
```

Listing 1-12. Simple LDF Based on Assembly Source File Only

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

// Declare specific DSP Architecture here (for linker)
ARCHITECTURE(ADSP-21535)
// LDF macro which equals all object files in project command
line
$OBJECTS = $COMMAND_LINE_OBJECTS;

  // Describe the physical system memory below
```

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

PROCESSOR p0{
  OUTPUT($COMMAND_LINE_OUTPUT_FILE)

  SECTIONS{
    DXE_L2SRAM{
      // Align L2 instruction segments on a 2-byte boundaries
      INPUT_SECTION_ALIGN(2)
      INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
      // Align L2 data segments on 1-byte boundary
      INPUT_SECTION_ALIGN(I
      INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
    >L2SRAM
// end SECTIONS
// end PROCESSOR p0
```

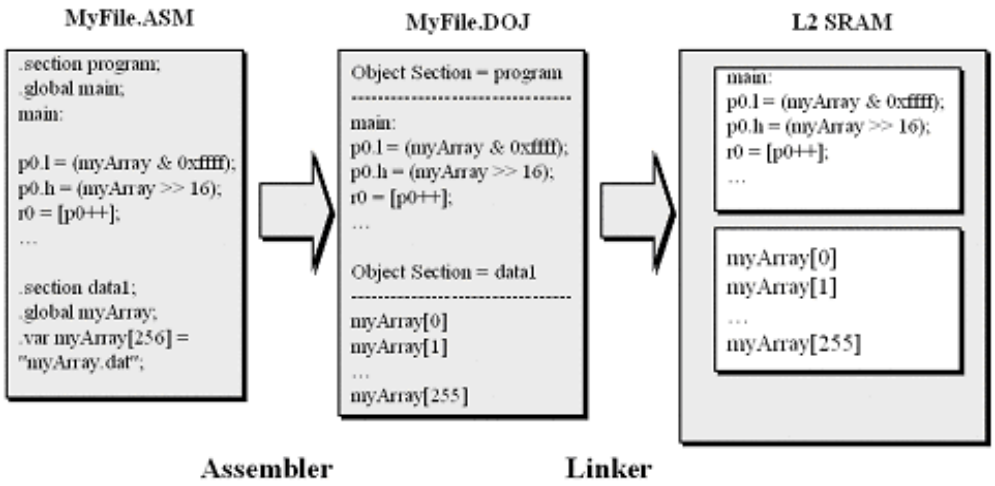

Figure 1-18. Assembly-to-Memory Code Placement

Linking for C Source File – Example 1

Listing 1-14 shows an example LDF that describes the memory placement of a simple C source file (Listing 1-13) which contains code and data that is to reside in, and execute from, L2 SRAM. This example also assumes that the code and data declared in L2 memory is cacheable within L1 code and data memories. The LDF file includes two commands, `MEMORY` and `SECTIONS`, which are used to describe specific memory and system information. Refer to Notes for Listing 1-1 on page 1-20 for information on items in this basic example.

Listing 1-13. Simple C Source File Example 1

```
int myArray[256];

void main(void){
  int i;

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

}// end main()
```

Listing 1-14. Simple C-based LDF Example for ADSP-21535 DSP

```
ARCHITECTURE(ADSP-21535)
SEARCH_DIR( $ADI_DSP\Blackfin\lib )

#define LIBS libc.dlb, libevent.dlb, libsftflt.dlb,
libcpp_blkfn.dlb, libcpprt_blkfn.dlb, libdsp.dlb, idle.doj

$LIBRARIES = LIBS, librt.dlb;

$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY{
  // 248KB of L2 SRAM memory segment for user code and data
  MemL2SRAM{TYPE(RAM) START(0xf0000000) END(0xF003dfff) WIDTH(8)}
```

```
  // 4KB of L2 SRAM memory segment for C runtime stack (user mode)
  MemStack {TYPE(RAM) START(0xf003e000) END(0xf003efff) WIDTH(8)}
  // 4KB of L2 SRAM for stack memory segment (supervisor mode)
  MemSysStack {TYPE(RAM) START(0xf003f000) END(0xf003ffff)
  WIDTH(8)}
  // 4KB of Scratch SRAM for Heap memory segment
  MemHeap {TYPE(RAM) START(0xFFB00000) END(0xFFB00FFF) WIDTH(8)}
}

PROCESSOR p0{

  OUTPUT($COMMAND_LINE_OUTPUT_FILE)

  SECTIONS{}
  // Declare L2 Input objects below…
  DXE_L2_SRAM{
   // Align L2 instruction segments on a 2-byte boundaries
   INPUT_SECTION_ALIGN(2)
   INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
   // Align L2 data segments on a 1-byte boundary
   INPUT_SECTION_ALIGN(1)
   INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
   // Align L2 constructor data segments on a 1-byte boundary
   // (C++ only)
   INPUT_SECTION_ALIGN(1)
   INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
  }>MemL2SRAM

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

  // Allocate memory segment for system stack sysstack{
   // assign start address of sys stack to 'ldf_sysstack_space'
   // variable using the LDF's current location counter " ."
   ldf_sysstack_space = .
```

```
    // assign end address of stack to 'ldf_sysstack_end' variable
    ldf_sysstack_end = ldf_sysstack_space + MEMORY_SIZEOF
    (MemSysStack) - 4;
   }>MemSysStack

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

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

Linking for Complex C Source File – Example 2

Listing 1-3 shows an example LDF that describes the memory placement of a C source file, which contains code and data that is to reside in, and execute from, L1, L2, and Scratchpad SRAM, as well as external SDRAM Banks 0 through 3. The LDF file includes two commands, `MEMORY` and `SECTIONS`, which are used to describe specific memory and system information. Refer to Notes for Listing 1-1 on page 1-20 for information on items in this complex example.

Listing 1-15. Complex C Source File Example

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

...
...

}

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

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

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

int i;
x = 0x5555;

...
}
```

The following is an example of an LDF file (for ADSP-21535 DSP) which is based on the complex C source from Listing 1-13.

Listing 1-16. C LDF File Example - SDRAM.LDF

```
ARCHITECTURE(ADSP-21535)
SEARCH_DIR($ADI_DSP\Blackfin\lib
#define OMEGA idle.doj
#define LIB1 libc.dlb, libevent.dlb, libsftflt.dlb
#define LIB2 libcpp_blkfn.dlb, libcpprt_blkfn.dlb, libdsp.dlb

$LIBRARIES = LIB1, LIB2, OMEGA, librt.dlb;
$OBJECTS = $COMMAND_LINE_OBJECTS;

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

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

 // 248KB of user code and data in L2 SRAM segment
 Mem_L2_SRAM{TYPE(RAM) START(0xF0000000) END(0xF003DFFF) WIDTH(8)}
 // 4KB for heap segment in L2 SRAM (for dynamic memory allocation)
 Mem_Heap{TYPE(RAM) START(0xF003E000) END(0xF003EFFF) WIDTH(8)}

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

 // 4 x 128MB External SDRAM memory segments
 Mem_SDRAM_Bank0{TYPE(RAM) START(0x00000000) END(0x07FFFFFF)
 WIDTH(8)}
 Mem_SDRAM_Bank1{TYPE(RAM) START(0x08000000) END(0x0FFFFFFF
```

```
 WIDTH(8)}
 Mem_SDRAM_Bank2{TYPE(RAM) START(0x10000000) END(0x17FFFFFF)
 WIDTH(8)}
 Mem_SDRAM_Bank3{TYPE(RAM) START(0x18000000) END(0x1FFFFFFF)
 WIDTH(8)}
}// end MEMORY{}

PROCESSOR p0{
 OUTPUT($COMMAND_LINE_OUTPUT_FILE)

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

  // Input section declarations for L1 data bank A memory
  DXE_L1_DataA_SRAM{
   // Align L1 data segments on a 1-byte boundary
   INPUT_SECTION_ALIGN(1)
   INPUT_SECTIONS($OBJECTS(Data_BankA)
  }>Mem_L1_BankA_SRAM

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

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

  sysstack{
    ldf_sysstack_space = .;
```

```
  ldf_sysstack_end = ldf_sysstack_space +
  MEMORY_SIZEOF(Mem_SysStack) - 4;
}>Mem_SysStack

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

DXE_L2_SRAM{
 // Align L2 code segments on a 2-byte boundary
 INPUT_SECTION_ALIGN(2)
 INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
 // Align L2 data segments on a 1-byte boundary
 INPUT_SECTION_ALIGN(1)
 INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1))
 // Align L2 constructor data segments on a 1-byte boundary
 // (C++ only)
 INPUT_SECTION_ALIGN(1)
 INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
}>Mem_L2_SRAM

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

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

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

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

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

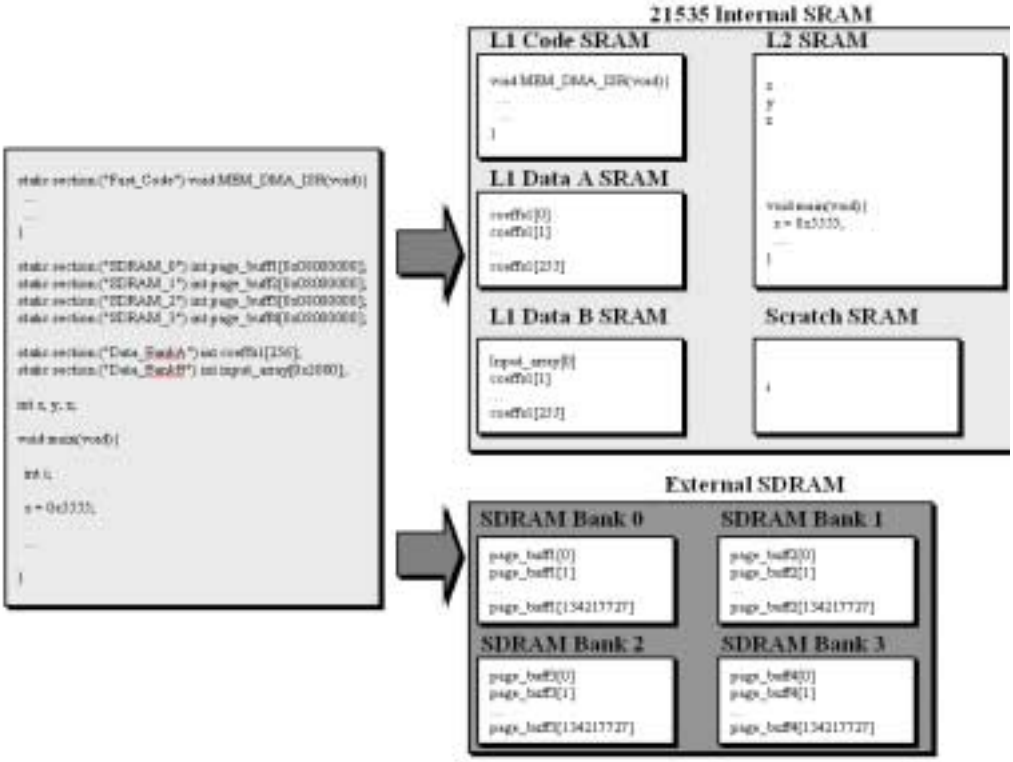

Figure 1-19. C-to-Memory Code Placement

Linking for Overlay Memory Example

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

Listing 1-17 shows an example overlay memory LDF. For more information on this LDF, see the comments in the listing.

Listing 1-17. Overlay-Memory System LDF Example

```
ARCHITECTURE(ADSP-21535)
SEARCH_DIR( $ADI_DSP\Blackfin\lib )

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

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

PROCESSOR p0
```

```
{
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
   SECTIONS
   {
      dxe_reset { INPUT_SECTIONS($OBJECTS(IVreset))
      } >MEM_PROGRAM
      dxe_itab  { INPUT_SECTIONS($OBJECTS(IVpwrdwn))

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

PLIT {
   R0.h = PLIT_SYMBOL_OVERLAYID;
      // overlay ID of the resolved symbol
   R1.h = PLIT_SYMBOL_ADDRESS;
      //run address of the resolved symbol
   R0.l = PLIT_SYMBOL_OVERLAYID;
      // overlay ID of the resolved symbol
   R1.l = PLIT_SYMBOL_ADDRESS;
      //run address of the resolved symbol
   p0.h=_overlayID;
   p0.l=_overlayID;
   [p0] = R0;
   p0.h=_pf;
   p0.l=_pf;
   [p0] = R1;
   JUMP _OverlayManager;
}

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

```
$OBJECTS = $COMMAND_LINE_OBJECTS;

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

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

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

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

      heap
      {
```

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

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

    OVERLAY_OUTPUT( overlay1.ovl )

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

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

        ALGORITHM( ALL_FIT )
        SIZE(0x100)

    } > mem_ovly

        // This is the second overlay. Note that these
        // OVERLAY_INPUT commands must be contiguous in the LDF
        // in order for them to occupy the same "runtime" memory.
        OVERLAY_INPUT {
```

```
    OVERLAY_OUTPUT( overlay2.ovl )
    INPUT_SECTIONS(overlay2.doj( program)))
    ALGORITHM( ALL_FIT )
    SIZE( 0x100)
  } > mem_ovly

} > program

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

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

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

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

        heap
        {
           // Allocate a heap for the application
           ldf_heap_space = .;
```

```
        ldf_heap_end =
            ldf_heap_space + MEMORY_SIZEOF(HEAP) - 1;
        ldf_heap_length = ldf_heap_end - ldf_heap_space;
    }>MEM_HEAP
}
```

2 EXPERT LINKER

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

You can also use the *Expert Linker* tool for linking data and creating an executable. The Expert Linker is a VisualDSP++ graphical tool that provides an interactive graphic environment of mapping code or data to specific memory segments.

This chapter contains:

Expert Linker Overview

The Expert Linker (EL) is a graphical tool that lets its users:

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

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

You can use [default] LDF files that come with your DSP and VisualDSP++ tools, or you can use an interactive wizard, provided in the Expert Linker, to create new LDF files.

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

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

EL is able to show graphically how much space is allocated for your program's heap and stack. After loading and running the program, it 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 up too much memory. This allows you to free up the memory to store other things like your 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 select **Open in Expert Linker**
- From the VisualDSP++ main menu, choose **Tools -> Expert Linker-> Create LDF**

Launching the Create LDF Wizard

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

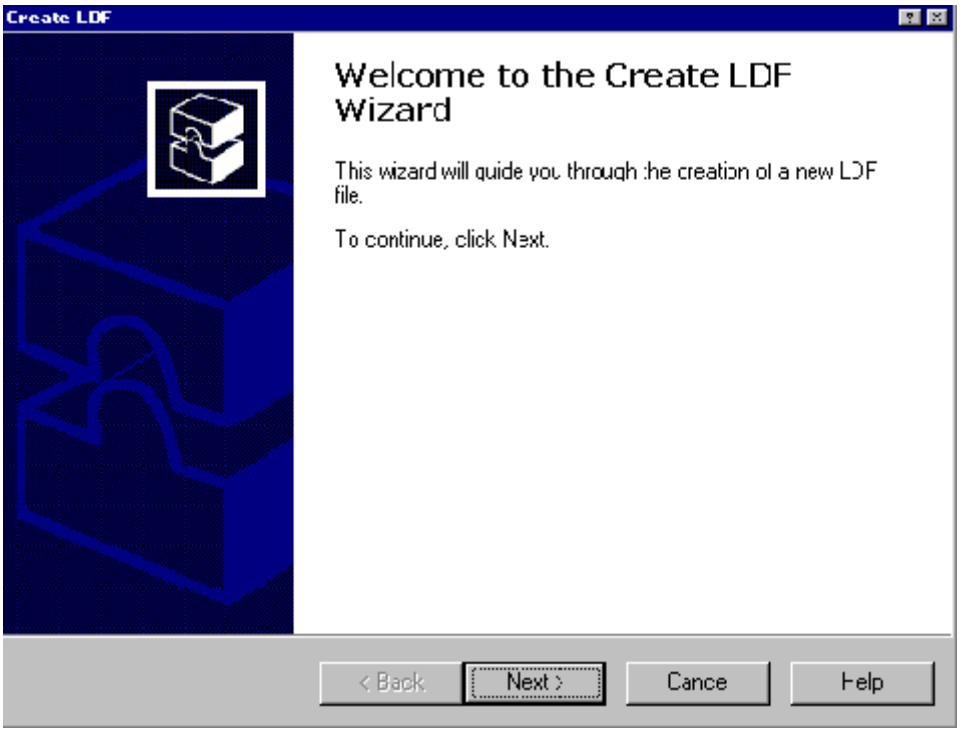

Figure 2-1. Welcome Page of the Create LDF Wizard

If there is already an LDF in the project, you are prompted to confirm whether you wish to create a new .LDF file to replace the existing one. This menu command is disabled if VisualDSP++ does not have a project opened. Press **Next** to run the wizard.

Step 1: Specifying Project Information

The first wizard window is displayed.

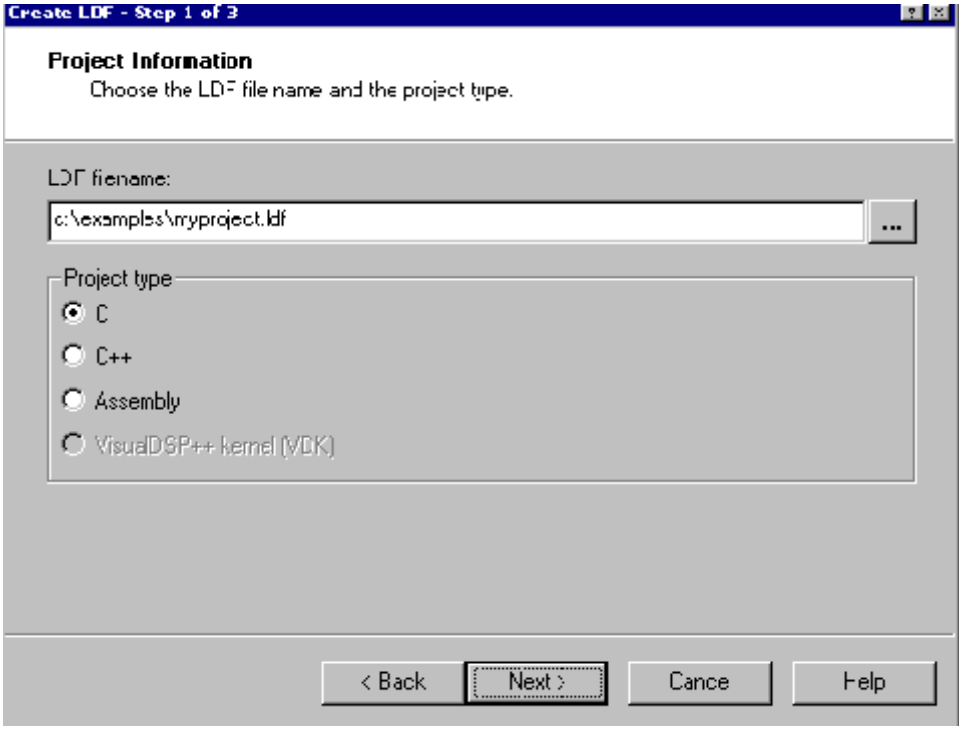

Figure 2-2. Selecting File Name and Project Type

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

The **Project type** selection specifies whether the LDF is for a C, C++, assembly, or a VDK project. The default setting depends on the source files in the project. For example, if 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**, etc. This setting determines which base 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 multiprocessor (MP) system.

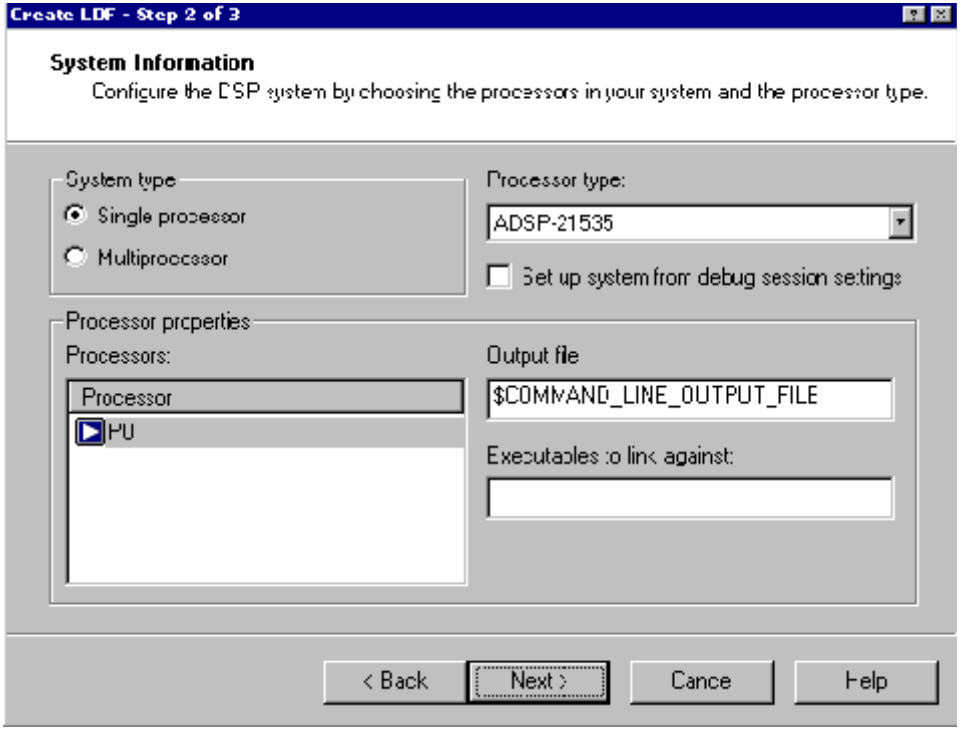

Figure 2-3. Selecting System and Processor Types

- If it is for a single processor system, the **Processors** list shows only one processor and the MP address columns disappear.
- If it is 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 memory's address range.

The **Processor type** specifies the processor architecture used in the DSP system. This setting defaults to the processor target used in the current project in VisualDSP++.

If you select **Set up system from debug session settings**, the processor information (i.e., the number of processors and the processor names) will be filled automatically from the current settings in the debug session.

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

When you select a processor in the **Processors** list, the *output file name* and *Executables to link against files* for that processor are shown 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.

