VisualDSP++® 4.0
Linker and Utilities Manual

Revision 1.0, January 2005

Part Number
82-000420-03

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

Copyright Information

Disclaimer

Trademark and Service Mark Notice

CONTENTS

PREFACE

INTRODUCTION

LINKER

LINKER DESCRIPTION FILE

MEMORY OVERLAYS AND ADVANCED LDF COMMANDS

ARCHIVER

MEMORY INITIALIZER

FILE FORMATS

UTILITIES

LDF PROGRAMMING EXAMPLES FOR TIGERSHARC PROCESSORS

LDF PROGRAMMING EXAMPLES FOR SHARC PROCESSORS

LDF PROGRAMMING EXAMPLES FOR BLACKFIN

PROCESSORS

INDEX

PREFACE

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

Purpose of This Manual

The *VisualDSP++ 4.0 Linker and Utilities Manual* contains information about the linker and utilities programs for 16-bit fixed-point Blackfin® (ADSP-BFxxx) processors, and 32-bit (floating-point and fixed-point) TigerSHARC® (ADSP-TSxxx) and SHARC® (ADSP-21xxx) processors which set a new standard of performance for digital signal processors, combining multiple computation units for floating-point and fixed-point processing as well as wide word width.

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 utilities to produce DSP programs and provides reference information on the file utility software.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents

The manual contains:

- Chapter 1, "Introduction"
  This chapter provides an overview of the linker and utilities.
- Chapter 2, "Linker"
  This chapter describes how to combine object files into reusable library files to link routines referenced by other object files.
- Chapter 3, "Linker Description File"
  This chapter describes how to write an `.LDF` file to define the target.
- Chapter 4, "Expert Linker"
  This chapter describes Expert Linker, which is an interactive graphical tool to set up and map processor memory.
- Chapter 5, "Memory Overlays and Advanced LDF Commands"
  This chapter describes how overlays and advanced LDF commands are used for memory management.
- Chapter 6 "Archiver"
  This chapter describes the `elfar` archiver utility used to combine object files into library files, which serve as reusable resources for code development.
- Chapter 7 "Memory Initializer"
  This chapter describes the Memory Initializer utility that is used to generate a single initialization stream and save it in a section in the output executable file.

- Appendix A, "File Formats"
  This appendix lists and describes the file formats that the development tools use as inputs or produce as outputs.
- Appendix B, "Utilities"
  This appendix describes the utilities that provide legacy and file conversion support.
- Appendix C, "LDF Programming Examples for TigerSHARC Processors"
  This appendix provides code examples of `.LDF` files used with TigerSHARC processors.
- Appendix D, "LDF Programming Examples for SHARC Processors"
  This appendix provides code examples of `.LDF` files used with SHARC processors.
- Appendix E, "LDF Programming Examples for Blackfin Processors"
  This appendix provides code examples of `.LDF` files used with Blackfin processors.

What's New in This Manual

This is a new manual that documents linking support for all currently available Analog Devices 32-bit floating-point and fixed-point SHARC and TigerSHARC processors, as well as 16-bit fixed-point Blackfin processors.

Loader/splitter information is now available in a separate Loader manual.

Refer to *VisualDSP++ 4.0 Product Bulletin* for information on all new and updated features and other release information.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at `http://www.analog.com/processors/technicalSupport`
- E-mail tools questions to `dsptools.support@analog.com`
- E-mail processor questions to `dsp.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

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

Supported Processors

The following is the list of Analog Devices, Inc. processors supported in VisualDSP++ 4.0.

TigerSHARC (ADSP-TSxxx) Processors
The name “TigerSHARC” refers to a family of floating-point and fixed-point [8-bit, 16-bit, and 32-bit] processors. VisualDSP++ currently supports the following TigerSHARC processors:

ADSP-TS101	ADSP-TS201	ADSP-TS202	ADSP-TS203

SHARC (ADSP-21xxx) Processors
The name “SHARC” refers to a family of high-performance, 32-bit, floating-point processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC processors:

ADSP-21020	ADSP-21060	ADSP-21061	ADSP-21062
ADSP-21065L	ADSP-21160	ADSP-21161	ADSP-21261
ADSP-21262	ADSP-21266	ADSP-21267	ADSP-21363
ADSP-21364	ADSP-21365	ADSP-21366	ADSP-21367
ADSP-21368	ADSP-21369		

Blackfin (ADSP-BFxxx) Processors

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

ADSP-BF531	ADSP-BF532 (formerly ADSP-21532)
ADSP-BF533	ADSP-BF535 (formerly ADSP-21535)
ADSP-BF536	ADSP-BF537
ADSP-BF538	ADSP-BF539
ADSP-BF561	ADSP-BF566
AD6532	