Blackfin DSPs do not support the multiprocessor (MP) system architecture. However, for multiprocessor systems, the window will show the list of processors in the project and the MP address range for each processor.

For example, if the current debug session is "multiprocessor", the **Multiprocessor** button is automatically selected and processor names from the debug session are shown in the **Processors** list. In addition, if EL can detect the ID of the processor, it places the processor in the right position in the processor list.

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

Step 3: Completing the LDF Wizard

The system displays the **Wizard Completed** pane. You can use it to go back and verify and/or modify choices made up to this point.

When you click the **Finish** button, EL makes a copy of the base template .LDF file and places it in the same directory as the project file. The new .LDF file is added to the current project. The EL pane is displayed with a new .LDF file.

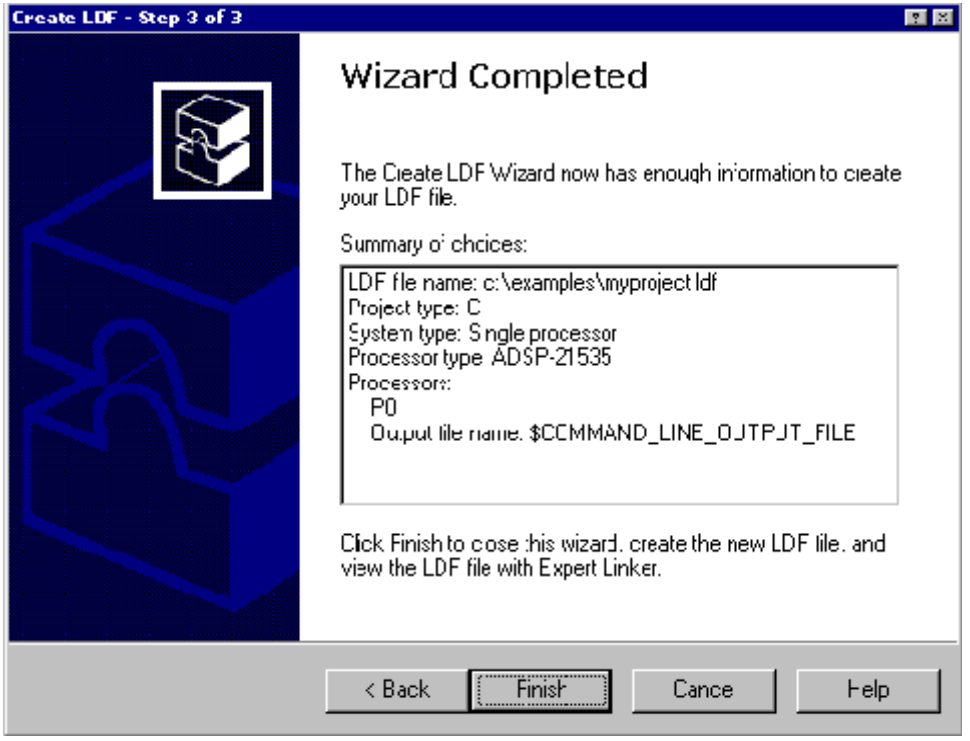

Figure 2-4. Wizard Completed Page of the Create LDF Wizard

Expert Linker Window Overview

The Expert Linker window, which can dock or float in VisualDSP++, contains two main panes:

- An **Input Sections** pane uses a tree to display the list of input sections (see "Using the Input Sections Pane" on page 2-12)
- A **Memory Map** pane displays each memory map in a tree or graphical representation (see "Using the Memory Map Pane" on page 2-18)

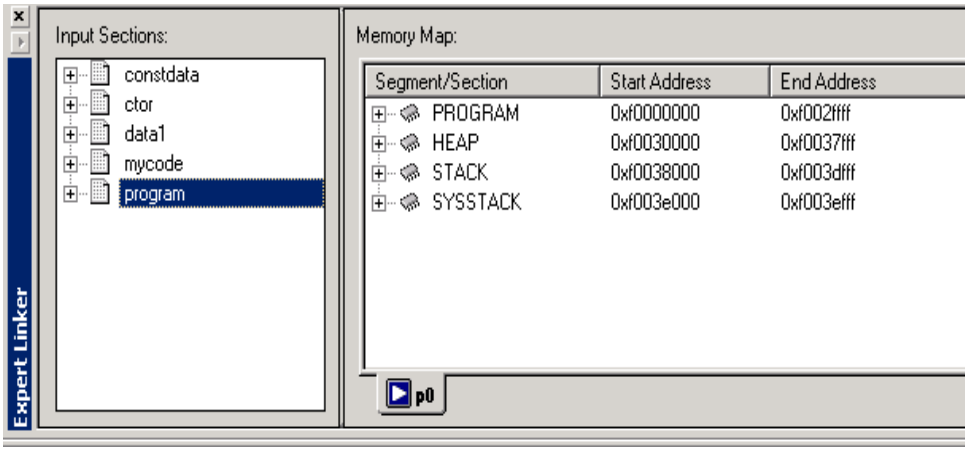

Figure 2-5. Expert Linker Window

Using commands in LDF, the linker reads the Input Sections in object files 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 has to place the Input Sections.

Using the Expert Linker, you can map an input section to an output section in the memory map by dragging-and-dropping it onto the output section. Each memory segment may have one or more output sections under it. Input sections that have been mapped to an output section dis-

play under that output section. For more information, refer to "Using the Input Sections Pane" on page 2-12 and "Using the Memory Map Pane" on page 2-18.

You can access various functions of the Expert Linker using the right button on your mouse. Right-click the mouse to display appropriate menus and make function selections.

The following code example shows an example .LDF (formatted for easy reading). The linker (linker.exe) maps your program code (and data) to the external system memory and processor memory. The linker uses the target system's memory map (as determined in the .LDF) and segments (defined in your source file) to create an executable program. Note that the .LDF file includes two commands (MEMORY and SECTIONS) that combine program and system information.

```
ARCHITECTURE(ADSP-21535)
SEARCH_DIR($ADI_DSP\Blackfin\lib)
$OBJECTS = $COMMAND_LINE_OBJECTS */
MEMORY /* Define/label system memory  */
{      /* List of global Memory Segments */
   MEM_PROGRAM
   { TYPE(RAM) START(0xF0000000) END(0xF002FFFF) WIDTH(8) }
   MEM_HEAP
   { TYPE(RAM) START(0xF0030000) END(0xF0037FFF) WIDTH(8) }
   MEM_STACK
   { TYPE(RAM) START(0xF0038000) END(0xF003DFFF) WIDTH(8) }
   MEM_SYSSTACK
   { TYPE(RAM) START(0xF003E000) END(0xF003FDFF) WIDTH(8) }
   MEM_ARGV
   { TYPE(RAM) START(0xF003FE00) END(0xF003FFFF) WIDTH(8) }
}
PROCESSOR p0 /* The processor in the system */
{
   OUTPUT($COMMAND_LINE_OUTPUT_FILE)
    SECTIONS
   {   /* List of sections for processor P0 */
      program
      {
         INPUT_SECTION_ALIGN(2)
         /* Align all code sections on 2 byte boundary */
```

```
            INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
            INPUT_SECTION_ALIGN(1)
            INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
            INPUT_SECTION_ALIGN(1)
            INPUT_SECTIONS(
               $OBJECTS(constdata)
               $LIBRARIES(constdata))
            INPUT_SECTION_ALIGN(1)
            INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
         } >MEM_PROGRAM

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

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

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

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

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

Using the Input Sections Pane

The **Input Sections** pane (Figure 2-5 on page 2-9) initially displays a list of all the input sections, referenced by the `.LDF` file, and all input sections contained in the object files and libraries. Under each input section, there may be a list of LDF macros, libraries, and object files contained in that input section. You can add or delete input sections, LDF macros, or objects/library files in this pane.

Using the Input Sections Menu

When you right-click an object in the **Input Sections** pane, the menu is displayed.

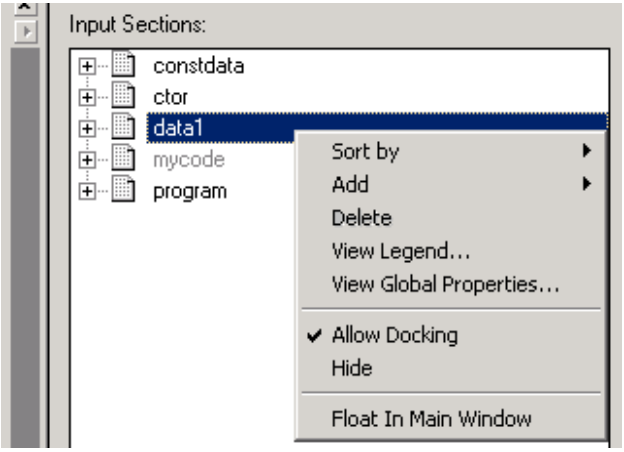

Figure 2-6. Input Sections Right-Click Menu

The main menu functions include:

- **Sort by** — Allows you to sort objects by input sections or LDF macros. These selections are mutually exclusive.

- **Add** — Allows you to add input sections, object/library files, and LDF macros. Appropriate menu selections are grayed out if you right-click on a position (area) in which you cannot create a corresponding object.

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

- **Delete** — Allows you to delete the selected object (input sections, object/library files, or LDF macros).

- **View Legend…** — Displays the **Legend** dialog box that shows all possible **Icons** and **Colors** that can be used by the Expert Linker.

- **View Section Contents** — Opens the **Section Contents** dialog box, which displays the contents of the input. This command is available only after you link or build the project and then right-click on an input.

- **View Global Properties…** — Displays a **Global Properties** dialog box that provides the map file name (of the map file generated after linking the project) as well as access to some processor and setup information (see Figure 2-31 on page 2-42).

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 cursor to an output section, and then release the mouse button and drop the input section onto the output section.

All objects, like an LDF macro or object file 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.

If an input section is dragged into a memory segment with no output section in it, an output section with a default name is automatically created and displayed.
The red cross mark on a icon (for example, ) indicates that this object/file is unmapped yet. Once an input section has been complete mapped (all object files that contain the section are mapped), the icon next to the input section changes to indicate that it has been mapped; the '**x**' disappears. See Figure 2-7 on page 2-15.

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 2-7 on page 2-15).

The red cross mark on a icon indicates this object/file was not mapped yet.

Click the **Colors** tab to view the **Colors** pane (Figure 2-8 on page 2-15). It contains a list of colors used in the [graphical] memory map view; each item's color can be customized.

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 2-9).

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

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

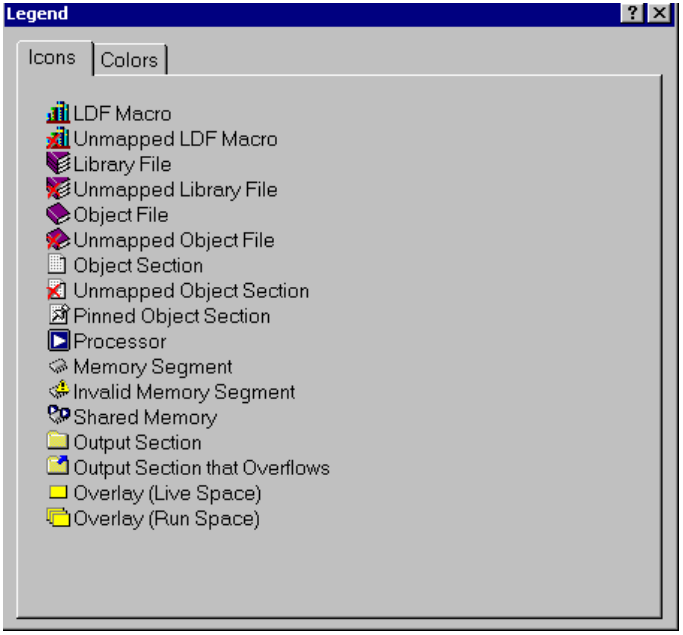

Figure 2-7. Legend Dialog Box–Icons Pane

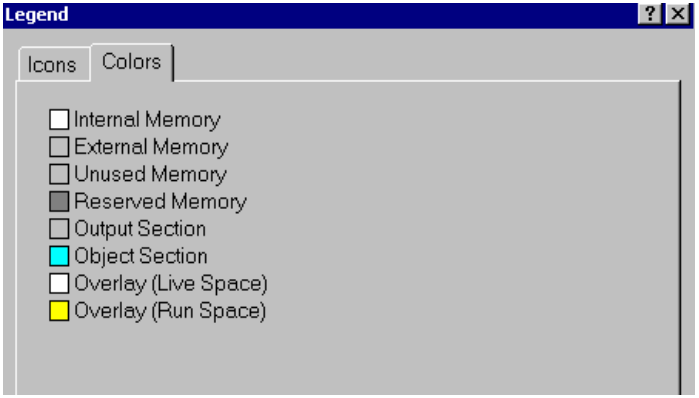

Figure 2-8. Legend Dialog Box–Colors Pane

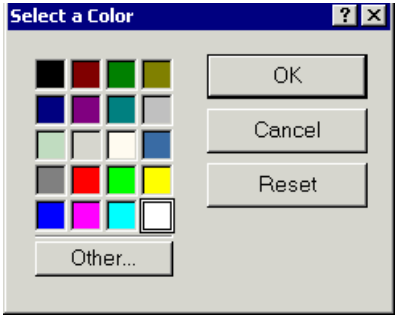

Figure 2-9. Selecting Colors

Use the **Reset** button to reset all memory map colors to the default colors.

Sorting Objects

Objects in the **Input Sections** pane can be sorted 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,

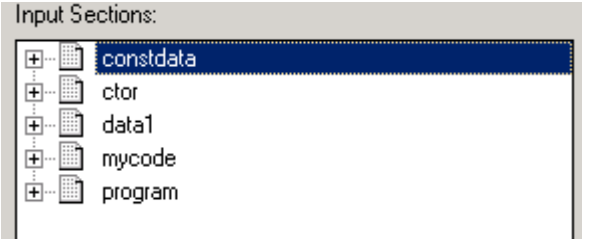

Figure 2-10. Expert Linker Window—Sorted by Input Sections

Under each macro, there may be other macros, object files, or libraries. Under each object file, there are input sections contained in that object file.

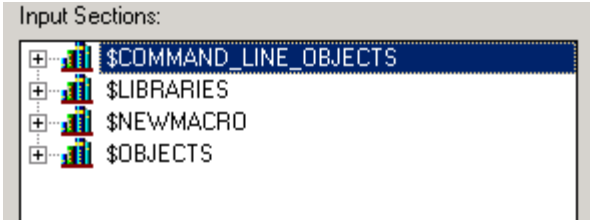

Figure 2-11. Expert Linker Window – Sorted by LDF Macros

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

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 (program, data, stack...). It thereby connects your program to the target system. The OUTPUT() command directs the linker to produce an executable (.DXE) file, specifying the file name.

In the Expert Linker, the **Memory Map** pane contains tabbed pages (panes) that can display memory maps for a processor or shared memory. This window provides two viewing modes—a **tree** view and a **graphical** view.

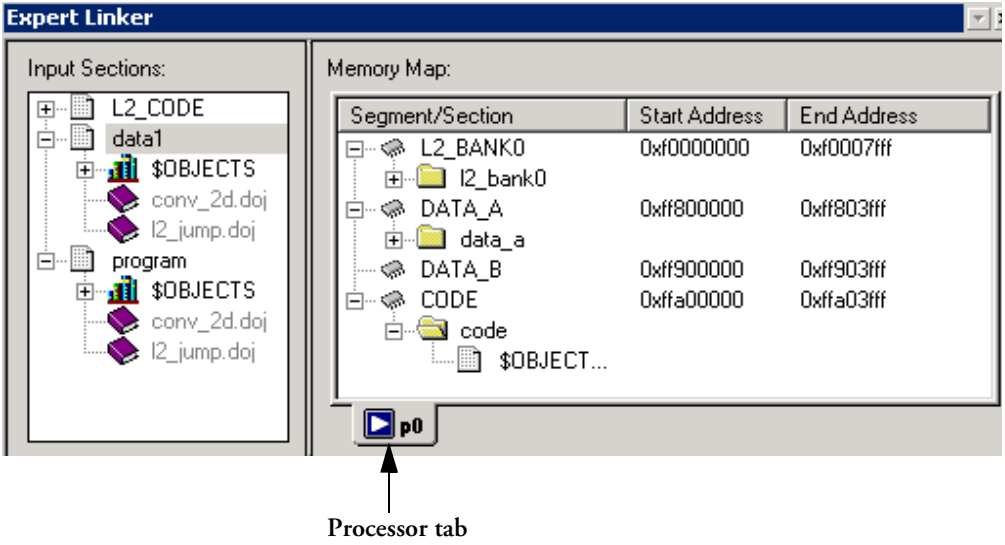

Figure 2-12. Expert Linker Window – Memory Map

You can select these views and other memory map features using the context (or right-click) menu. All procedures involving memory map handling assume that the Expert Linker window is open.

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

Invalid Memory Segment Notification:

If a memory segment is invalid (for example, the memory range overlaps another memory segment, the memory width is invalid, and so on), the tree shows an **Invalid Memory Segment** icon (see also Figure 2-7 on page 2-15). When you move the mouse cursor over the icon, a tooltip displays a message, describing why the segment is invalid.

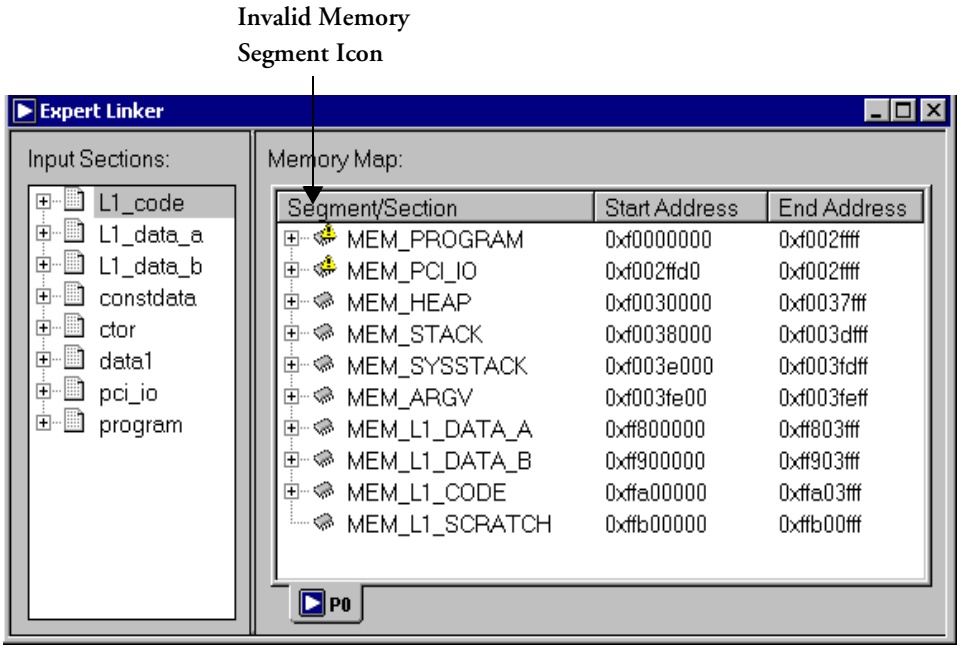

Figure 2-13. Memory Map with Invalid Memory Segment icons

Using the Context Menu

To display the context menu, right-click in the **Memory Map** pane. The context (right-click) menu allows you to select and perform the major functions including:.

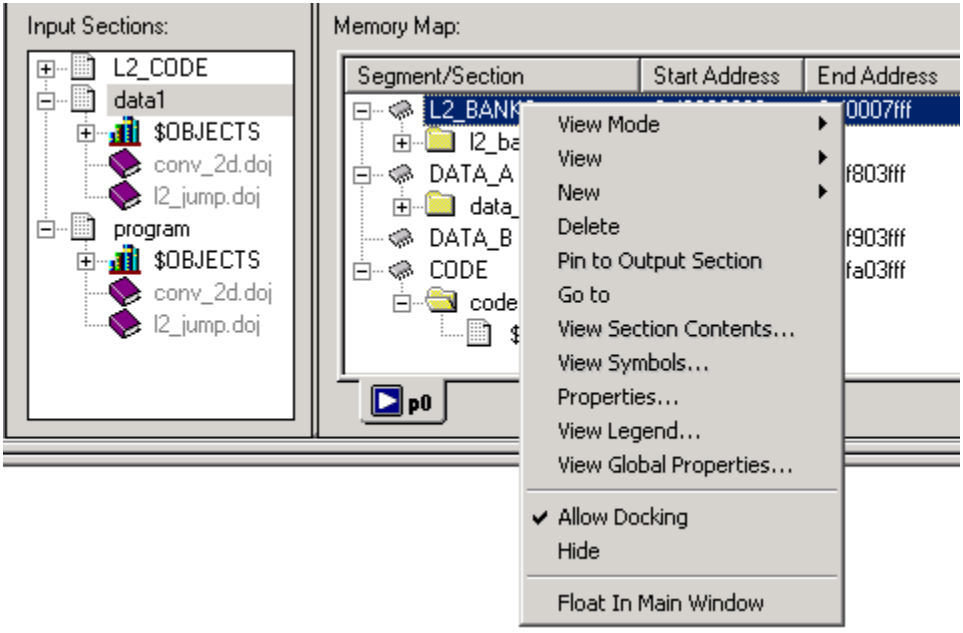

Figure 2-14. Context (Right-Click) Menu

View Mode

- **Memory Map Tree** — Displays the memory map in a "tree "representation (see Figure 2-15 on page 2-23).
- **Graphical Memory Map** — Displays the memory map in graphical blocks (see Figure 2-16 on page 2-24).

View

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

New

- **Memory Segment…** — Allows you to specify the name, address range, type, size, and so on of the memory segment you want to add.
- **Output Section** — Adds an output section to the selected memory segment (right-click on the memory segment to access this command). This menu option is disabled if you do not right-click on a memory segment.

  The options are: **Name**, **Overflow** (**output section to overflow** or **None**), **Packing**, and **Number of bytes** (number of bytes to be re-ordered at one time. This value does not include the number of null bytes inserted into memory).
- **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 2-43 on page 2-58).

Delete — Deletes the selected object.

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

View Section Contents… — Available only after linking or building the project and then right- clicking on an input or object section. Invokes a dialog box that displays the contents of the input or output section (see Figure 2-27 on page 2-37).

View Symbols… — Available only after linking the project and then right-clicking on a processor, overlay, or input section. Invokes a dialog box that displays the symbols for the project, overlay, or input section (see Figure 2-30 on page 2-40).

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

View Legend… — Displays a **Legend** dialog box that shows all possible icons in the tree window and a short description of each icon. On the **Colors** page, there is a list of colors used in the graphical memory map. Each object's color can be customized. See Figure 2-7 on page 2-15 and Figure 2-8 on page 2-15.

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 2-31 on page 2-42).

Tree View Memory Map Representation

In the tree view, chosen via the **View Mode -> Memory Map Tree** menu selection, the memory map displays with memory segments at the top-level.

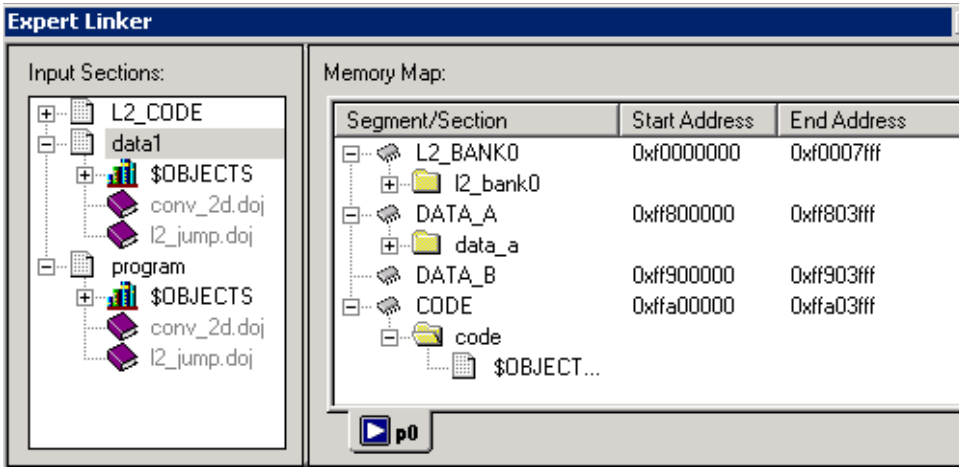

Figure 2-15. Expert Linker Window – Memory Map

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

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

Graphical View Memory Map Representation

In the graphical view, chosen via the **View Mode -> Graphical Memory Map** menu selection, the graphical memory map displays the processor's hardware memory map (refer to your DSP's Hardware Reference Manual or datasheet). Each hardware memory segment contains a list of memory segments defined by the user.

You can view the memory map from two perspectives—pre-link and post-link (see "Specifying Pre- and Post-Link Memory Map View" on page 2-29).

A graphical memory map might look like this.

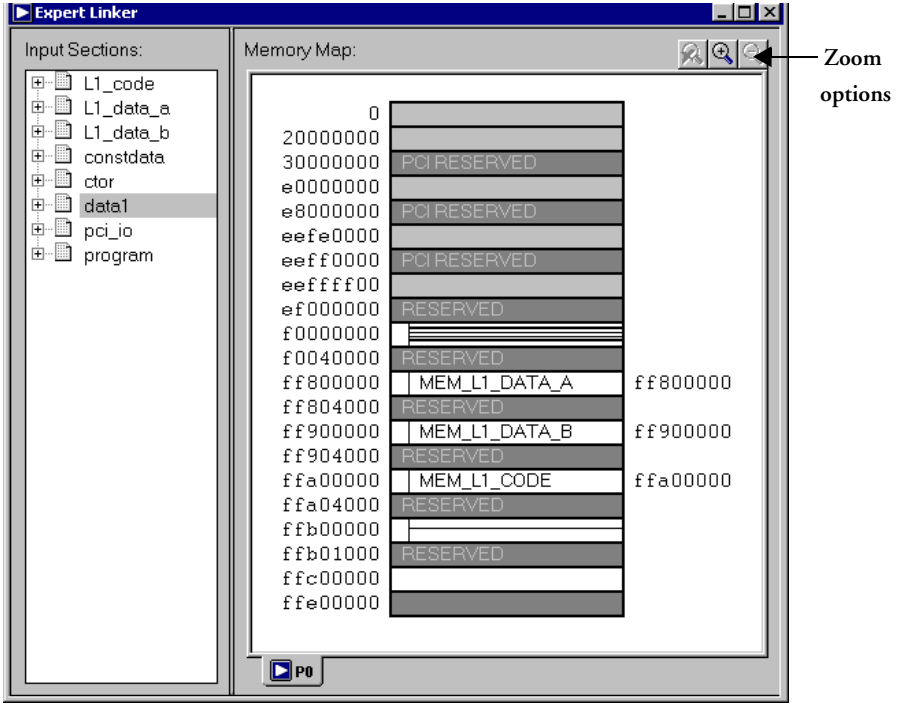

Figure 2-16. Graphical Memory Map Representation

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

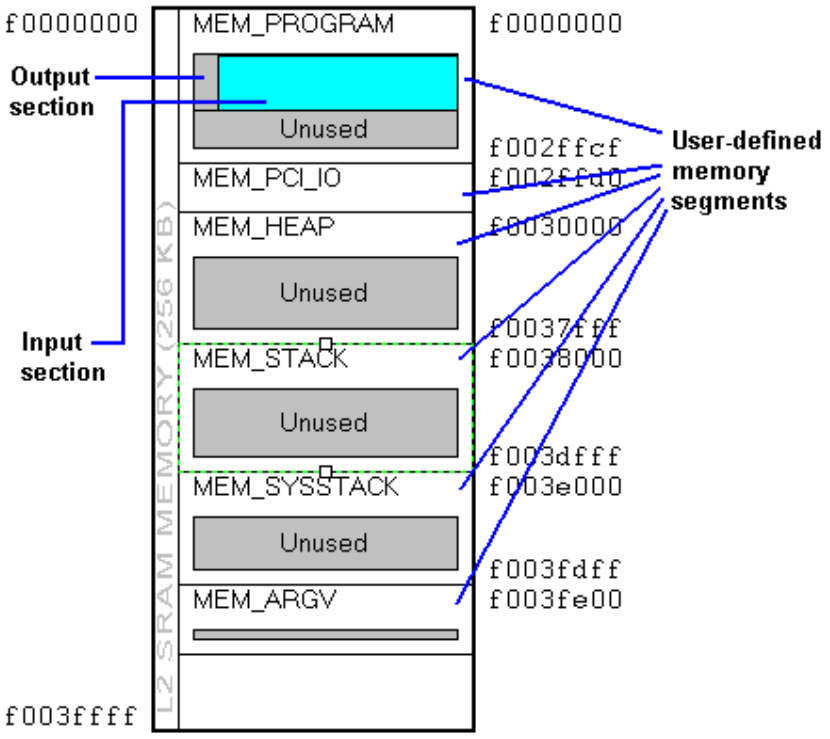

Figure 2-17. Viewing Sections and Segments in Memory Map

- The output section is drawn as a vertical header with a group of object to the right of it.
- A memory segment's border and text change to red (from its normal black color) to indicate that it is invalid. When you move the mouse cursor 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 linking, the height of the objects is scaled as a percentage of the total memory segment size. The object section name is displayed only if there is enough room to display it.
- Addresses are ordered in ascending order from top to bottom.

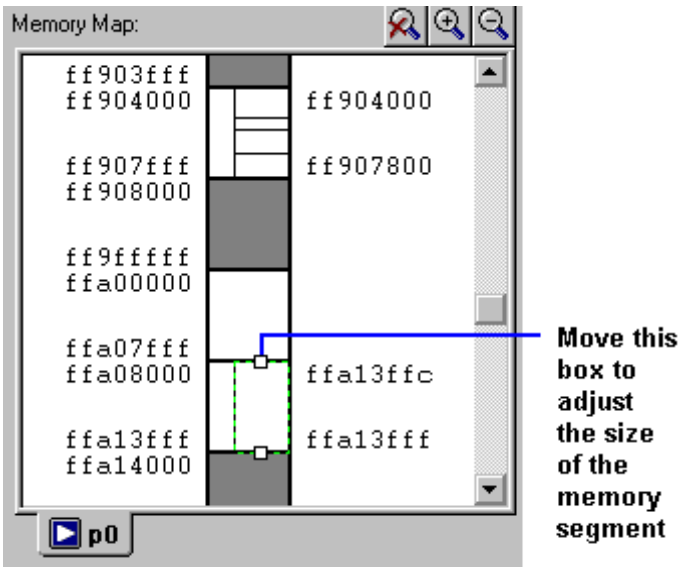

Figure 2-18. Adjusting Memory Segment Size

You can zoom in/out incrementally or zoom in/out completely using the three buttons at the top right of the memory pane. If there is not enough room to display the memory map when zoomed in, there are horizontal and/or vertical scroll bars available to view the entire memory map (for

more information, see "Zooming In and Out on the Memory Map" on page 2-29

You can drag-and-drop any object except memory segments.

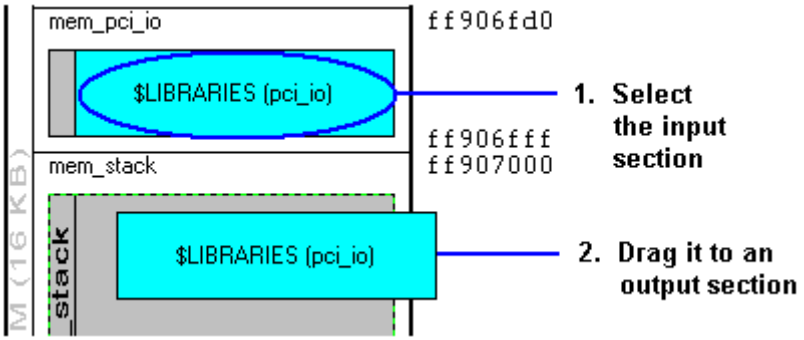

Figure 2-19. Dragging-and Dropping Objects

Memory segments, when selected, have a border box around them. If the box is clicked and dragged to reduce the size of the memory segment, the size of both the selected and adjacent memory segments change.

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

When an object is selected in the memory map, it is highlighted as shown in Figure 2-20. If you move the mouse cursor over an object in the graphical memory map, a yellow tooltip box appears containing the information about the object, for example, its name, address, and size.

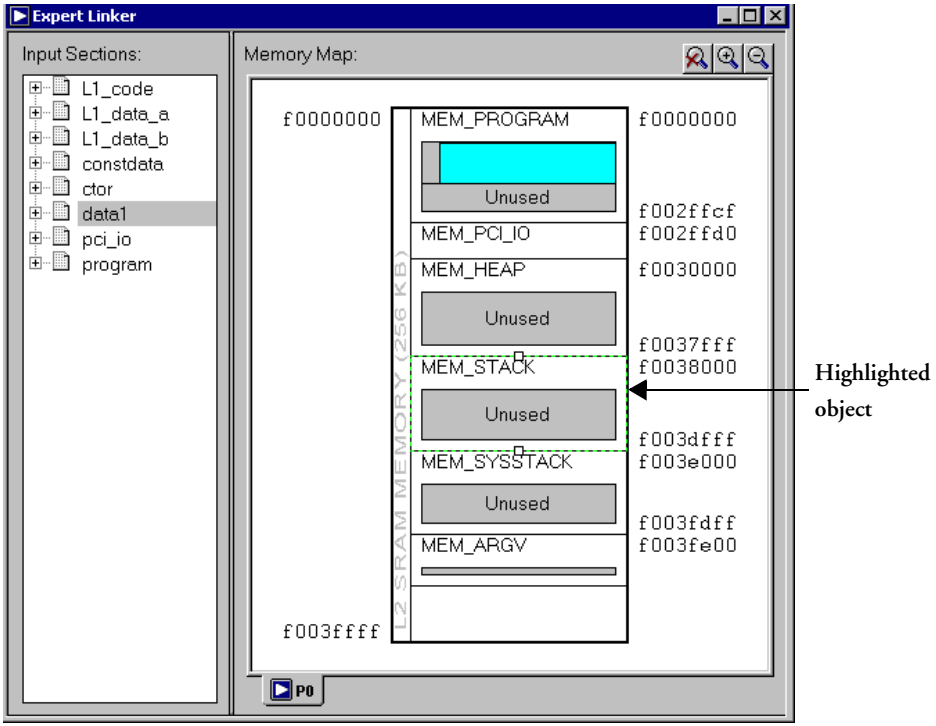

Figure 2-20. Memory Map Showing Highlighted Memory Segment

Specifying Pre- and Post-Link Memory Map View

You can view the memory map from two perspectives: pre-link and post-link. Pre-link view is typically used to view where you wanted to place your input sections. Post-link view is typically used to view where the input sections got placed after linking your project. There is other information available after linking (like being able to see the sizes of each section, viewing symbols, and viewing the contents of each section).

- To enable pre-link view from the **Memory Map** pane, right-click and choose **View** and **Mapping Strategy** (**Pre-Link**)
  Figure 2-21 illustrates memory map before linking.
- To enable post-link view from the **Memory Map** pane, right-click and choose **View** and **Link Result** (**Post-Link**).
  Figure 2-22 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 allow you to 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.

You can:

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

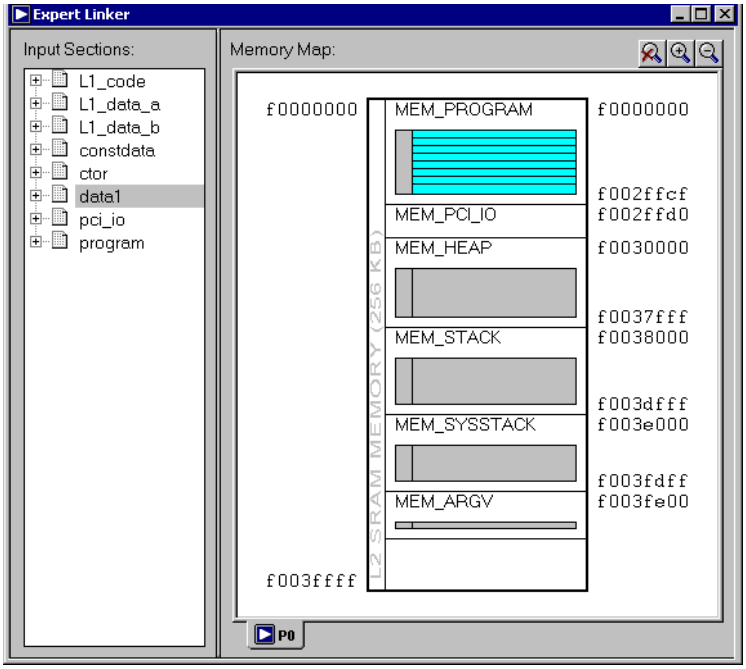

Figure 2-21. Memory Map – Pre-Link View

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

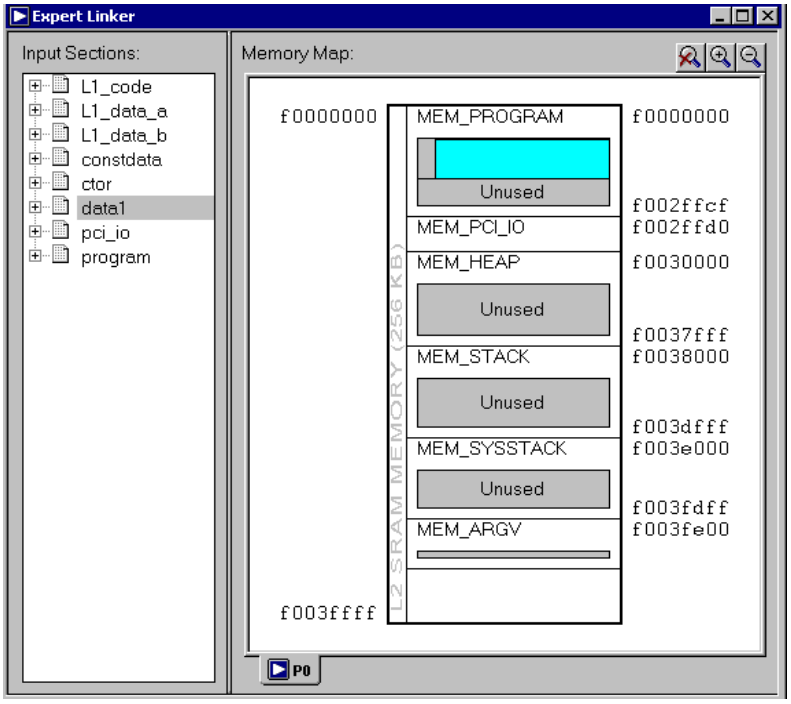

Figure 2-22. Memory Map – Post-Link View

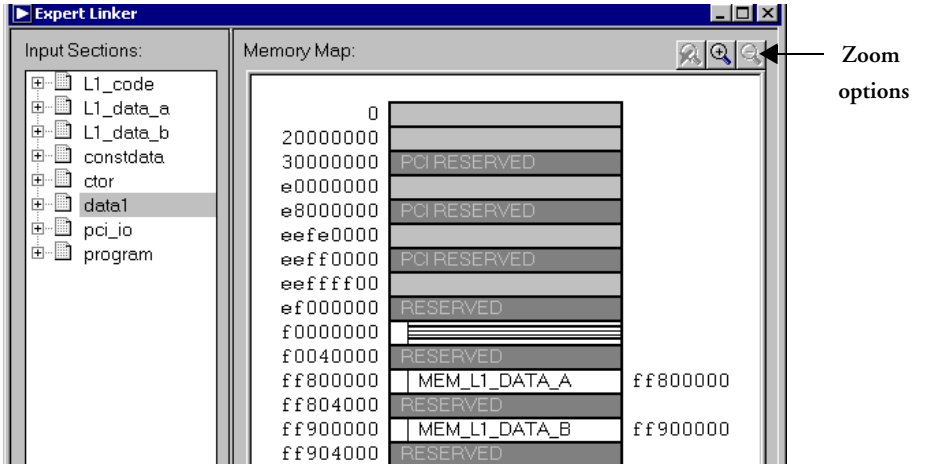

Figure 2-23. Memory Map – Zoom Options

Inserting a Gap into 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. Select **Insert gap...** in the menu. The **Insert Gap** dialog box appears.

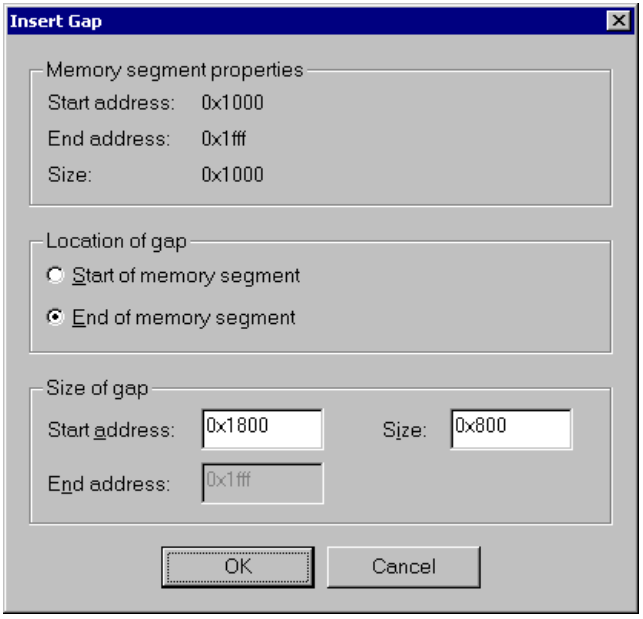

Figure 2-24. Insert Gap Dialog Box

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

Working with Overlays

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

Figure 2-25 shows the overlay in live space and Figure 2-26 shows the overlay in run space.

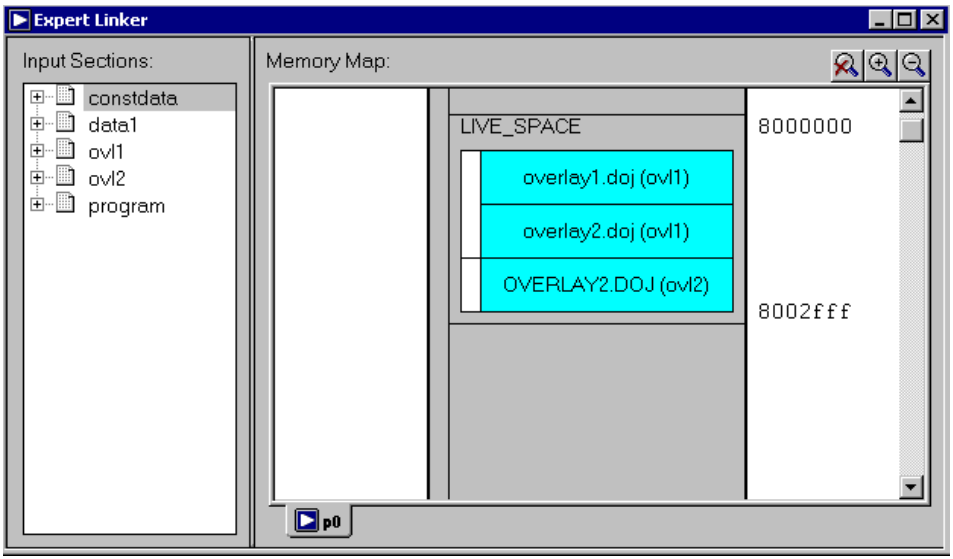

Figure 2-25. Graphical Memory Map Showing Overlay Live Space

Overlays in a “run” space are displayed one at a time in the graphical memory map. The scroll bar next to an overlay in “run” space allows you to pick 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.

You can use the Up/Down arrow in the header to display previous or next overlay in the “run” space. You can also use the browse button to display the list of all available overlays you can choose from. The header shows the number of overlays in this run space and the current overlay number.

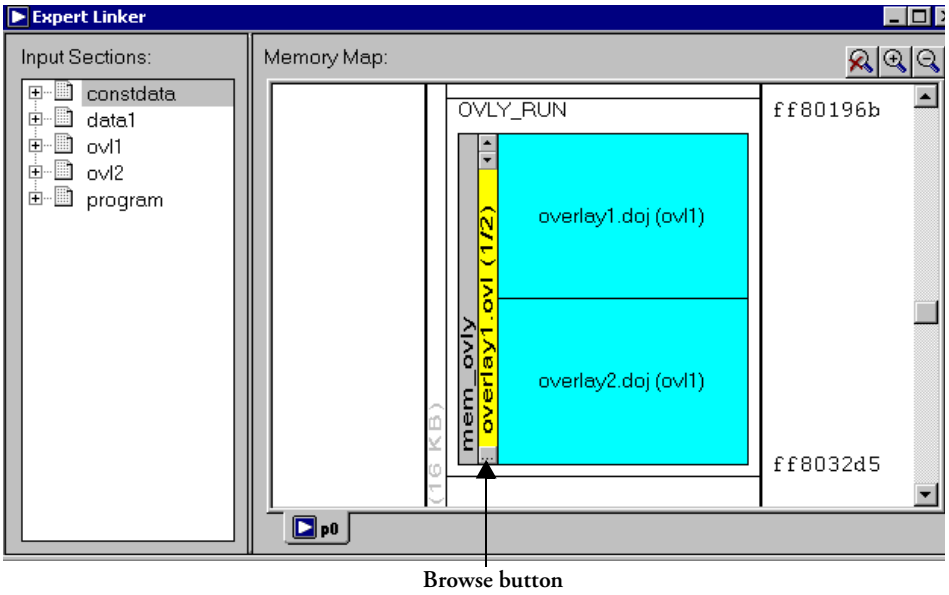

Figure 2-26. Graphical Memory Map Showing Overlay Run Space

To create an overlay in the “run” space:

1. Right-click on an Output Section.
2. Choose **New -> Overlay** from the menu.
3. Select the “live” space from the **Overlay Properties** dialog box. The new overlay is displayed in the “run” and “live” spaces in two different colors in the memory map.
4. Drag the overlay in the “live” space to a different output section. This can change the “live” to “run” space.

Viewing Section Contents

You can view the contents of an input or output section including a particular memory address and specifying the display's format.

Use this feature to employ `elfdump` to obtain the section contents and display it in a window that is similar to a Memory window in VisualDSP++. Multiple **Section Contents** dialog boxes may be displayed.

For example, to display the contents of an output section:

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

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

Figure 2-28 and Figure 2-29 illustrate other memory data formats available for the selected output section.

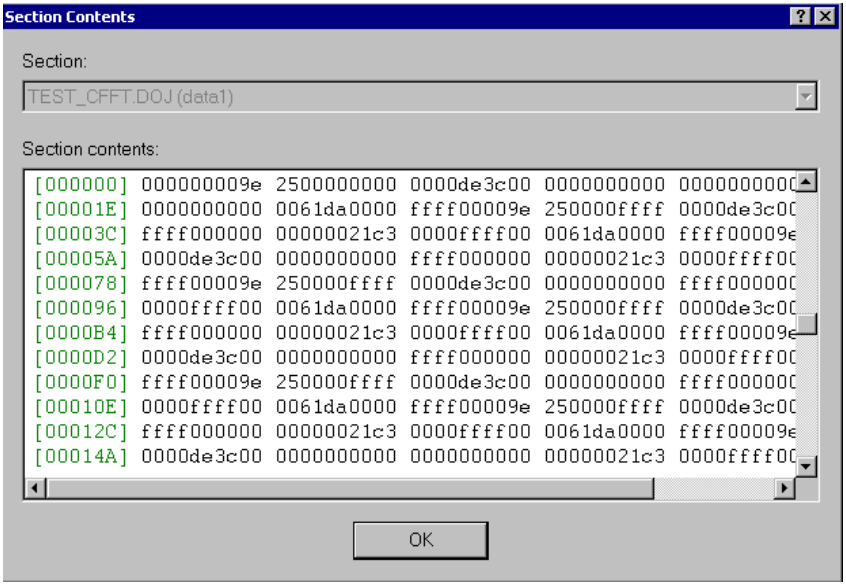

Figure 2-27. Output Section Contents in Hex Format

Viewing Symbols

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

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.

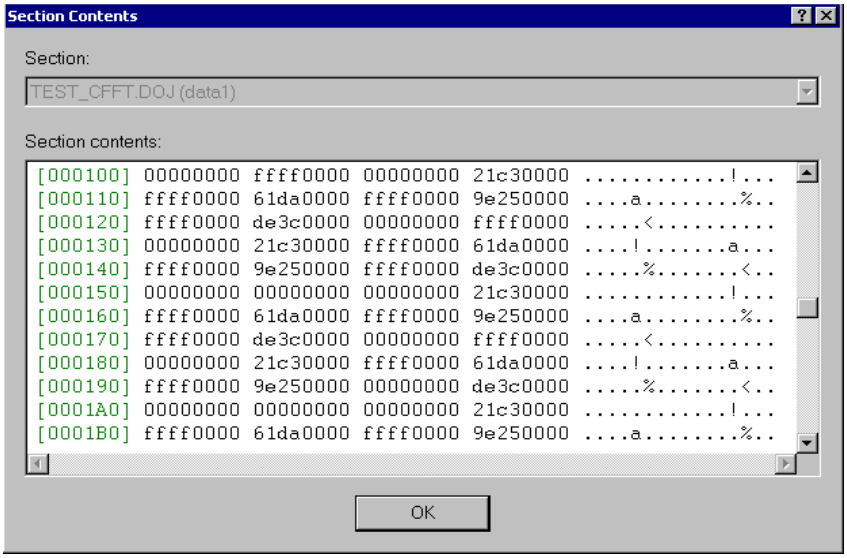

Figure 2-28. Output Section Contents in Hex and ASCII Format

2. Right-click to choose **View Symbols…**.

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

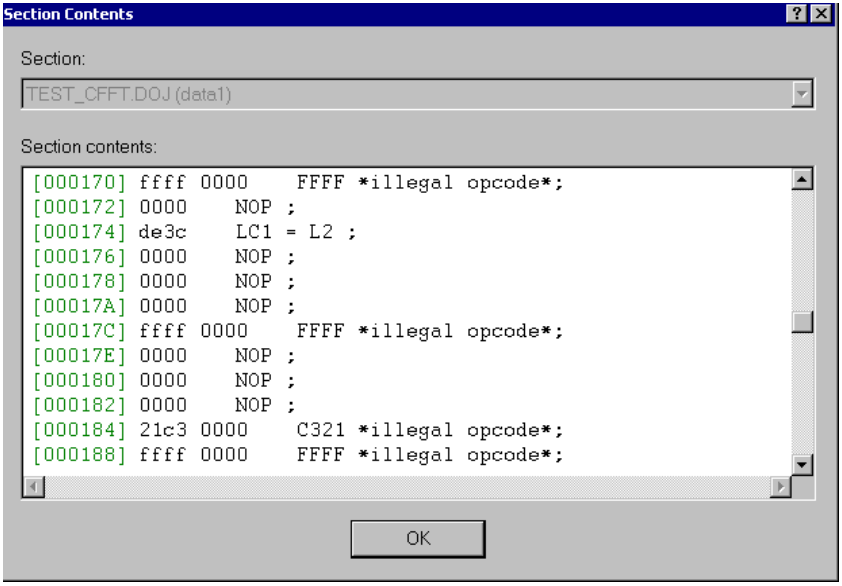

Figure 2-29. Output Section Contents in Hex and Assembly Format

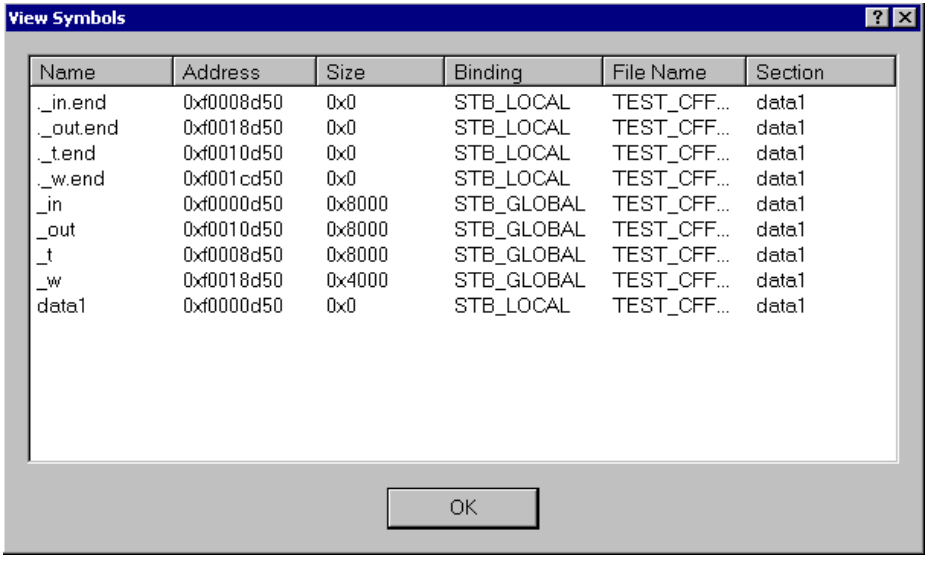

Figure 2-30. View Symbols Dialog Box

Managing Object Properties

You can display different properties for each type of object. Since different objects may share certain properties, their **Properties** dialog boxes share property pages. All these procedure assume that the Expert Linker window is open.

To display a property dialog box, right-click an object and choose **Properties** from the menu. You may choose the following functions:

Managing Global Properties

Use the context menu to display Global Properties:

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

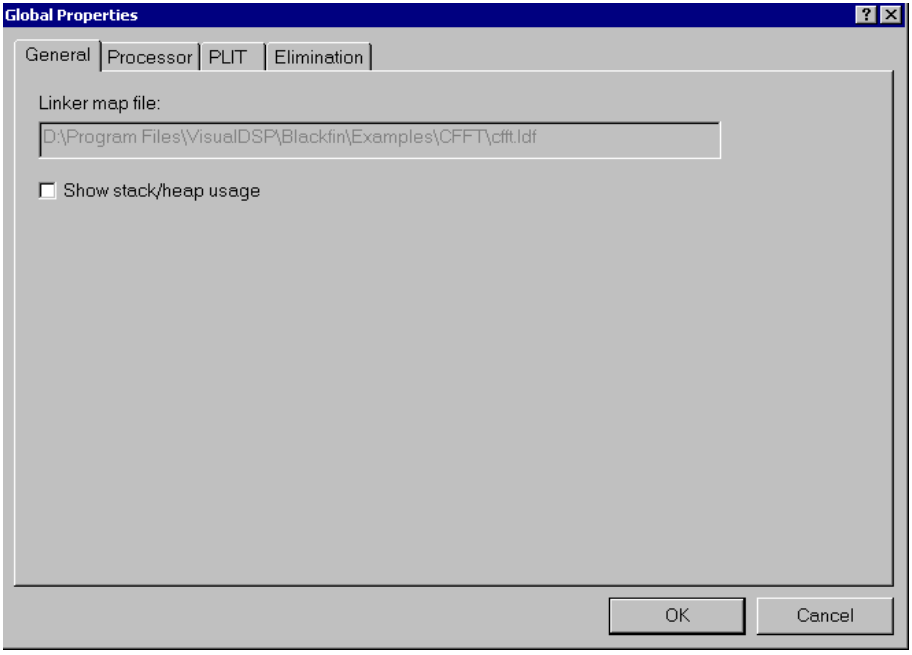

Figure 2-31. Global Properties - General Tab

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

- The **Linker map file** displays the map file generated after linking the project —Read-only field.
- If **Show stack/heap usage** is selected, after you run a project, EL is going to show how much of the stack and heap were used.

Managing Processor Properties

To specify Processor Properties:

1. Right-click the **Processor** tab anywhere in the Expert Linker window.
2. Choose **Properties**.
   The **Global Properties** dialog box appears.

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

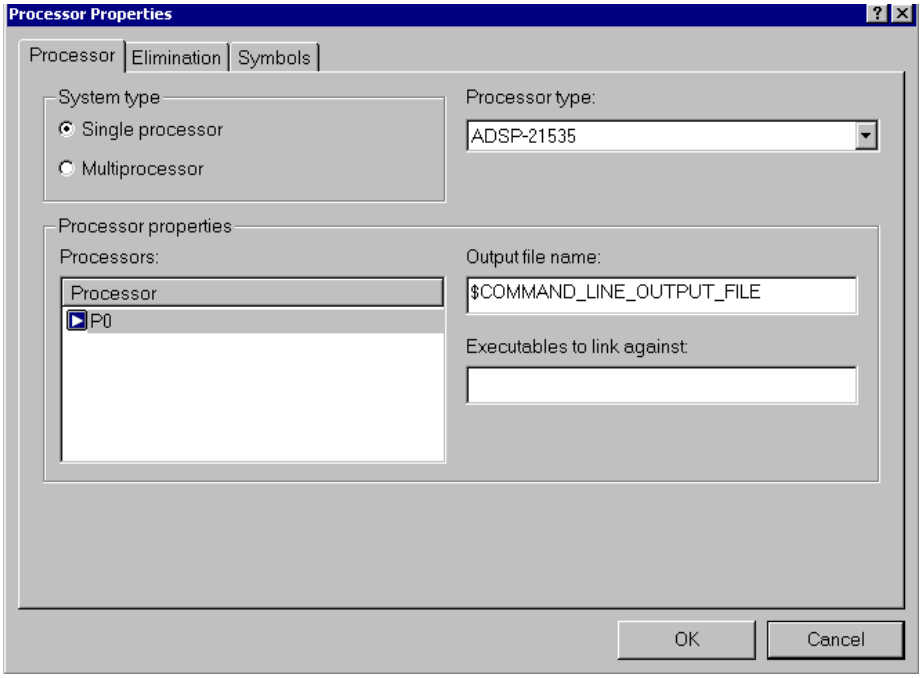

Figure 2-32. Global Properties – Processor Tab

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

- Specify **System Type —** Use the **Single processor** selection.
- Select a **Processor type** (such as ADSP-21535 or ADSP-21532).
- 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. The files must be `.SM`, `.DLB`, or `.DXE` files. 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**.

Then type the new name.

Managing PLIT Properties for Overlays

The **PLIT** pane allows you to view and edit the function template used in overlays. You can type in assembly instructions, same as syntax-colored assembly code specified via the Editor.

You can only enter assembly code—no comment entry is allowed.

To view and edit PLIT information:

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

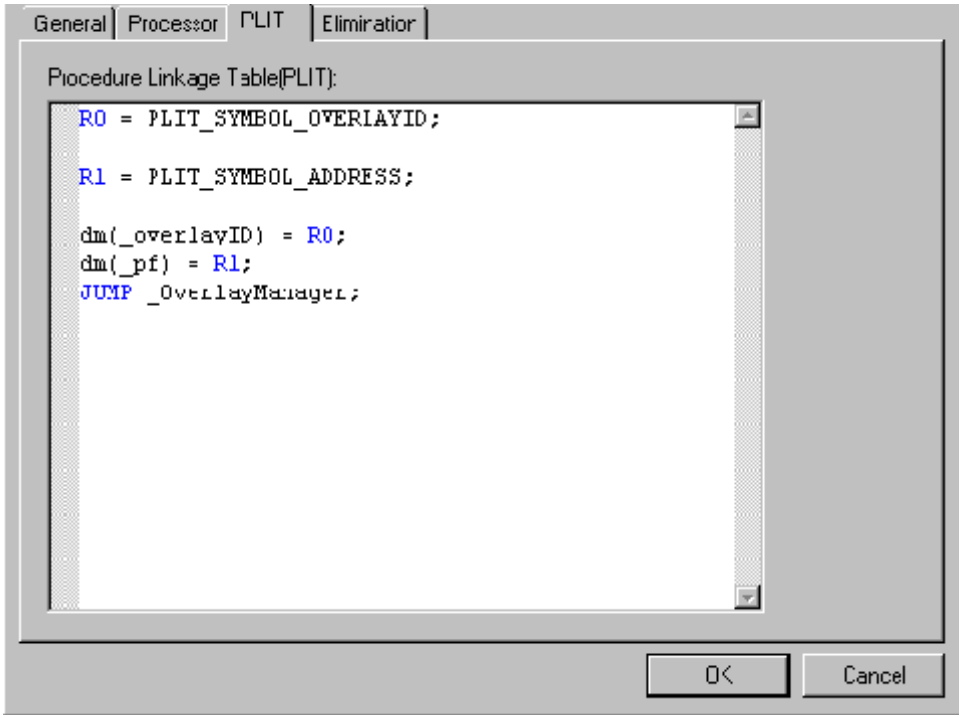

Figure 2-33. Global Properties – PLIT Tab

Managing Elimination Properties

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

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

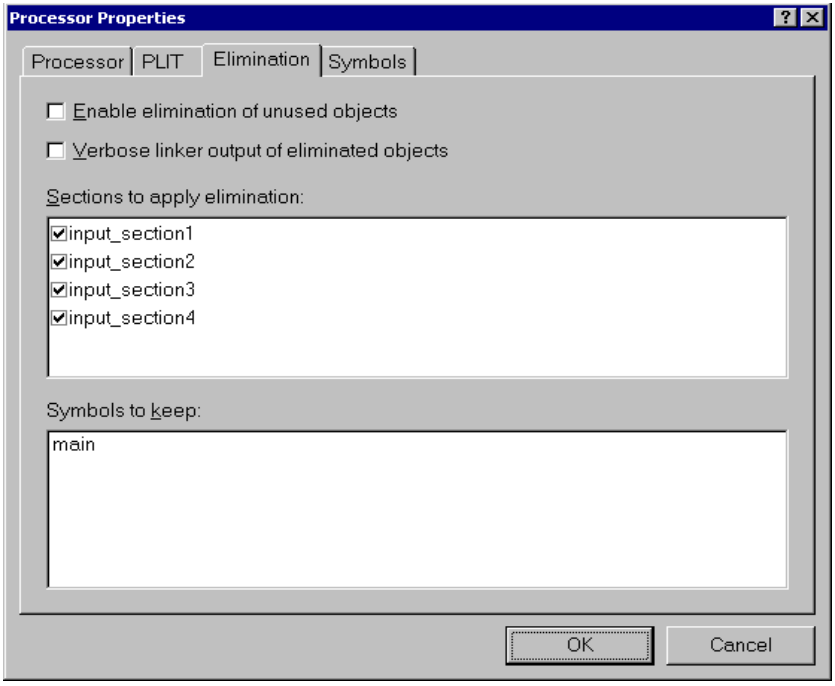

Figure 2-34. Global Properties – Elimination Tab

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

If **Verbose linker output of eliminated objects** is selected, the eliminated objects will be shown as linker output in the **Output** window's **Build** tab of VisualDSP++ during linking. This checkbox is grayed out if **Enable**

elimination of unused objects is unchecked. 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. The elimination applies to the sections that are selected. By default, all input sections are selected.

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

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

Managing Symbols Properties

You can view the list of symbols that the linker is to resolve. You can also add and remove symbols from the list of symbols kept by the linker. The symbols can be resolved to an absolute address or to a program file (.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 EL window.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. To add or remove a symbol, click the **Elimination** tab.

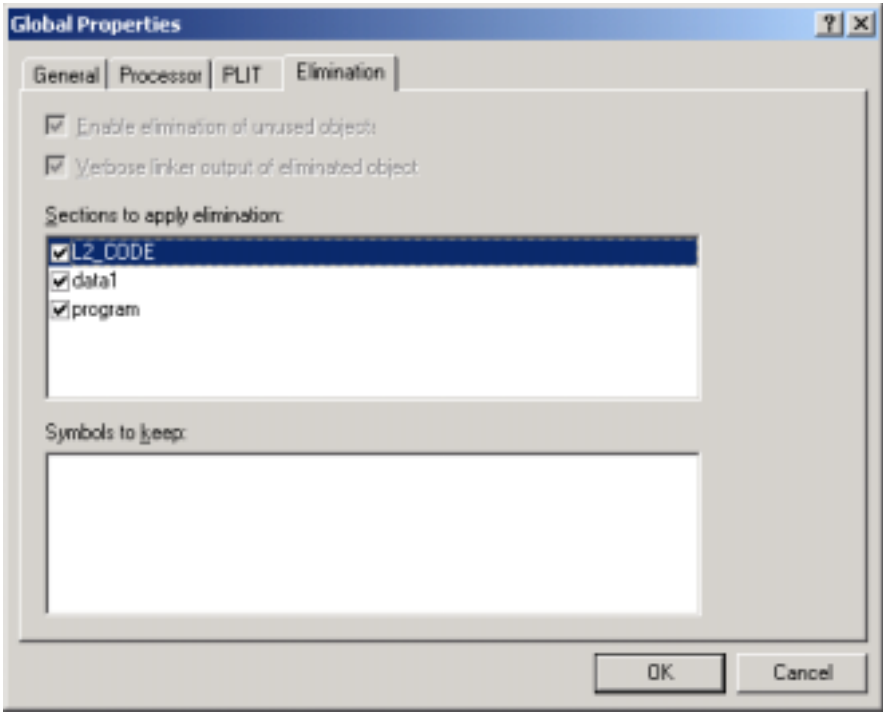

Figure 2-35. Elimination Properties – Symbols to Keep Window

4. Right-click in the **Symbols to keep** window.

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

Specifying Symbol Resolution

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

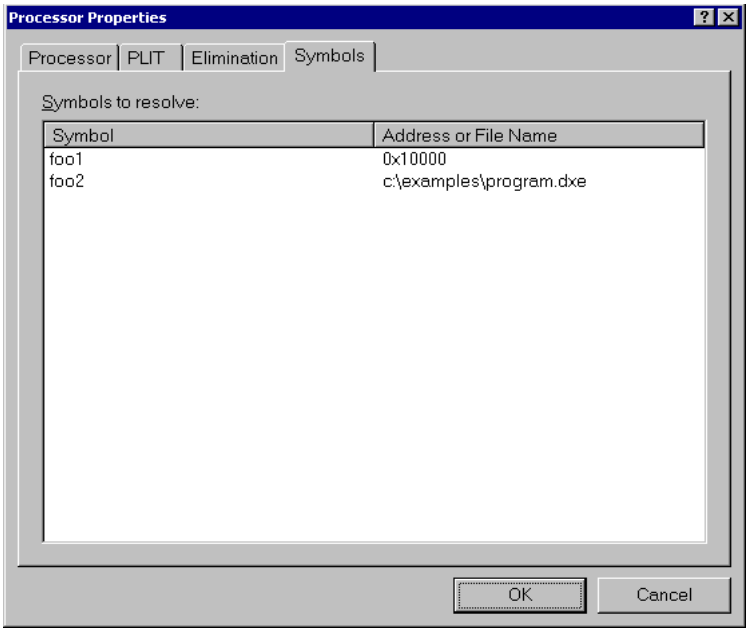

Figure 2-36. Processor Properties – Symbols Tab

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

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

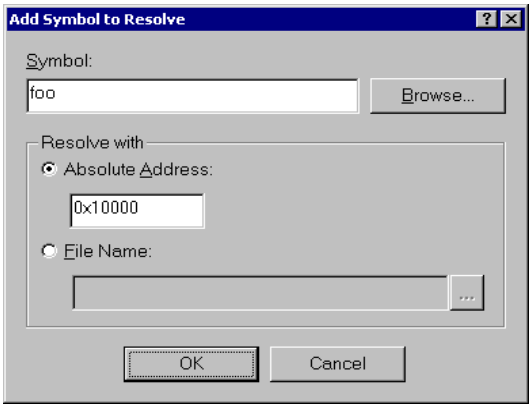

Figure 2-37. Add Symbol to Resolve Dialog Box

The **Browse** button is grayed out when no symbol list is available; for example, if the project has not been linked yet. When this button is active, click it to display the **Browse Symbols** dialog box with a list of all the symbols (see Figure 2-38).

You can select a symbol from that list and it will appear in the **Symbols** box.

Deleting a Symbol from the Resolve List

Use the Browse button to display the **Symbols to resolve** list in the **Symbols** pane (Figure 2-36). Select a symbol you want to delete. Right-click and choose **Remove Symbol**.

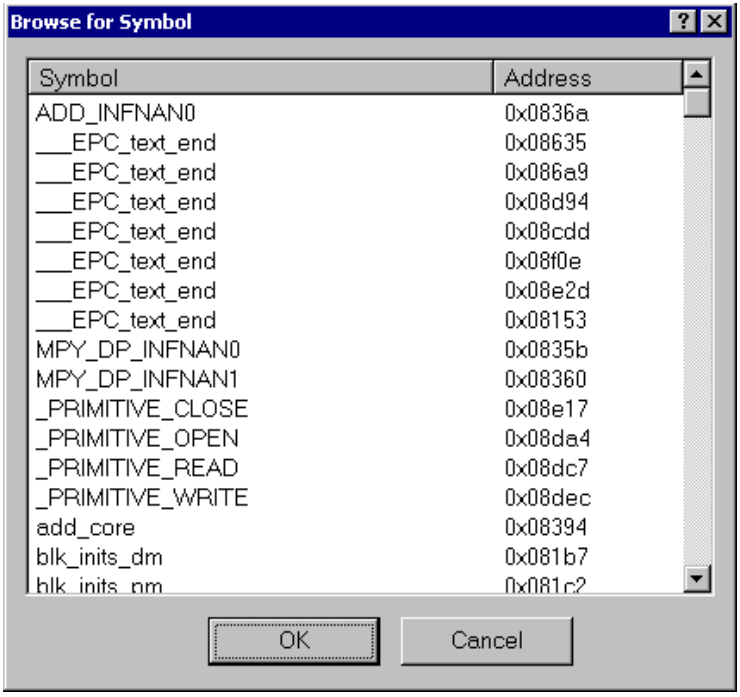

Figure 2-38. Browse Symbols Dialog Box

Managing Memory Segment Properties

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

The **PM/DM** option buttons are grayed out if the current processor architecture has a unified memory space (such as Blackfin DSPs, SHARC DSPs or TigerSHARC 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.

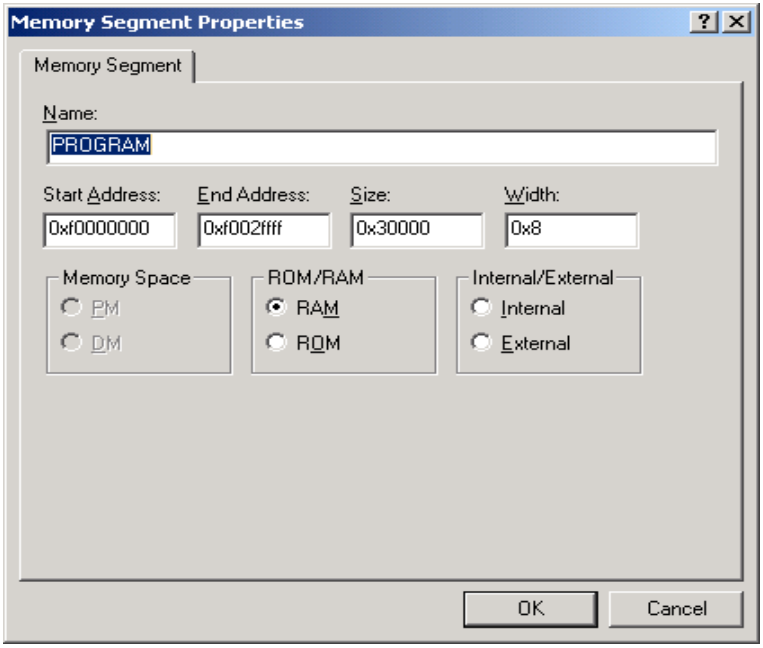

Figure 2-39. Memory Segment Properties – Memory Segment Tab

Managing Output Section Properties

The **Output Section** tab allows you to change the output section's name or to set the overflow. The overflow means objects that do not fit in the current output section can overflow to the specified output section. By default, all objects that do not fit overflow to the specified section, except for objects that are manually pinned to the current output section.

To display the Output Section Properties dialog box:

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

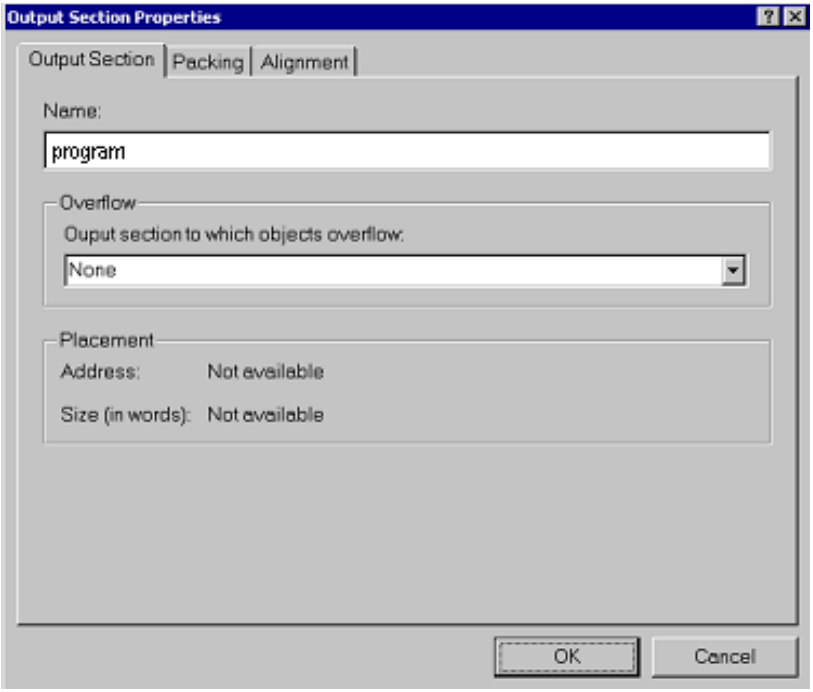

Figure 2-40. Output Section Properties – Output Section Tab

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

You can:

- In **Name**, type a name for the output section.
- In **Overflow**, select an output section into which the selected output section will overflow, or select **None** for no overflow. This setting appears in the **Placement** box.

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

You should also specify **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 display the Packing Properties dialog box:

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

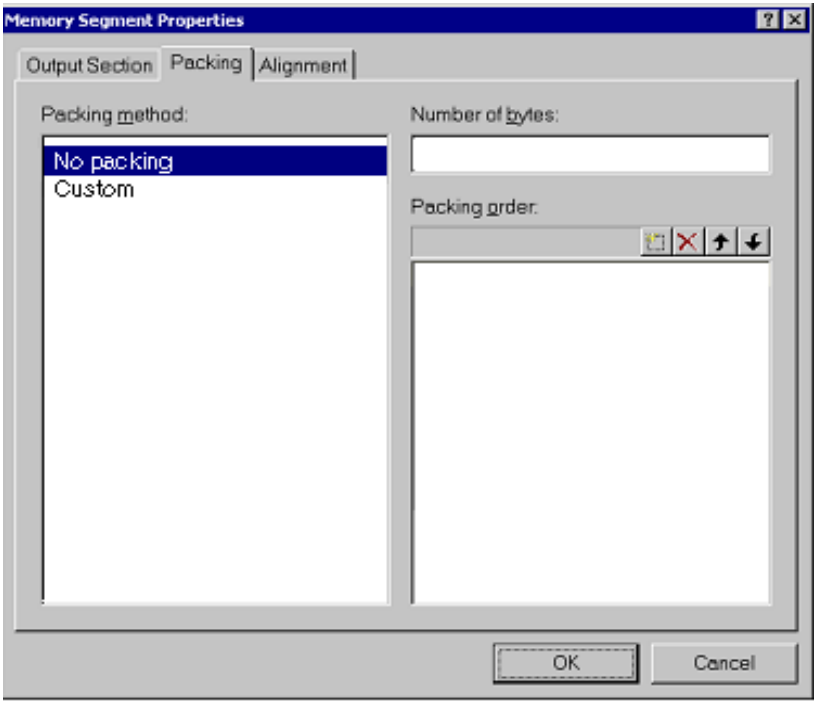

Figure 2-41. Output Section Properties – Generic Packing Tab

Managing Alignment and Fill Properties

The **Alignment** tab allows you to set the alignment and fill values for the output section. When the output section is aligned on an address, the linker fills the gap with zeroes (0), `NOP` instructions, or a specified value.

To display the Alignment Properties dialog box:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties**.
3. Click the **Alignment** tab.

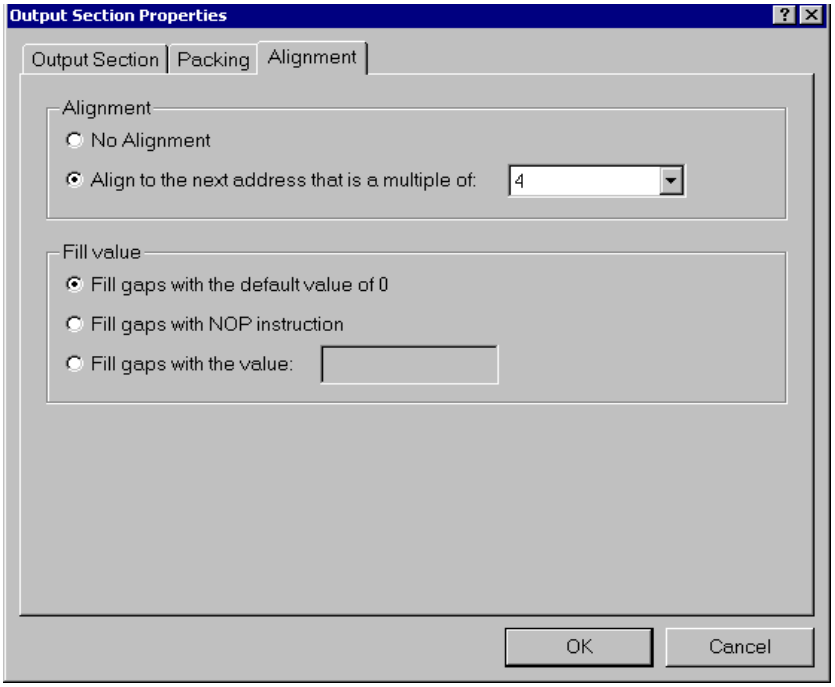

Figure 2-42. Output Section Properties – Alignment Tab

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

To display the Overlay Properties dialog box:

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

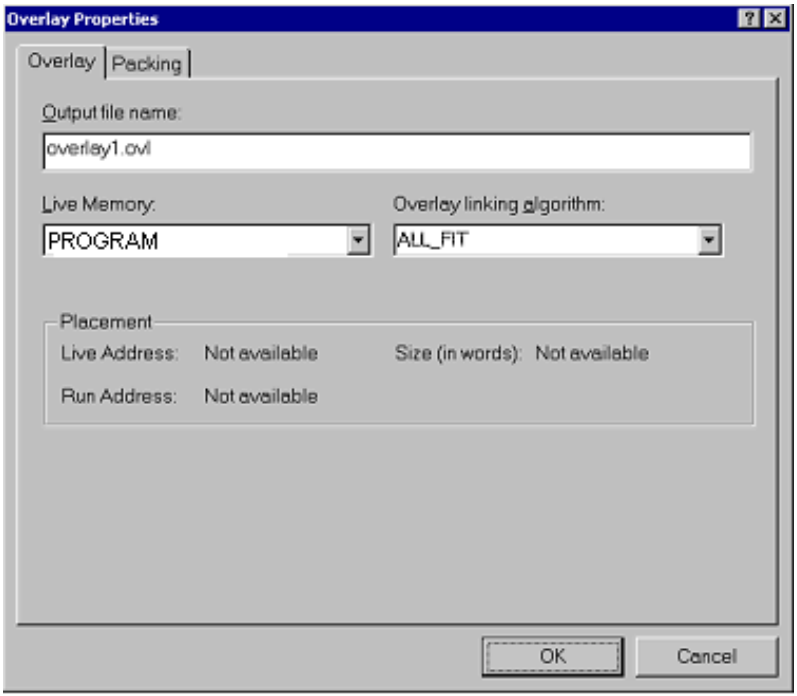

Figure 2-43. Overlay Properties – Overlay Tab

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 shows the only allowed overlay algorithm: `ALL_FIT`. The Expert Linker would not allow you to change that 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. When you click the **Browse** button, the **Browse Symbols** dialog box (see Figure 2-38 on page 2-51) is displayed.

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 is able to show graphically how much space is allocated for your program's heap and stack. For Blackfin DSPs, be aware of these stack/heap restrictions:

- The heap, stack, and system stack must be defined in output sections named `HEAP`, `STACK`, and `SYSSTACK`, respectively.
- The `HEAP`, `STACK`, and `SYSSTACK` must be the only items in those output sections. You cannot have other objects placed into these sections.

Figure 2-44 shows stack/heap output sections in the **Memory Map** pane. Right-click on either of them to display properties.

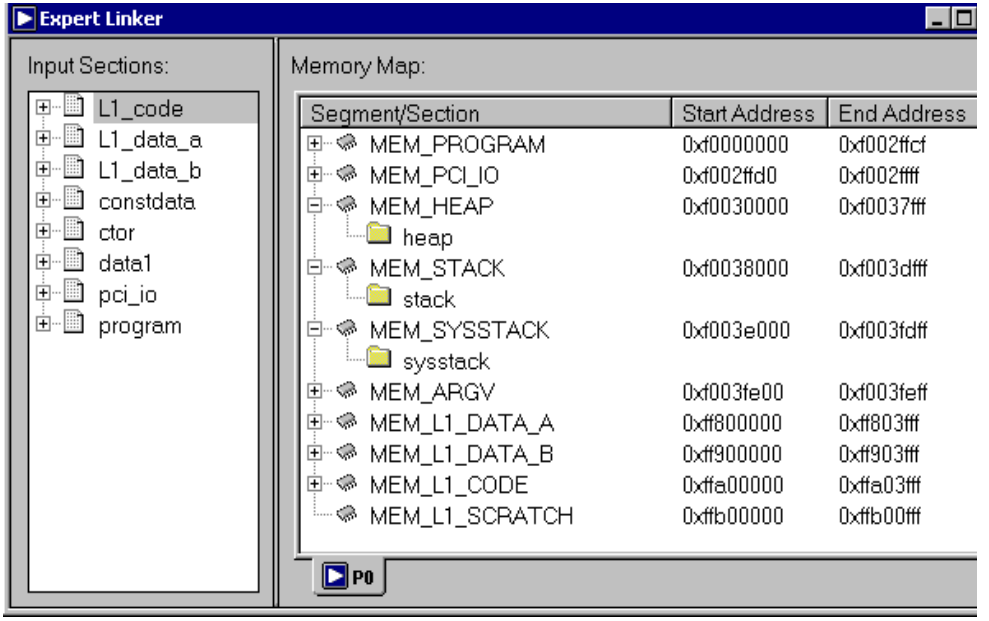

Figure 2-44. Memory Map Window with STACK/HEAP Sections

Use the **Global Properties** page to select **Show stack/heap usage** to graphically display the stack/heap usage in the memory (Figure 2-46).

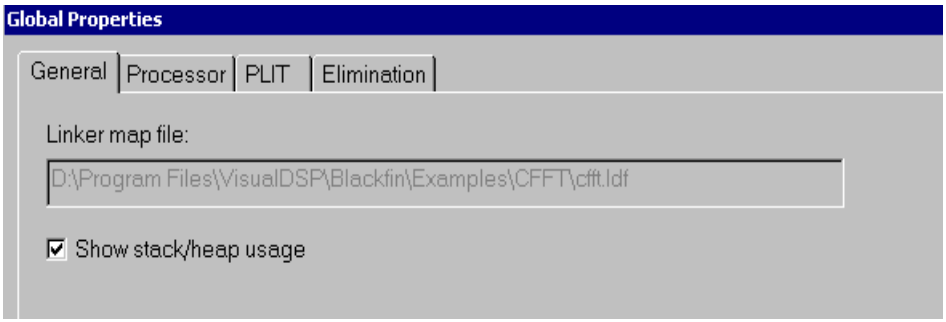

Figure 2-45. Global Properties – Selecting Stack/Heap Usage

The Expert Linker can:

- Locate stacks and heaps and fill them with a marker value.

  This occurs after you load the program into a DSP target. The stacks and heaps are located by their output section names, which may vary across processor families.

- Search the heap and stack for the highest memory locations written to by the DSP program.

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

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

The following code example shows stack and heap sections of an example LDF file.