Product Information

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

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

MyAnalog.com

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

Registration

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

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

Processor Product Information

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

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

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

Related Documents

For information on product related development software, see these publications:

- *VisualDSP++ 4.0 Getting Started Guide*
- *VisualDSP++ 4.0 User's Guide*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors*
- *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors*
- *VisualDSP++ 4.0 Assembler and Preprocessor Manual*
- *VisualDSP++ 4.0 Linker and Utilities Manual*
- *VisualDSP++ 4.0 Product Release Bulletin*
- *VisualDSP++ Kernel (VDK) User's Guide*
- *Quick Installation Reference Card*

For hardware information, refer to your processors's hardware reference, programming reference, or data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

`http://www.analog.com/processors/resources/technicalLibrary`

Online Technical Documentation

Online documentation includes the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and Flexible License Manager (FlexLM) network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest using the Search function of VisualDSP++ Help system. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

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

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

Accessing Documentation From VisualDSP++

From the VisualDSP++ environment:

- 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).

Accessing Documentation 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`) are located in the `Help` folder of VisualDSP++ environment. The `.PDF` files are located in the `Docs` folder of your VisualDSP++ installation CD-ROM. The `Docs` folder also contains the Dinkum Abridged C++ library and the FlexLM network license manager software documentation.

Using Windows Explorer

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

Using the Windows Start Button

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

Accessing Documentation From the Web

Download manuals in PDF format at the following Web site:
`http://www.analog.com/processors/resources/technicalLibrary/manuals`

Select a processor 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

To purchase VisualDSP++ manuals, call **1-603-883-2430**. The manuals may be purchased only as a kit.

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

Hardware Tools Manuals

To purchase EZ-KIT Lite® and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**), or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Only production (final) data sheets (Rev. 0, A, B, C, and so on) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**); they also can be downloaded from the Web site.

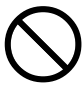
To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

Text conventions used in this manual are identified and described as follows.

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

Example	Description
Close command (**File** menu)	Titles in reference sections indicate 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`. One or the other is required.
`[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 ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.

Example	Description
	**Note:** For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	**Caution:** Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	**Warning:** Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 INTRODUCTION

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

> The code examples in this manual have been compiled using VisualDSP++ 4.0. The examples compiled with other versions of VisualDSP++ may result in build errors or different output although the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

This chapter includes:

- "Software Development Flow" on page 1-2
  Shows how linking, loading, and splitting fit into the project development process.
- "Compiling and Assembling" on page 1-3
  Shows how compiling and assembling the code fits into the project development process.
- "Linking" on page 1-7
  Shows how linking fits into the project development process.
- "Loading and Splitting" on page 1-10
  Shows how loading and splitting fit into the project development process.

Software Development Flow

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

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

After running the linker, you test the output with a simulator or emulator. Refer to the *VisualDSP++ 4.0 User's Guide* and online Help for information about debugging.

Finally, you process the debugged executable file(s) through the loader or splitter to create output for use on the actual processor. The output file may reside on another processor (host) or may be burned into a PROM. The *VisualDSP++ 4.0 Loader Manual* describes loader/splitter functionality for the target processors.

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

1. Compiling and Assembling – Input source files C (`.C`), C++ (`.CPP`), and assembly (`.ASM`) yield object files (`.DOJ`)

2. Linking – Under the direction of the Linker Description File (`.LDF`), a linker command line, and VisualDSP++ **Project Options** dialog box settings, the linker utility consumes object files (`.DOJ`) to yield an executable (`.DXE`) file. If specified, shared memory (`.SM`) and overlay (`.OVL`) files are also produced.

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

Compiling and Assembling

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

Inputs – C/C++ and Assembly Sources

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

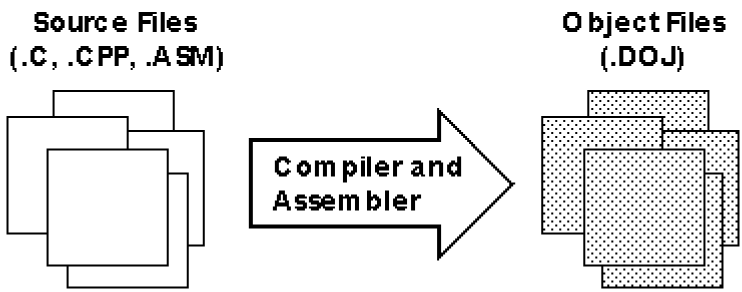


Figure 1-1. Compiling and Assembling

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

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

Each input section in the .LDF file requires a unique name, as specified in the source code. Depending on whether the source is C, C++, or assembly, different conventions are used to name an input section (see Chapter 3, "Linker Description File").