```
MEMORY /* Define/label system memory  */
{      /* List of global Memory Segments */
   MEM_PROGRAM
   { TYPE(RAM) START(0xF0000000) END(0xF002FFFF) WIDTH(8) }
   MEM_HEAP
   { TYPE(RAM) START(0xF0030000) END(0xF0037FFF) WIDTH(8) }
   MEM_STACK
   { TYPE(RAM) START(0xF0038000) END(0xF003DFFF) WIDTH(8) }
   MEM_SYSSTACK
   { TYPE(RAM) START(0xF003E000) END(0xF003FDFF) WIDTH(8) }
}
PROCESSOR p0 /* The processor in the system */
{
   OUTPUT($COMMAND_LINE_OUTPUT_FILE)
    SECTIONS
   {   /* List of sections for processor P0 */
      program
      {
         INPUT_SECTION_ALIGN(2)
         /* Align all code sections on 2 byte boundary */
         INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
         INPUT_SECTION_ALIGN(1)
         INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
         INPUT_SECTION_ALIGN(1)
         INPUT_SECTIONS(
            $OBJECTS(constdata)
            $LIBRARIES(constdata))
         INPUT_SECTION_ALIGN(1)
         INPUT_SECTIONS( $OBJECTS(ctor) $LIBRARIES(ctor) )
      } >MEM_PROGRAM

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

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

```
        heap
        {   /* Allocate a heap for the application */
           ldf_heap_space = .;
           ldf_heap_end =
              ldf_heap_space + MEMORY_SIZEOF(MEM_HEAP) - 1;
           ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } >MEM_HEAP
    } /* end SECTIONS */
} /* end PROCESSOR p0 */
```

Use the graphical view, chosen via the **View Mode -> Graphical Memory Map** menu selection, to display stack/heap memory map blocks. Figure 2-46 shows a possible memory map after running a Blackfin DSP project program.

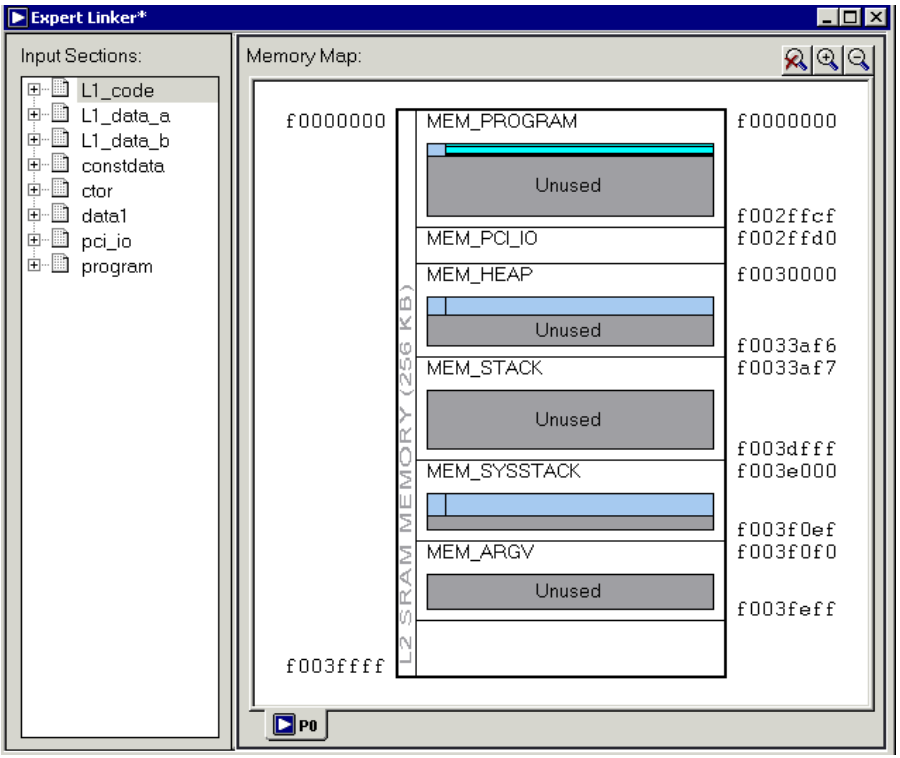

Figure 2-46. Graphical Memory Map Showing Stack/Heap Usage

3 LOADER

The loader (`elfloader.exe`) generates a boot-loadable file for Blackfin ADSP-2153x DSPs by processing executable files (for file name extensions, see Table 3-3 on page 3-15). To generate a loadable file, the loader processes data from a boot kernel file (`.DXE`), the linker's executable file (`.DXE`) and, in some cases, overlay files (`.OVL`).

The ADSP-21535 DSP loader may or may not need a boot kernel file. The ADSP-21532 DSP loader does not need a boot kernel file to generate a loadable file. The ADSP-21535 DSP loader can use the boot ROM only without a boot kernel file. Refer to "Booting Sequence" on page 3-6 for more detailed information.

Once you have fully debugged your program, use the loader to generate a set of boot-loadable files for your target system. You can load the loader output into the a simulator session in the VisualDSP++ debugger; this allows simulation of the boot process as well as of the boot loaded application.

This chapter contains:

- "Loader Guide" on page 3-2

  Provides information on booting and kernel use.

- "Using the Loader" on page 3-13

  Provides reference information on loader commands, loader configurations, and operations.

Loader Guide

The loader converts the DSP executable files produced by the linker into boot-loadable files for an ADSP-2153x DSP. Programs can be automatically downloaded to the internal memory of a DSP after power-up or after a software reset. This process is called *booting*.

This section describes these loader features:

Loader options control how the loader processes your executable files, letting you select features such as loader kernel, boot type, and file format. These options are accessible either via the loader command-line switches or the **Load** page of the **Project Options** dialog box in the VisualDSP++ environment. See "Configuring the Loader" on page 3-19 for more information.

Before selecting options for the loader's operation, you should understand how the loader's features apply to the boot type that you are using. This section describes the boot types for ADSP-2153x DSPs and relates loader features that support these boot types (the booting process).

Hardware Reset and Boot Sources

The ADSP-2153x chip reset is an asynchronous reset event. The RESET input pin must be deasserted to perform a hardware reset.

A hardware-initiated reset results in a system-wide reset that includes both core and peripherals. After the RESET pin is deasserted, the ADSP-2153x DSPs ensure that all asynchronous peripherals have recognized and completed a reset. After the reset, the DSP transitions into the boot mode sequence configured by the BMODE input pins. These pins determine the boot sequence and execution start address (the reset vector).

The ADSP-2153x DSPs can be booted via a variety of methods including execution from an external 16-bit memory, or booting from the on-chip ROM configured to load code from 8-bit FLASH memory or a serial ROM (8- or 16-bit address range).

See appropriate datasheets for more information. Also refer to the *Hardware Reference Manual* of the appropriate processor for more information on system configuration, registers, peripherals, etc.

ADSP-21535 DSP Boot Mode Selection Information

Table 3-1 shows the supported pin configurations for ADSP-21535 System Reset Configuration Register (SYSCR).

If the BMODE [2:0] pins indicate booting from FLASH or Serial ROM, the reset vector points to the start of the internal boot ROM, where a small bootstrap kernel resides. The bootstrap code reads the System Reset Configuration Register to determine the value of the BMODE [2:0] pins, which determine the appropriate boot sequence.

The various configuration parameters are distributed to the appropriate destinations from SYSCR (see Figure 3-1).

Table 3-1. ADSP-21535 Reset Vector Addresses and Boot Mode Selections

Boot Source	BMODE[2:0]	Execution Start Address
Execute from 16-bit external memory (Async Bank 0); Bypass boot ROM	000	0x2000 0000
Use boot ROM to boot from 8-bit FLASH memory	001	0xF000 0000
Use boot ROM to configure and load code from SPI0 serial EEROM (8-bit address range)	010	0xF000 0000
Use boot ROM to configure and load code from SPI0 serial EEROM (16-bit address range)	011	0xF000 0000
Reserved	100 — 111	N/A

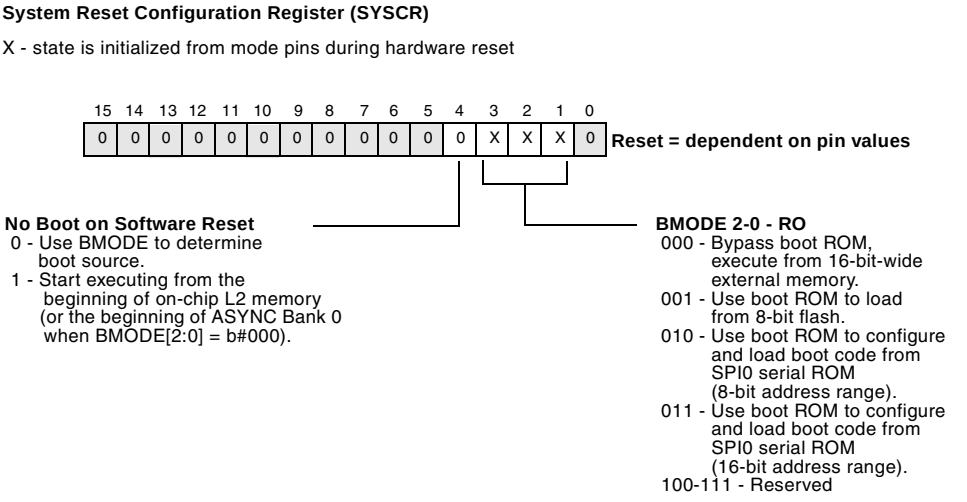

Figure 3-1. ADSP-21535 System Reset Configuration Register (SYSCR)

If the BMODE[2:0] pins indicate to bypass boot ROM, the reset vector points to the start of the external asynchronous memory region (0x2000 0000). In this mode, the internal boot ROM is not used. To support reads from this memory region, the External Bus Interface Unit (EBIU) uses the default external memory configuration that results from hardware reset.

The values of the BMODE[2:0] pins are latched into the System Reset Configuration register (SYSCR) upon the deassertion of the RESET pin. They are made available for software access and modification after the hardware reset sequence. Software can modify only the No Boot on Software Reset bit.

ADSP-21532 DSP Boot Mode Selection Information

Table 3-2 shows the supported pin configurations for ADSP-21532 System Reset Configuration Register (SYSCR).

Table 3-2. ADSP-21532 DSP Reset Vector Addresses and Boot Mode Selections

Boot Source	BMODE[1:0]	Execution Start Address
Execute from 16-bit external memory (Async Bank 0); Bypass boot ROM	00	0x2000 0000
Use boot ROM to boot from 8-bit FLASH memory	01	0xFFA0 0000
Use boot ROM to configure and load code from SPI serial EEROM (8-bit address range)	10	0xFFA0 0000
Use boot ROM to configure and load code from SPI serial EEROM (16-bit address range)	11	0xFFA0 0000

The values of the BMODE [1:0] pins are latched into the System Reset Configuration register upon the deassertion of the RESET pin. They are made available for software access after the hardware reset sequence. Software can modify only the No Boot on Software Reset bit.

The various configuration parameters are distributed to the appropriate destinations from SYSCR (see Figure 3-2).

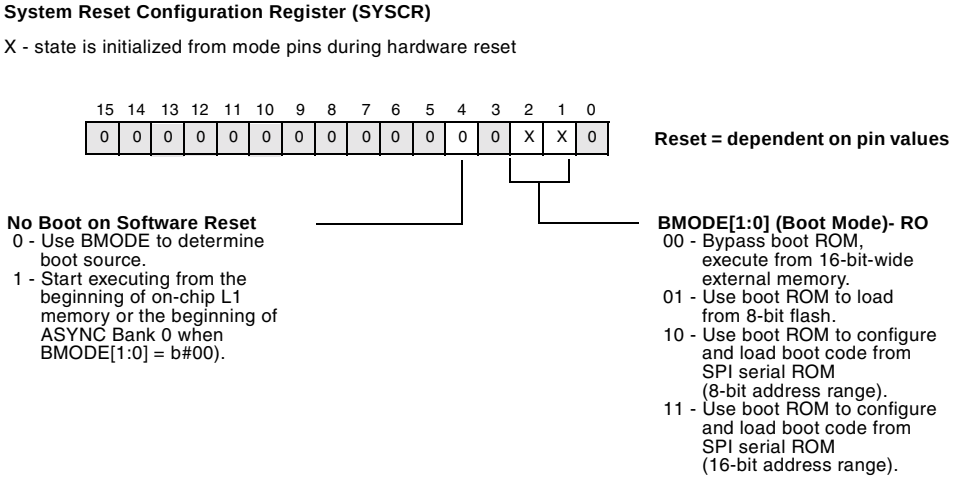

Figure 3-2. ADSP-21532 System Reset Configuration Register (SYSCR)

Booting Sequence

At powerup, after the reset, the DSP transitions into the boot mode sequence configured by the BMODE state. All ADSP-2153x DSPs can be booted from an 8-bit FLASH memory, a serial ROM (8- or 16-bit address range), or from an external 16-bit memory.

The ADSP-2153x DSPs uses one on-chip ROM bootstrap kernel for automatic booting from an external memory device (see “ADSP-21532 DSP Booting” on page 3-7”). In addition, the ADSP-21535 DSP is supported by a variety of loader kernels (see “ADSP-21535 DSP Booting” on page 3-8 on page 3-8).

ADSP-21532 DSP Booting

The internal bootstrap ROM includes a small boot kernel that can either be bypassed or used to load user code from an external memory device (FLASH/PROM memory or SPI EEPROM). The boot kernel reads the `BMODE [1:0]` pin state at reset to determine the download source (see Table 3-2 on page 3-5).

For each boot mode (except the Bypass mode), user code read in from the memory device is placed at the starting location of L1 memory (`0xFFA0 8000`). Additional sections are read into internal memory as specified within headers in the loader file (refer to "ADSP-21532 DSP Boot Stream" on page 3-32).

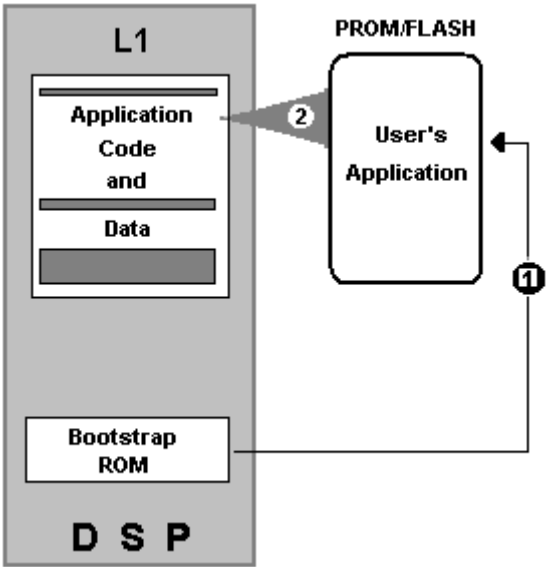

Figure 3-3. Booting from on-chip ROM

The boot kernel terminates the boot process with a jump to the start of the L1 instruction memory space. The processor then begins execution from this address.

When in Bypass mode, the processor is set to execute from 16-bit wide external memory at address `0x2000 0000`. If booting from SPI, the general-purpose flag pin 2 is used as the SPI chip select. This line must be connected for proper operation.

ADSP-21535 DSP Booting

The ADSP-21535 DSPs are supported by two boot kernels:

- The on-chip ROM bootstrap kernel for automatic booting from FLASH/PROM memory or SPI EEPROM.
- The boot kernel that can boot from an external FLASH/PROM memory or SPI EEPROM (boot kernel supplied).

The booting and application loading sequence includes (see Figure 3-4 and Figure 3-5):

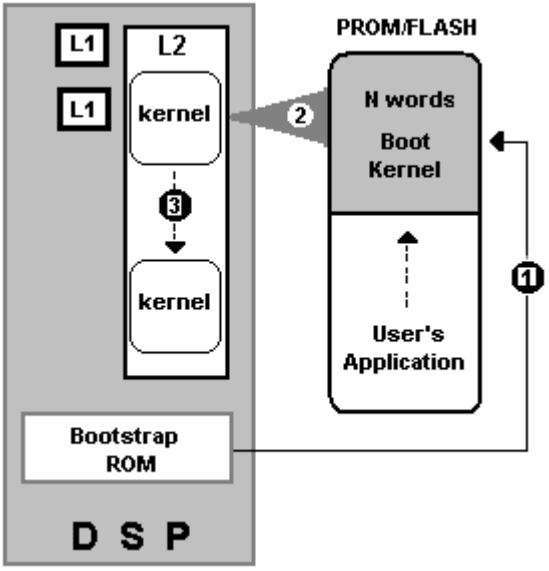

Figure 3-4. Loading Boot Kernel

1. On reset, the DSP downloads *N words* (boot kernel) from the FLASH/PROM memory to address `0xF0000000` in L2 memory. The bootstrap code reads the `BMODE [2:0]` pin state at reset to identify the download source.
2. The DSP begins executing instructions.
3. The boot kernel moves itself to the bottom of L2 memory, to memory block range specified by the upper and lower addresses should be large enough to store the boot kernel.

   The upper address is specified in the `SEG_LDR` input section of the file. When the loader utility parses the `.DXE` file, it extracts this address and places it into the `.LDR` file. This address is then used by the kernel to determine where to move the boot kernel.

   The lower address is `0xF003FFFF`, opening space for application code and data.

   Note: Do not initialize this memory block range for application-code. However, this space becomes available after the boot process is done.
4. The boot kernel imports the application (code and data) from FLASH/PROM or SPI EEPROM into internal (or external) memory of the processor.

   The booting is complete.
5. The boot kernel jumps to the start of the L2 memory space (the beginning of the application) and starts application execution (see Figure 3-5).

   The boot kernel is not needed if the application code resides only

in L2 memory (starting at location 0xF000 0000) and does not have sections that are far apart. In this case, the application has to be loaded from a FLASH/SPI/PROM device, as shown in Figure 3-5.

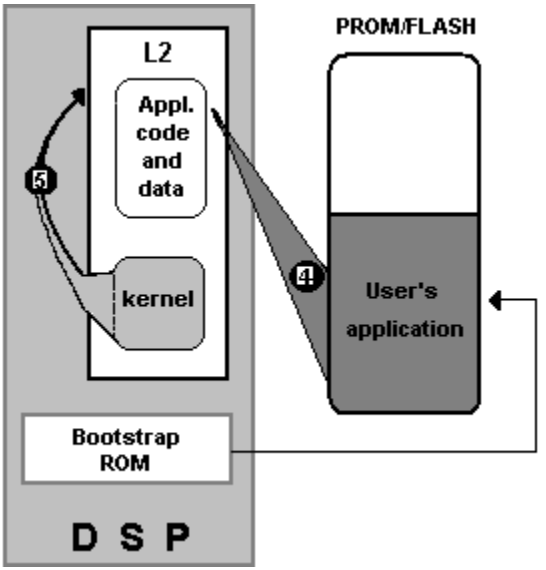

Figure 3-5. Executing the Application

For each boot mode other than BMODE[2:0] = 000, the user code read in from the memory device is placed at the starting location of L2 memory (0xF0000000). The DSP then starts executing the kernel or application code from this address.

In Bypass mode (BMODE[2:0] = 000), the processor is set to execute from 16-bit external memory at address 0x20000000 (ASYNC Bank 0).

Boot Loading and Boot Kernel

The application code must start from 0xF0000000 if the loader output file is using ROM booting only (without a boot kernel). The loader issues an error message if the address of the application starts from any other location other than 0xF0000000 because the output loader file has no address information to be passed and the ROM booting assumes every application code starts from the same memory address 0xF0000000.

Boot kernel loading supports booting from FLASH/PROM or SPI EEPROM external devices. It is much more efficient than bootstrap ROM booting because it provides flexible data placement and much smaller boot image (.LDR file). The boot kernel copies only the necessary data and uses zero fills to reduce the buffer size of the boot image. It also supports a complete memory map of the system including both L1 and L2 memory.

Although the on-chip ROM kernel is included with your DSP, you can use and modify the boot kernel while developing your applications.

Loader Input Files

The input files for the loader are:

- An executable (.DXE) as well as associated overlay files (.OVL) from the linker. The executable file is required for any invocation of the loader. Overlay files are also required if the executable references the overlay files.

  The loader processes the single input file and places the processed data in an output file (.LDR).

- A boot kernel (.DXE). The boot kernel, which may be used with ADSP-21535 DSPs, is not supported by ADSP-21532 DSPs. By default, the loader searches the predetermined directory for the boot kernel and includes this kernel in the loader output file.

The loader does not search for the boot kernel if you decide to boot from the on-chip ROM only (by setting the "-no2kernel" command-line switch).

The boot kernels currently available are:
`535_prom8.dxe`, `535_prom16.dxe` and `535_spi.dxe`.
The `535_prom8.dxe` kernel supports the 8-bit width output for PROM and FLASH, and the `535_prom16.dxe` kernel supports the 16-bit output.

Ensure that a segment named "`code`" is defined in your Linker Description File. The boot loader should have a `code` segment defined in the associated `.LDF` file, and all the code for the kernel is located in this segment. You can find the source for this segment in the default `.LDF` file located in the `...\Blackfin\ldr` directory.

What ELFLOADER.EXE Does

The `elfloader` loader utility is used to convert the input files (`.DXE` and `.OVL`) into a boot image file (`.LDR`).

The loader first converts the boot kernel file (`.DXE`) into the SPI or PROM boot image, adding all the necessary information for the boot kernel loading. Then, the loader converts the application file (`.DXE`) into the boot image (`.LDR`), adding all the necessary information from the boot kernel.


Currently, only the ADSP-21535 DSPs use the boot kernel files.

As a final step, the loader concatenates these two formatted boot images into a single output file (`.LDR`).

Using the Loader

This section describes how to set up and run the loader. It provides reference information on loader command-line switches and VisualDSP++ IDDE option settings for the loader.

When developing a DSP project, you may modify the loader's default option settings in the VisualDSP++ environment. This is done using command-line switches or the **Load** page on the **Project Options** dialog box.

This section contains the following information:

Running the Loader from a Command Line

The following syntax represents the generic single-processor loader command line

```
elfloader -proc ADSP-2153x sourcefile -switch [-switch …]
```

where:

sourcefile — This is the name of the executable file (`.DXE`) to be processed for a single boot-loadable file. A file name can include the drive, directory, file name and file extension.

`-switch` — This is the switch to be processed. The loader has many optional switches that select the operations and modes for the loader.

`-proc ADSP-2153x` — This is the name of a target processor. You must specify your target as `ADSP-21532` or `ADSP-21535` using either the `-Dprocessor` or `-proc` *`processorID`* switch. Refer to Table 3-4 on page 3-15 for switch descriptions.

The loader command line is case sensitive. For example,

```
elfloader -proc ADSP-21535 -bSPI Input.dxe <switches>
```

Some loader switches take a file name as an optional parameter. File searches are important in the loader's process. The loader supports relative and absolute directory names. File searches are performed via:

- *Specified path* — If you include relative or absolute path information with a file name, the loader searches only in that location for the file.
- *Default directory* — If you do not include path information with the file name, the loader searches for the file in the current directory.

When you provide an input or output file name as a command-line parameter:

- Enclose long file names within straight quotes, "`long file name`".
- Append the appropriate file name extension to each file.

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` that refers to them, so the loader can find the these files when generating the `.LDR` file.

The loader follows the conventions for file name extensions that appear in Table 3-3.

Table 3-3. File Name Extension Conventions

Extension	File Description
.dxe	Executable files and boot-kernel files (input files)
.ldr	Loader output files
.knl	Loader output files containing kernel code only

Loader Command-Line Switches

Table 3-4 provides a summary of the loader's command-line switches.

Table 3-4. Loader Command-Line Switches

Switch	Description
-b type	The -b (boot-type) switch directs the loader to prepare a boot-loadable file for a specific booting mode. Valid *type* selections are FLASH, PROM and SPI. If the -b switch does not appear on the command line, the default setting is -bPROM. **Note:** The boot-type that you select with the -b switch must correspond to the boot-kernel that you select with the -l switch and the file format that you select with the -f switch.
-baudrate #	The -baudrate # switch accepts a baud rate for SPI booting only. The valid baud rates and corresponding input values are: 500K = 500 kHz (default) 1M = 1 MHz 2M = 2 MHz Both 8-bit addressable SPI serial PROMs are supported. Boot kernel loading supports an SPI baud rate up to 2 MHz. **Note:** Currently supported only for ADSP-21535 DSPs.

Table 3-4. Loader Command-Line Switches (Cont'd)

Switch	Description
`-f format`	The `-f` (boot file format) switch directs the loader to prepare a boot-loadable file in the specified format. Valid formats depend on the `-b` switch's boot type selection. The boot-type formats are as follows: For `FLASH/PROM` — `hex` (Intel Hex) For `SPI` — `ASCII` or `binary` If the `-f` switch does not appear on the command line, the default format for boot types are: `Hex` for `FLASH/PROM` and `ASCII` for `SPI`.
`-h`	The `-h` (help) switch outputs the list of command-line switches to standard output.
`-HoldTime #`	The `-HoldTime #` switch allows the loader to specify a number of hold-time cycles for FLASH boot. The valid values are from 0 through 3. Default value is 3.
`-kb KernelBootMode`	The `-kb KernelBootMode` switch specifies the boot mode for the boot kernel output file if you select to generate two output files from the loader: one for the boot kernel and another for the user application code. This switch must used in conjunction with the `-o2` switch. If the `-kb KernelBootMode` switch is absent on a command line, the loader generates the file for the boot kernel in the same boot mode as used to output the user application code file. **Note:** Currently supported only for ADSP-21535 DSPs.
`-kf KernelFormat`	The· `-kf KernelFormat` switch specifies the output file format for the boot kernel if you select to output two files from the loader: one for the boot kernel and another for the user application code. This switch must be used in conjunction with the `-o2` switch. If the `-kf KernelFormat` switch is absent on the command line, the loader generates the file for the boot kernel in the same format as for the user application code file. **Note:** Currently supported only for ADSP-21535 DSPs.

Table 3-4. Loader Command-Line Switches (Cont'd)

Switch	Description
`-kWidth #`	The `-kWidth #` switch specifies the width of the boot kernel output file in case there are two output files: one for boot kernel and the other for user application code. The valid values are 8 and 16 for PROM/FLASH and 8 for SPI. If this switch is absent from a command line, the file width is: • The `-width #` switch value when booting from PROM/FLASH, or • The default value (8), when booting from SPI. This switch should be used in conjunction with the `-o2` switch. **Note:** Currently supported only for ADSP-21535 DSPs.
`-l` *`userkernel`*	The `-l` *`userkernel`* (boot-kernel) switch specifies the boot kernel. The loader uses this user-specified kernel and ignores the default boot kernel if this switch is present in a command-line. **Note:** Currently supported only for ADSP-21535 DSPs.
`-no2kernel`	The `-no2kernel` switch produces the output file without the boot kernel, but using the bootstrap code (from the internal boot RAM). In this case, the boot stream generated by the loader is different from the one with the boot kernel loader. **Note:** Currently supported only for ADSP-21535 DSPs.
`-o2`	The `-o2` switch directs the loader to produce two output files: one for the boot kernel and another for the user application code. If you want the output kernel file to have a different format from the application code output file, use the `-kf` switch to specify the format for the output kernel file. **Note:** Currently supported only for ADSP-21535 DSPs.
`-o` *`filename`*	The `-o` (output file) switch specifies the loader's output file. If not specified, the default name is *`sourcefile`*`.ldr`. If you choose to have two output files, one for the boot kernel and another for user application code, both files will have the same root name with different extensions: • The boot kernel file has a `.KNL` extension • The user application code file has an `.LDR` extension

Table 3-4. Loader Command-Line Switches (Cont'd)

Switch	Description
`-proc` *ProcessorID*	The `-proc` (processor) switch selects a target processor: `ADSP-21532` or `ADSP-21535` The processor parameter allows the loader to select a kernel from the default kernel directory based on the specified processor. Alternatively, you can use the `-l` option to specify a particular loader kernel.
`-waits #`	The `-waits #` switch specifies the number of the wait states for external access. The valid inputs are 0 through 15. The wait states apply to the FLASH/PROM mode only. Default is 15. **Note:** Currently supported only for ADSP-21535 DSPs.
`-width #`	The `-width #` switch specifies the word width of the loader output file. The valid values are 8 and 16, depending on the boot mode. The default value is 8 (bits). The switch has no effect on boot kernel code processing. The loader processes the kernel in 8-bit widths regardless of selection of the output data width. For FLASH/PROM booting, the size of the output file depends on the `-width #` switch. For SPI booting, the size of the output .LDR file is the same for both `-width 8` and `-width 16`. The only difference is in the header information. **Note:** Currently supported only for ADSP-21535 DSPs.
`-v`	The `-v` (verbose loader messages) switch outputs status information as the loader processes your files.

Configuring the Loader

You can configure the loader for boot loading and output file generation via the **Load** property page of the **Project Options** dialog box. The **Load** property page may consist of one or two panes opened consequently.

The ADSP-21532 loader does not use the boot kernel and uses only one **Load** page. The ADSP-21535 DSP loader can be booted with or without the boot kernel; it may use two **Load** panes.

You can also configure the loader using the command-line switches. The following sections provides examples of loader configuration using the Load property page:

- "Specifying Basic Loader Settings"
- "Specifying Loader Settings for Boot Kernel Loading" on page 3-21

Specifying Basic Loader Settings

For all ADSP-2153x DSPs, the default **Load** pane is the same. For example, this is the default **Load** pane for booting from SPI EEPROM.

When you open the **Load** page, the default loader settings for the selected DSP are already set (see Figure 3-6 on page 3-20). Using the **Load** pane, you can modify loader settings if you do not wish to use default settings. Using the **Load** page, you can:

1. Select or modify basic settings:
   - Use the **Category** drop-down box to select booting modes. The selections are default **Loader file options** (without boot kernel) and **Boot kernel options** (with boot kernel) described on page 3-21. If you do not use the boot kernel, the second **Load** pane appears with all kernel option fields grayed out.
   - **Boot source**—PROM, FLASH, or SPI

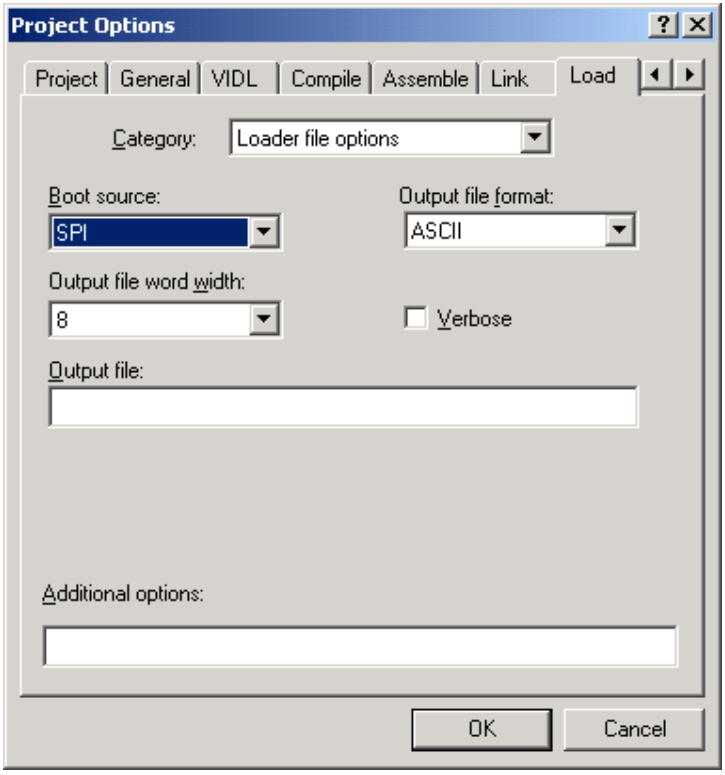

Figure 3-6. Loader Setting Options in Default Load Pane

- **Output file format**—Hex, ASCII, or Binary
- **Output file word width** —8 or 16 bits
- **Verbose**—Generates status information as the loader processes the files
- **Output file**—Enter the name of the loader's output file (`.LDR`).
- **Additional options**—Enter the appropriate file names and options that do not have corresponding controls on the **Load** page are available as loader switches.
  See Table 3-4 on page 3-15 for more information).

2. If you are satisfied with default settings, do not change any settings and click **OK** to complete the loader setup. The loader will be set up to produce one output file (.LDR).

Specifying Loader Settings for Boot Kernel Loading

Figure 3-7 shows, as an example, the second, boot kernel-oriented, **Load** pane that supports booting from SPI EEPROM.

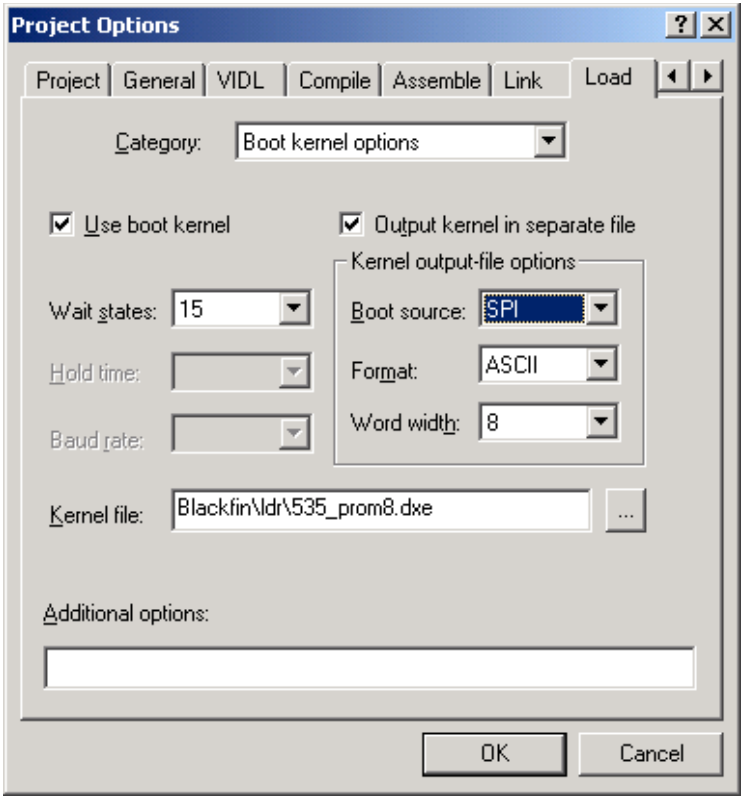

Figure 3-7. Setting for Boot Loading using Boot Kernel

Whence the first **Load** pane allows you to use default settings, the second **Load** pane provides additional boot kernel and output settings.

To configure the loader for boot loading and output file generation using the boot kernel:

1. Use the default **Load** property pane to set up basic booting options.
2. Select **Boot kernel options** from the **Category** drop-down box to open the second **Load** property pane with boot kernel settings.
3. Select **Use boot kernel**.
4. If you want to produce two output files (boot kernel file and application code file), select the **Output kernel in separate file** check box.
5. You can also enter the **Kernel file** (`.DXE`).

   The following boot kernels are currently available: `535_prom8.dxe`, `535_prom16.dxe` and `535_spi.dxe`.

   The `535_prom8.dxe` kernel supports the 8-bit width output for PROM and FLASH, and the `535_prom16.dxe` kernel supports the 16-bit output.
6. Select boot kernel output file parameters, such as **Boot source**, **Format**, **Word width**, etc.
7. Click **OK** to complete the loader setup.

   The loader will be set up to produce one output file (`.LDR`) or two output files (`.LDR` and `.KLN`) depending on your selections.

Loader Boot Streams

The loader generates the boot stream and places the boot stream in one or two output files. The ADSP-21532 DSP loader generates only one output file. The ADSP-21535 DSP loader may generate one or two output files. The loader prepares the boot stream in such a way that the boot kernel can correctly load the application code and data to the DSP memory; therefore, the boot stream contains not only the user application code and kernel code but also overhead information that is used by the boot kernel.

The ADSP-2153x DSP supports the following boot stream structures:

- "ADSP-21535 DSP Boot Stream with Boot Kernel"
- "ADSP-21535 DSP Boot Stream without Boot Kernel"
- "ADSP-21532 DSP Boot Stream"

The following application code example for ADSP-21535 DSPs illustrates the bootstream structure as shown in Figure 3-8.

```
.SECTION program1;
r0=1234;                    // header1 describes this entry
   .
   .
   .
.SECTION data2;
.var temp(1000)=0,0,...;    // header2 describes this entry

.SECTION data3;
.var array[ ]=1,2,3,...;    // header3 describes this entry
```

ADSP-21535 DSP Boot Stream with Boot Kernel

Figure 3-8 illustrates the information included in the boot stream that uses the boot kernel.

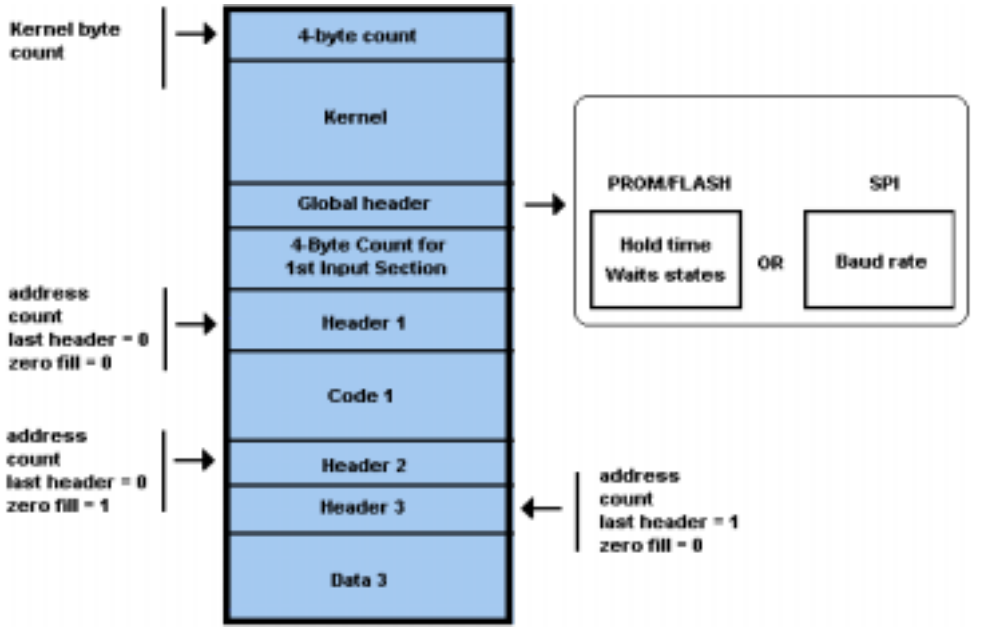

Figure 3-8. Bootstream Output for ADSP-21535 DSP

The boot stream includes:

- The *4-byte count* for the boot kernel code, which stores the total byte count of the boot kernel code. The byte count does not include padded zeros. This information is required to import the boot kernel from the internal boot ROM.
- The boot kernel.

- The *4-byte global header*, which contains information (hold time, wait states, baud rate, etc.). The boot kernel sets the booting condition based on the information given in the global header, code, and user application code. The kernel needs this information to set up the processor correctly.
- The *input section byte counter*, which contains byte count of the first input section (header s and data).
- User's application code (Code 1) and data block (or data blocks) each with a 10-byte block header (header1, header2, etc.). In each header, the first four bytes store the start address of that data block, the second four bytes store the byte count of that data block, and the last two bytes store the flags indicating whether the data block is a last (final) block or a zero block (zero fill).

  If it is a zero fill, the block contains only a header without data. Thus, there is no Data 2 block in Figure 3-8. The boot kernel populates the specified memory block with a number of zeros given by the byte count starting at the given start address from the block header. If it is the last data block, the kernel stops loading the data after the data of this block is loaded, and the user application code starts to execute.

The loader accepts the input file (`.DXE`) and processes it. The loader counts the bytes (including these of the instructions, data, and the block headers in the boot stream) and places the byte count number on the top of the boot stream for each of the input section.

A 4-byte space in a boot stream is allocated to hold the byte count number for each input section. This allows the boot kernel to selectively load the code from one or more input sections or switch the code from one input section to another at runtime by tracing the byte count of each input section and determining the size in bytes of each input section.

The loader is set to produce a zero block for the L1 memory with its byte count always in a multiple of 8 (for both 8-bit and 16-bit outputs). The loader also fills a non-zero block with zeros to make its byte count be a multiple of 8 (for 8-bit output only).

To support this feature, the loader is doing the following.

- Sorts all the input sections from an input `.DXE` file before processing the data in the sections to avoid overwriting because of padding. The sorting is used for all types of the processes.
- Split a `struct` directive in the data-linked list with a value of zero into two — one with its byte count being a multiple of 8 and the other with its byte count less than 8. This step helps to avoid any padding to a zero block and to minimize overall padding zeros.
- Fill a non-zero block in a "non-multiple-of-8-byte" count with zeros to make its byte count be a multiple of 8.

8-Bit Data Structure

The 8-bit boot stream format is:

D7	D0
LSB of boot kernel byte count	
Bits 8-15 of boot kernel byte count	
Bits 16-23 of boot kernel byte count	
MSB of boot kernel byte count	
Byte3 of boot kernel	
Byte2 of boot kernel	
Byte1 of boot kernel	
Byte0 of boot kernel	
LSB of boot kernel start address	
Bits 8-15 of boot kernel start address	
Bits 16-23 of boot kernel start address	
MSB of boot kernel start address	

:
LSB of the Global Header
Bits 8-15 of the Global Header
Bits 16-23 of the Global Header
MSB of the Global Header
LSB of the 1st input (.dxe) section
Bits 8-15 of the 1st input (.dxe) section
Bits 16-23 of the 1st input (.dxe) section
Bits 24-31 of the 1st input (.dxe) section
LSB of the address of the 1st data block
Bits 8-15 of the address of the 1st data block
Bits 16-23 of the address of the 1st data block
MSB of the address of the 1st data block
LSB of the byte count of the 1st data block
Bits 8-15 of the byte count of the 1st data block
Bits 16-23 of the byte count of the 1st data block
MSB of the byte count of the 1st data block
LSB of the 1st data block flags
MSB of the flags of the 1st data block
Byte3 of the 1st data block
Byte2 of the 1st data block
Byte1 of the 1st data block
Byte0 of the 1st data block
:
LSB of the address of the nth data block
Bits 8-15 of the address of the nth data block
Bits 16-23 of the address of the nth data block
MSB of the address of the nth data block
LSB of the byte count of the nth data block
Bits 8-15 of the byte count of the nth data block
Bits 16-23 of the byte count of the nth data block
MSB of the byte count of the nth data block

LSB of the flags of the nth data block
MSB of the flags of the nth data block
Byte1 of the nth data block
Byte0 of the nth data block
:
:

16-Bit Data Structure

There are two types of the data in a boot stream if the word width is selected to be 16 bit.

- The first data type is the boot kernel, which always has an 8-bit width (with an 8-bit padded zero) in the loader output file.
- The second data type, which includes the Global Header and the user application code, has a 16-bit width.

Although the loader pads a zero following each byte in a 16-bit wide boot stream, the image in the FLASH memory is reversed from the boot stream in the loader file.

The 16-bit wide data image format in the FLASH memory is:

D15 D8	D7 D0
00	LSB of boot kernel byte count
00	Bits 8-15 of boot kernel byte count
00	Bits 16-23 of boot kernel byte count
00	MSB of boot kernel byte count
00	Byte3 of boot kernel (24-31)
00	Byte2 of boot kernel (16-23)
00	Byte1 of boot kernel (8-15)
00	Byte0 of boot kernel (0-7)
	:
00	LSB of boot kernel start address

00	Bits 8-15 of boot kernel start address
00	Bits 16-23 of boot kernel start address
00	MSB of boot kernel start address
Bits 0-15 of the Global Header	
Bits 16-31 of the Global Header	
Bits 0-15 of the byte count of the 1st input section	
Bits 16-31 of the byte count of the 1st input section	
Bits 0-15 of the address of the 1st data block	
Bits 16-31 of the address of the 1st data block	
Bits 0-15 of the byte count of the 1st data block	
Bits 16-31 of the byte count of the 1st data block	
Flag of the 1st data block	
Byte3 of the 1st data block |	Byte2 of the 1st data block (16-31)
Byte1 of the 1st data block |	Byte0 of the 1st data block (0 -23)
:	
Bits 0-15 of the address of the nth data block	
Bits 16-31 of the address of the nth data block	
Bits 0-15 of the byte count of the nth data block	
Bits 16-31 of the byte count of the nth data block	
Flag of the nth data block	
Byte1 of the nth data block |	Byte0 of the nth data block (0-15)
:	
:	

ADSP-21535 DSP Boot Stream without Boot Kernel

The loader accepts a single input file (.DXE file). The boot stream contains:

- The 4-byte header that provides the total word (byte) count for the data block.
- Data without a global header and data block headers. There is only one data block following the 4-byte header which contains the total byte count of the data block. This type of the data format is generally used for small applications.

The loader has to fill any address gaps between data segments with zeros to make the data address in the loader file contiguous. The loader needs to calculate the number of the bytes to be filled in a gap.

8-Bit Data Structure

The 8-bit boot stream format is:

D7 D0
Byte0 (0-7 of the data byte count)
Byte1 (8-15 of the data byte count)
Byte2 (16-23 of the data byte count)
Byte3 (21-31 of the data byte count)
Byte1 (8-15 of 16-bit data)
Byte0 (0-7 of 16-bit data)
Byte3 (24-31 of 32-bit data)
Byte2 (16-23 of 32-bit data)
Byte1 (8-16 of 32-bit data)
Byte0 (0-7 of 32-bit data)
:
:

16-Bit Data Structure

The 16-bit boot stream format used with booting from the FLASH memory is:

D15 D8	D7 D0
Byte0 (0-7 of the data byte count)	00
Byte1 (8-15 of the data byte count)	00
Byte2 (16-23 of the data byte count)	00
Byte3 (24-31 of the data byte count)	00
Byte1 (8-15 of 16-bit data)	00
Byte0 (0-7 of 16-bit data)	00
Byte3 (24-31 of 32-bit data)	00
Byte2 (16-23 of 32-bit data)	00
Byte1 (8-16 of 32-bit data)	00
Byte0 (0-7 of 32-bit data)	00
:	:
:	:

ADSP-21532 DSP Boot Stream

The ADSP-21532 DSP boot stream is similar to the boot stream that uses the boot kernel with ADSP-21535 DSPs. However, since this DSP does not employ a boot kernel, its boot stream does not include the kernel code and the associated 4-byte header on the top of the kernel code. There is also no 4-byte global header. The ADSP-21532 DSP boot stream contains:

- The input section byte counter which stores four bytes.
- A number of data block headers, similar to the ones used with ADSP21535 DSP.s
- A number of the data blocks, which have the same data format as data blocks used with ADSP-21535 DSPs.

8-Bit Data Structure

The 8-bit boot stream format is:

D7 D0
LSB of the 1st input (.dxe) section
Bits 8-15 of the 1st input (.dxe) section
Bits 16-23 of the 1st input (.dxe) section
Bits 24-31 of the 1st input (.dxe) section
LSB of the address of the 1st data block
Bits 8-15 of the address of the 1st data block
Bits 16-23 of the address of the 1st data block
MSB of the address of the 1st data block
LSB of the byte count of the 1st data block
Bits 8-15 of the byte count of the 1st data block
Bits 16-23 of the byte count of the 1st data block
MSB of the byte count of the 1st data block
LSB of the 1st block flags

MSB of the 1st block flags
Byte3 of the 1st data block
Byte2 of the 1st data block
Byte1 of the 1st data block
Byte0 of the 1st data block
:
LSB of the address of the nth data block
Bits 8-15 of the address of the nth data block
Bits 16-23 of the address of the nth data block
MSB of the address of the nth data block
LSB of the byte count of the nth data block
Bits 8-15 of the byte count of the nth data block
Bits 16-23 of the byte count of the nth data block
MSB of the byte count of the nth data block
LSB of the flags of the nth data block
MSB of the flags of the nth data block
Byte1 of the nth data block
Byte0 of the nth data block
:

Loader Output Files and Formats

The Blackfin DSP loader produces the output file(s) per target DSP:

- For **ADSP-21532 DSPs**, the loader generates one output file (`.LDR`). The ADSP-21532 DSP loader does not support any boot kernel options available for the ADSP-21535 DSP loader.
- For **ADSP-21535 DSPs**, the loader generates one or two output files depending on the switches selected on the loader's command line.

The loader output file formats are:

Boot Mode	Data Width (bits)	File Format
FLASH/PROM	8 (default), 16	Intel Hex
SPI	8 (default), 16	ASCII (default) Binary

By default, the ADSP-21535 DSP loader generates one output file, which includes both the boot kernel and user application code. However, you have an option to generate two separate output files—one file for the boot kernel and another for the user application code. You can even choose different boot modes and file formats for these two output files (`.LDR` and `.KLN`).

With ADSP-21535 DSPs, you can choose to produce two separate output files. You can specify the loader's output by:

- Using the `-o` (output file) switch (described in Table 3-4 on page 3-15), or
- Selecting the **Output kernel in separate file** check box on the **Load** page (see Figure 3-7 on page 3-21)

If not specified, the default name for the loader's output is specified by the *`sourcefile.ldr`* file. If you choose to have two output files—one for the kernel and another for user application code, both files will have the same root name with different extensions:

- The output file for the kernel has the extension of `.KNL`.
- The output file for the user application code has the extension of `.LDR`.

Use the `-no2kernel` switch (described in Table 3-4 on page 3-15) to select automatic bootstrap (ROM) loading only, therefore, selecting *not* to use or include the boot kernel in the output file. If this switch is set, the loader generates the output file with the extension of `.LDR`. You can do the same by *not* choosing **Boot kernel options** in the default **Load** pane.

Rebuilding the Boot Kernel

The boot kernel (used in the project definition) can be rebuild from the VisualDSP++ IDDE. If VisualDSP++ is not used, the following command lines demonstrate how to rebuild default boot kernels for the ADSP-2153x DSPs.

For example, the default boot kernel for FLASH booting is `FLASH.asm`. After copying the default file to `my_FLASH.asm` and modifying it to suit your system, use the following command lines to rebuild the boot kernel:

```
easmblkfn -adsp21535 my_FLASH.asm
```

or

```
easmblkfn -proc ADSP21535 my_FLASH.asm
linker -T 21535ldr.ldf my_FLASH.doj
```

:

4 ARCHIVER

The VisualDSP++ archiver[1], `elfar.exe`, combines object files[2] into archive (library) files, which can serve as a reusable resource for code development. The linker can rapidly search the archive files for routines (archive members) referred to in other objects and link these routines into your executable program. You can run the archiver from a command line, or produce an archive file as the output of a VisualDSP++ project.

This chapter contains the following information on the archiver:

- "Archiver Guide" on page 4-2 introduces the archiver's functions
- "Archiver Command-Line Reference" on page 4-6 reference information on archiver operations

[1] Also called "librarian."

[2] The archiver is general-purpose: it can combine (and extract) arbitrary files. This manual refers to DSP object files because they are relevant to DSP code development.

Archiver Guide

The `elfar.exe` combines and indexes object (or any other) files, producing a searchable archive file. It can perform the following operations, as directed by options on its command line:

- Append one or more object files to an existing archive file
- Create an archive file from a list of object files.
- Delete file(s) from an archive file.
- Extract file(s) from an archive file.
- List the filename contents of an existing archive file (to `stdout`)
- Replace file(s) in an existing archive file

The archiver can only run one of these operations at a time. However, for commands that can take a list of files as arguments, it can input a file containing the names of object files (separated by white space) which makes long lists easily manageable.

Creating an Archive From VisualDSP++

Within the VisualDSP++ IDDE, you can choose to create an archive file as your project's output. To do so, specify **Archive file** as the target type in the project's property page. That property page appears when you create a new project, or click **Project Options** menu option on an existing project.

VisualDSP++ writes its output to `<projectname>.DLB`. To modify or list the contents of an archive file, or perform any other operations on it, you must use the archiver from the command line.

> **Syntax:** `elfar -[a|c|d|e|p|r][-v][-i filename] library_file object_file ..`

Filename Conventions

To maintain consistency within your code, it is recommend you use the conventions in Table 4-1. (Note that VisualDSP++ always writes out `<projectname>.DLB` when it creates an archive.)

Table 4-1. File Name Extension Conventions

Extension	File Description
`.dlb`	Archive file
`.doj`	Object file -- input to archiver
`.txt`	Archiver `-i` switch input (list file)

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, namely:

- Writing *archive routines*, functions which can be called from other programs.
- Using archive routines: accessing archived functions in your code.

This section describes both tasks.

Writing Archive Routines: Creating Entry Points

An archive routine is simply a routine in your code that can be accessed by other programs. Each such routine must have a globally visible start label (*entry point*). Any code accessing that routine must know the entry point's name and declare it as an external variable in the calling code.

To create these entry points, use the following steps:

1. Declare the start label of each routine as a global symbol with the assembler's `.GLOBAL` directive. That is the entry point.

   The following code fragment has two entry points, `dIriir` and `FAE`.

   ```
   ...
   .global dIriir;
   .section data1;
   .byte2 FAE = 0x1234,0x4321;

.section program;
.global FAE;
dIriir: R0=N-2;
P2 = FAE;
   ```

2. Assemble and archive the code containing these routines. You can do so in two ways.

   - Direct the VisualDSP++ IDDE to produce an archive (see above on how to do so). When you build the project, your object code containing the entry points is packaged in `<projectname>.DLB`. [You can extract the object (`.DOJ`) whenever you want, for example to incorporate it in another project.]
   - If you create executable or unlinked object files from the IDDE, you can archive them afterward from the command line. The result is the same.

Using Archive Routines

Programs that call an archive routine must define the routine's start label as an external label with the assembler's `.EXTERN` directive. When you link the program, you specify the archive file (`.DLB`) to the linker, along with the names of the object files 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. The advantage of linking archives over using individual object files is that the linker can search archives faster, and you do not need to enter all of the file names, just the archive name.

In the following example, the archiver creates the `filter.dlb` archive, 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`, `sum.doj`, and `graph.doj`; uses the default linker description file, `062.ldf`; and creates the executable file (`main.dxe`):

```
linker -DADSP-21535 main.doj sum.doj graph.doj filter.dlb
```

Assuming that one or more library routines from `filter.dlb` are called from one or more of the object files, the linker searches the archive, extracts the required routines, and links the routines into the executable.

Archiver Command-Line Reference

The archiver processes object files into an archive file. Its output is an archive file with the file name extension `.DLB`[1]. It can also append, delete, extract, or replace member files in an archive, as well as list them to `stdout`.

This section provides reference information on the archiver command line and linking.

Running the Archiver

Use the following syntax to run `elfar` from the archiver command line. Table 4-2 describes each switch.

```
elfar -[a|c|d|e|p|r][-v][-i filename] library_file object_file ..
```

This command line is subject to the following constraints:

- You may select exactly one action switch (`a`, `c`, `d`, `e`, `p`, `r`) in a single command.
- The verbose operation switch `-v`, must not be in a position where it can be mistaken for an object file, meaning it may not follow the *`library_file`* on append or create.
- The file include switch, `-i`, must immediately precede the include file name.
- Use the archive filename first, following the switches. `[-i` and `-v` are not operational switches, and can appear later.]
- Enclose file names containing white space or colons within straight quotes.

---

[1] `.DLB` is the default extension for archive files,
`.DOJ` is the default extension for object files. You can use any name.

- Append the appropriate file name extension to each file. The archiver assumes nothing, and will not do it for you.
- You cannot use “wild cards”. To perform an archive operation on a list of member files, write the list into a text file and use it as input to the command line (with the `-i` switch).
- *`object_file`* — The name of an object file (`.DOJ`) to be added, removed, or replaced in the *`library_file`*.

  Note that the archiver’s `-i` switch lets you input a list of members from a text file, instead of listing all the members on the command line. Also note that when you use the archiver’s `-p` switch, you do not need to identify any members on the command line.

The archiver’s command line is *not* case-sensitive. For example, the following command line

```
elfar -v -c my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

runs the archiver with

`-v` — Selects verbose mode for the archiver

`-c my_lib.dlb` — Creates an archive file named `my_lib.dlb`

`fft.doj sin.doj cos.doj tan.doj` — Uses object files that the archiver puts in the archive file

The archiver takes file names as parameters. Table 4-1 on page 4-3 lists the relevant types of files, names, and extensions normally used in VisualDSP++.

Archiver File Search

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 only searches 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.

Command-Line Switch Descriptions

When you provide an input or output file name as a command line parameter, use the following guidelines:

Table 4-2 describes each archiver parameter and switch[1].

Table 4-2. Archiver Command-Line Switches

Switch	Description
archive-file	The archive that the archiver modifies. This parameter appears after the switch.
member-file	One or more object files that the archiver uses when modifying the archive. This parameter appears after *archive-file*. You can use the -i switch to input object file names as a list.
[-a]	Append one or more *member files* to the end of the named archive file.

[1] The switches in Table 4-2 must appear before the archive-file name on the command line, except that the -i switch appears in place of the member-files. Items shown in [ ] are optional. Items shown in *italics* are user defined.

Table 4-2. Archiver Command-Line Switches (Cont'd)

Switch	Description
[-c]	Create a new *archive-file* containing the *member-files* on the command line.
[-d]	Delete one or more *member-files* from the selected *archive-file*.
[-i] <*list file*>	Use *list-file*, containing*member-file* names, as input. This file lists the *member-files* to add or modify in the selected *archive-file* (.DLB).
[-p]	The -p (print archive contents) switch directs the archiver to print to standard output a list of the *member-files* (.DOJ) in the selected *archive-file* (.DLB).
[-r]	Replace the named archive file in the library.
[-v]	The -v (verbose archiver messages) switch directs the archiver to output status information as the archiver processes your files.

A FILE FORMATS

The development tools support many file formats, in some cases several for each development tool. This appendix describes the formats of files that you prepare as input for the tools and points out features of files produced by the tools.

The three types of file formats that you can learn about in this appendix are:

- "Source Files" on page A-2
- "Build (Processed) Files" on page A-6
- "Debugger Files" on page A-12

Most of the development tools use industry standard file formats. Sources that describe these formats appear in "Format References" on page A-13.

Source Files

This section describes the following types of input file formats:

C/C++ Source Files

These are text files (with such extensions as `.C`, `.CPP`, `.CXX`, etc.) containing C/C++ code, compiler directives, possibly a mixture of assembler code and directives, and (typically) preprocessor commands.

Several "dialects" of C code are supported: pure (portable) ANSI C, and at least two subtypes[1] of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives, used by the linker to structure and place executable files.

For information on using the C/C++ compiler and associated tools, as well as a definition of ADI extensions to ANSI C, see the *VisualDSP++ 3.0 C/C++ Compiler & Library Manual for Blackfin DSPs*.

For information on specifying the C dialect and general C code handling within VisualDSP++, see the *VisualDSP++ 3.0 User's Guide for Blackfin DSPs* and VisualDSP++ online Help.

[1] With and without builtin 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 the *Instruction Set Reference* manual for the corresponding DSP.

The DSP's instruction set is supplemented with assembler directives. Preprocessor commands control macro processing and conditional assembly or compilation.

For information on the assembler and preprocessor, see the *VisualDSP++ Assembler and Preprocessor Manual for Blackfin DSPs*.

Assembly Initialization Data Files (.DAT)

These are text files containing fixed-point data. These files can provide the initialization data for an assembler `.VAR` directive or serve in other tool operations. When a `.VAR` directive uses a `.DAT` file for initialization data, the assembler reads the data file and initializes the buffer in the output object file (`.DOJ`). Data files have one data value per line and may have any number of lines.

The `.DAT` extension is merely explanatory or mnemonic. A directive to `#include <file>` can of course take any file name (or extension) as an argument.

Fixed-point values (integers) in data files may be signed, and they may be decimal-, hexadecimal-, octal-, or binary-base values. The assembler uses the prefix conventions in Table A-1 for identifying a fixed-point value's numeric base.

For all numeric bases, the assembler uses 16-bit words for data storage; 24-bit data is for the program code only. The largest word in the buffer determines the size for all words in the buffer. If you have some 8-bit data

Table A-1. Fixed-Point Values in Data Files

Convention	Description
0x*number*, H*number* h*number*	An "0x", "H#" or "h#" prefix indicates a hexadecimal *number*
number, D*number* d*number*	A "#D", "#d", or no prefix indicates a decimal *number*
O*number*	An "#O" or "#o" prefix indicates an octal *number*
B*number* b*number*	A "B#" or "b#" prefix indicates a binary *number*

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

Header Files (.H)

Header files are text files that contain macros or other preprocessor commands that the preprocessor substitutes into source files. For information on macros or other preprocessor commands, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for Blackfin DSP*.

Linker Description Files (.LDF)

The linker's .LDF files are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language see "Linker Guide" on page 1-30.

Linker Command-Line Files (.TXT)

The linker's command-line files are ASCII text files that contain command-line input for the linker. For more information on the linkers command line, see "Linker Command-Line Reference" on page 1-95.

Build (Processed) Files

Build files are files that the development tools produce when building your VisualDSP++ project. This section describes the following types of build file formats:

Assembler Object Files (.DOJ)

The assembler's output object files are binary, Executable-Linkable-File (ELF) format. Object files contain relocatable code and debugging information for your program's segments. The linker processes your object files into an executable file. For information on the ELF format used for object files, see the "Format References" on page A-13.

Archiver Archive Files (.DLB)

The archiver's output archive files are binary, Executable-Linkable-File (ELF) format. Archive files contain one or more object files, called Archive Elements. The linker searches through archive files for any archive elements that your code uses. For information on the ELF format used for executable files, see the "Format References" on page A-13.

Linker Executable Files (.DXE, .SM, .OVL, .DLB)

The linker's output executable files are binary, Executable-Linkable-File (ELF) format. Executable files contain your program's code and debugging information. The linker may fully or partially resolve addresses in executable files, depending on the options given on the linker's command line and in your linker description file. For information on the ELF format used for executable files, see the TIS Committee texts cited in "Format References" on page A-13.

Linker Memory Map Files (.MAP)

The linker's memory map files are ASCII text files that contain memory and symbol information for your executable file(s). The map contains a summary of memory that you define with `MEMORY{}` commands in your linker description file, and provides a listing of the absolute addresses of all symbols.

Loader Hex Format Files (.LDR)

The loader's output Hex-format files are in ASCII, Intel Hex-32 format. Hex files from the loader support 8-bit wide PROMs. The files are used with an industry-standard PROM programmer to program memory devices for your hardware system. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel Hex-32 format appears in the loader's output file. Each line in the Intel Hex-32 file contains either a data record, an extended linear address record or the end of file record:

```
:020000040010E9                          extended linear address record
:0A0004003C40343434261422260850          data record
:00000401FB                              end of file record
```

The extended linear address record is used because a data record has only a 4-character (16-bit) address field, but the ADSP-2153x processors require 32 bits to address data memory and 24 bits to address program memory. The extended linear address record specifies address bits 16-31 for the data records that follow it. Data records are organized into the following fields:

:0A0004003C40343434261422260850	example record
:	start character
0A	byte count of this record
0004	address of first data byte
00	record type
3C	first data byte
08	last data byte
50	checksum

Extended linear address records have the following fields:

:02000000340850	
:	start character
02	byte count (always 02)
0000	address (always 0000)
00	record type
3408	offset address
50	checksum

The end of file record looks like this:

:00000001FF	
:	start character

00	byte count (zero for this record)
0000	address of first byte
01	record type
FF	checksum

Loader ASCII Format Files (.LDR)

The loader's ASCII-format file is similar to an assembler initialization data file (.DAT). The data order is one 16-bit hexadecimal value per line, providing lower-, middle-, then upper-16-bits of each 48-bit instruction. Use files of this format in the same manner as data files. For information on this format, see "Assembly Initialization Data Files (.DAT)" on page A-3.

Loader Include Format Files (.LDR)

The loader's include-format file is an ASCII text file that consists of 48-bit instructions one per line with each instruction presented as three 16-bit hexadecimal numbers. The data order is lower-, middle-, then upper-16-bits of each 48-bit instruction. A few example lines from an Include format file appear as follows:

```
0x005c, 0x0002, 0x0620,

0x0045, 0x0000, 0x1103,

0x00c2, 0x0002, 0x06be
```

This file format lets you include the loader file in a C program. To include this file in a C program, use the following code:

```
const unsigned loader_file[] =
{
#include "foo.ldr"
};
const unsigned loader_file_count = sizeof loader_file / sizeof
loader_file[0];
```

`loader_file_count` reflects the actual number of elements in the array, and can be used for processing the data.

Loader Binary Format Files (.LDR)

The loader's binary-format file supports a variety of PROM and microcontroller storage options and uses less space than the other loader file formats. The binary file contains 48-bit instructions in big-endian format (most significant bit first).

Debugger Files

Debugger files provide input to the debugger to define support for simulation or emulation of your program. The debugger supports all the executable file types produced by the linker (`.DXE`, `.SM`, `.OVL`, `.DLB`). To simulate I/O, the debugger also supports the data file formats (`.DAT`) from the assembler and the loadable file formats from the loader (`.LDR`).

The standard hexadecimal format for a SPORT data file is a single integer value per line. Hex numbers do not require the 0x prefix to indicate hexadecimal. A value can have any number of digits, but are read into the SPORT register, as follows:

- The hexadecimal number is converted to binary
- The number of binary bits read in matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zeros values that are shorter that the word size or, conversely, truncates any bits beyond the word size on the MSB end.

Example: a SPORT register is set for 20-bit words and the data file contains hex numbers. The simulator converts these hex numbers to binary, then fills/truncates to match the SPORT word size. In the following table, the number `A5A5` is filled and the number `123456` is truncated

Table A-2. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
A5A5A	1010 0101 1010 0101 1010	1010 0101 1010 0101 1010
FFFF1	1111 1111 1111 1111 0001	1111 1111 1111 1111 0001
A5A5	1010 0101 1010 0101	0000 1010 0101 1010 0101
5A5A5	0101 1010 0101 1010 0101	0101 1010 0101 1010 0101

Table A-2. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
11111	0001 0001 0001 0001 0001	0001 0001 0001 0001 0001
123456	0001 0010 0011 0100 0101 0110	0010 0011 0100 0101 0110

Format References

The following texts define industry standard file formats supported by VisualDSP++:

- (1993) Executable and Linkable Format (ELF) V1.1 from the Portable Formats Specification V1.1, Tools Interface Standards Committee {available from `ftp://ftp.intel.com/pub/tis`}
- (1993) Debugging Information Format (DWARF) V1.1 from the Portable Formats Specification V1.1, UNIX International, Inc. {available from `ftp://ftp.intel.com/pub/tis`}

B UTILITIES

Your Analog Devices development software comes with several file conversion utilities, which only run from a command line. Some of these utilities provide support for legacy code, and others are intended for a group of users who prefer to use the command line version of the tools instead of using them through the VisualDSP++ environment. This appendix describes these utilities and their command lines.

ELF File Dumper

The ELF file dumper (`elfdump.exe`) extracts data from ELF format executable files (`.DXE`) and provides a text output file that describes the ELF file's contents.

The ELF file dumper runs from the following command-line entry:

```
C:\Program Files\Analog Devices\VisualDSP>elfdump
```

Usage: `elfdump {option} {objectfile}`

Table B-1. ELF File Dumper Command-Line Switches

Switch	Description
`-fh`	Print file header.
`-arsym`	Print the archive symbol table
`-arall`	Print every archive member
`-ph`	Print program header table.

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

Switch	Description
`-sh`	Print section header table. The default is `-sh` if no options are specified.
-notes	Print note segments(s).
`-n` *name*	Print contents of the named section(s). Name may be a simple 'glob'-style pattern, using ? and * as wild card characters. Each section's name and type determines its output format, unless overridden by a modifier (see below).
`-i x0[-x1]`	Print contents of the sections numbered `x0` through `x1`, where `x0` and `x1` are decimal integers, and `x1` defaults to `x0` if omitted. Formatting rules as are for `-n`.
`-all`	Print everything. Same as `-fh -ph -sh -notes -n '*'`
`-ost`	Omit string table sections.
objectfile	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 an archive and `B` is an ELF element in the archive. `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 which forces section contents to be formatted in the following ways: `a` Dump contents in hex and ASCII, 16 bytes per line. `x` Dump contents in hex, 32 bytes per line. `xN` Dump contents in hex, N bytes per group (default is N=4). `t` Dump contents in hex, N bytes per line, where N is the section's table entry size. If N is not in the range 1...32, 32 is used. `i` Print contents as list of disassembled machine instructions.

Using the Archiver and Dumper For Disassembly

The file utilities can become much more effective when you combine their capabilities. One interesting application of these utilities is to disassemble a library member, converting it to source code. This application can be used when your source for a particularly useful routine is missing and is only available as a library routine.

The following procedure lists the objects in a library, extracts an object, and converts the object to a listing file. Using the following archiver command line, list the objects in the library and write the output to a text file:

```
elfar -p libc.dlb > libc.txt
```

Assuming the current directory is:
`C:\Program Files\Analog Devices\VisualDSP++\Blackfin\lib>`

Open the text file, scroll through it, and find the object file that you need. Then, use the following archiver command line 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 (almost source, just needs the line numbers and opcodes removed), use the following `elfdump` command line:

```
elfdump -ns * fir.doj > fir.asm
```

Using disassembly, you get a listing file with symbols. Assemble source with symbols can be useful if you are familiar with the code and hopefully have some documentation on what the code does. If symbols were stripped during linking, there are no symbols in the dumped file.

Using disassembly on a third party's library may violate the license for the third party's software. Check on copyright and license issues with the code's owner before using this disassembly technique.

Dumping Overlay Archive Files

Use the `elfar` and `elfdump` commands to extract and view the contents of the overlay archive file (*.`OVL`).

For example,

```
elfar -P CLONE2.OVL
```

will print the contents to CLONE2.ELF which will be an `*.ELF` file that can be viewed with `elfdump`.

Use `elfdump` to view CLONE2.ELF

```
elfdump -all CLONE2.OVL(CLONE2.elf)
```

or extract CLONE2.ELF and dump

```
elfar -e CLONE2.elf
elfdump -all CLONE2.elf
```

(or use whatever `elfdump` options one wants).

These commands are case sensitive.

I INDEX

P