Input Section Directives in Assembly Code

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

SHARC Code Example

```
.SECTION/DM asmdata;      // Declares section asmdata
.VAR input[3];            // Declares data buffer in asmdata

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

In this example, the /dm asmdata input section contains the array input, and the /pm asmcode input section contains the three line of code.

Blackfin Code Example

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

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

Input Section Directives in C/C++ Source Files

Typically, C/C++ code does not specify an input section name, so the compiler uses a default name. By default, the input section names `program` (for code) and `data1` (for data) are used. Additional input section names are defined in `.LDF` files (see "Specifying the Memory Map" on page 2-17 for more information on memory mapping).

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

Example 1

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

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

Example 2

In the following example, `section ("extern")` is optional. Note the new function (`funct2`) does not require `section ("extern")`. For more information on LDF sections, refer to "Specifying the Memory Map" on page 2-17.

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

For information on compiler default section names, refer to the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual* for appropriate target processors and "Placing Code on the Target" on page 2-40.

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

Linking

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

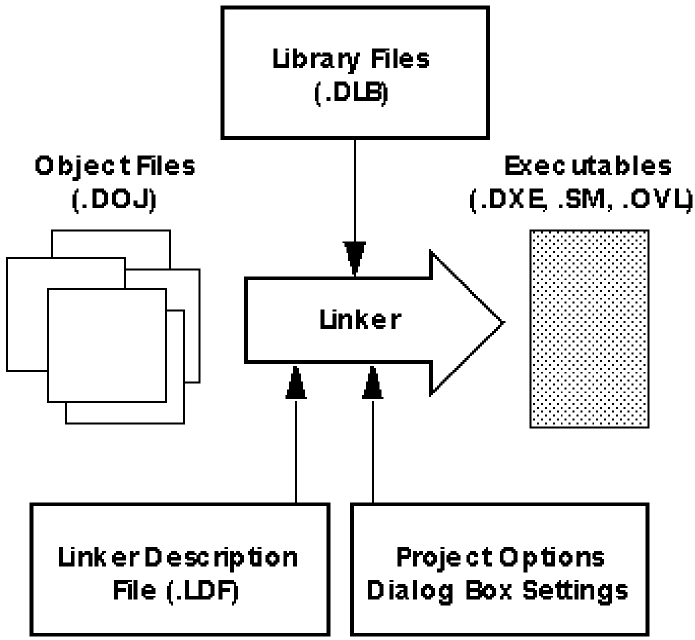


Figure 1-2. Linking Diagram

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

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

Linker and Assembler Preprocessor

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

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

Assembler/Linker Preprocessor treats the character "." as part of an identifier

The preprocessor matches the assembler which uses "." as part of assembler directives and as a legal character in labels. This behavior does create a possible problem for users that have written preprocessor macros that rely on identifiers to break when encountering the "." character, usually seen when processing register names. For example,

```
#define Loadd(reg, val) \
reg.l = val;\
reg.h = val;
```

The above example would not work in VisualDSP++ 4.0 becauseVisualDSP++ 4.0 does not provide any replacement since reg is not parsed as a separate identifier. The macro has to be rewritten using the operator ## such as:

```
#define Loadd(reg, val)\
reg ## .l = val;\
reg ## .h = val;
```

The preprocessor supports ANSI C standard preprocessing with extensions but differs from the ANSI C standard preprocessor in several ways. For more information on the `pp` preprocessor, see the *VisualDSP++ 4.0 Assembler and Preprocessor Manual*.

The compiler has it own preprocessor that allows you to use preprocessor commands within your C/C++ source. The compiler preprocessor automatically runs before the compiler. For more information, see the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual* for the appropriate target archtecture.

Loading and Splitting

After debugging the .DXE file, you process it through a loader or splitter to create output files used by the actual processor. The file(s) may reside on another processor (host) or may be burned into a PROM.

For more information, refer to the *VisualDSP++ 4.0 Loader Manual* which provides detailed descriptions of the processes and options used to generate boot-loadable .LDR (loader) files for the approprate target processors. This manual also describes the splitting utility, which (when used) creates the non-bootloadable files that execute from the processor's external memory.

In general:

- The SHARC ADSP-2106x/ADSP-21160 processors use the loader (elfloader.exe) to yield a boot-loadable image (.LDR file), which resides in memory external to the processor (PROM or host processor). Use the splitter utility (elfspl21k.exe) to generate non-bootable PROM image files, which execute from the processor's external memory (often used with the ADSP-21065L processors).
- The SHARC ADSP-2116x/2126x/2136x processors use the loader (elfloader.exe) to yield a boot-loadable image (.LDR file), which transported to (and run from) processor memory. To make a loadable file, the loader processes data from a boot-kernel file (.DXE) and one or more other executable files (.DXE).
- The TigerSHARC processors use the loader (elfloader.exe) to yield a boot-loadable image (.LDR file), which transported to (and run from) processor memory. To make a loadable file, the loader processes data from a boot-kernel file (.DXE) and one or more other executable files (.DXE).

- Both TigerSHARC and SHARC processors use the splitter utility (`elfspl21k.exe`) to generate non-bootable PROM image files, which execute from the processor's external memory.
- The Blackfin processors use the loader (`elfloader.exe`) to yield a boot-loadable image (`.LDR` file), which resides in memory external to the processor (PROM or host processor. To make a loadable file, the loader processes data from a boot-kernel file (`.DXE`) and one or more other executable files (`.DXE`).

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

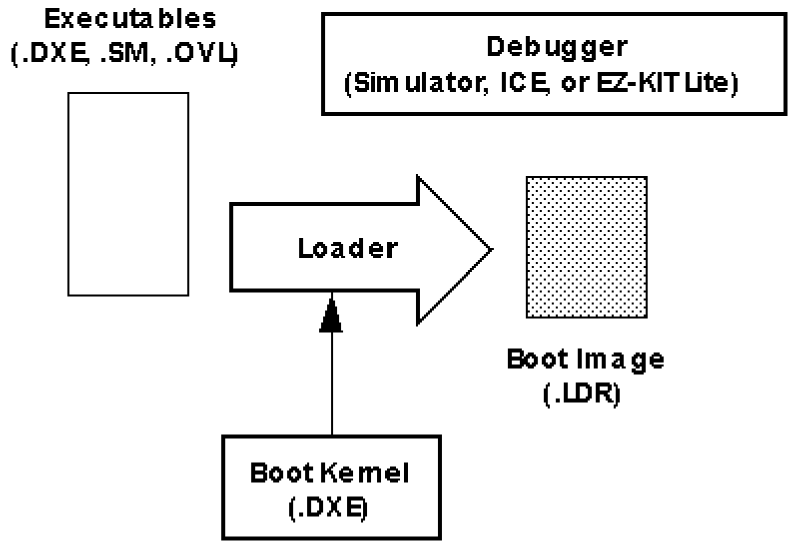


Figure 1-3. Loading Diagram

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

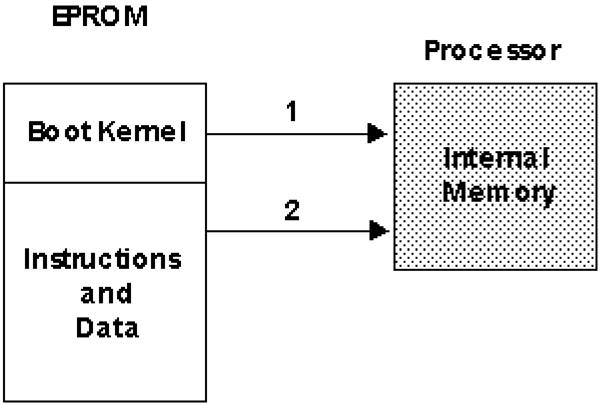


Figure 1-4. Booting from a Bootloadable (.LDR) File

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

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

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

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

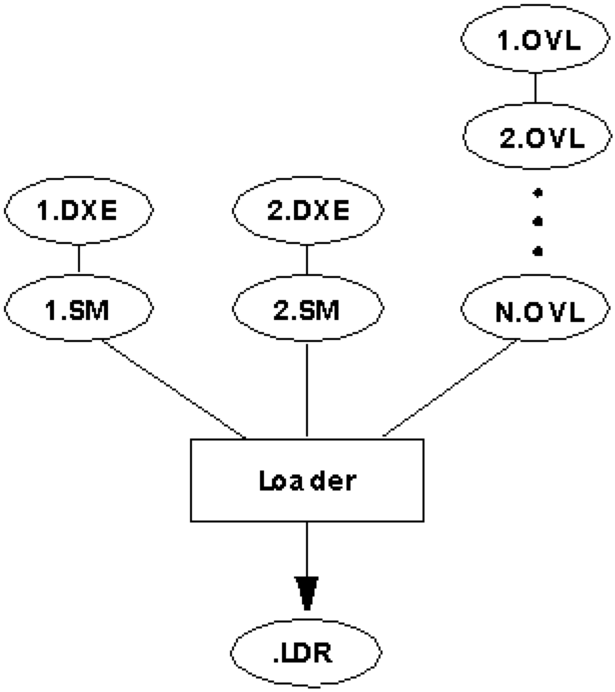


Figure 1-5. Input Files for a Multiprocessor System

2 LINKER

Linking assigns code and data to processor memory. For a simple single processor architecture, a single `.DXE` file is generated. A single invocation of the linker may create multiple executable (`.DXE`) files for multiprocessor (MP) architectures. Linking can also produce a shared memory (`.SM`) file for an MP system. A large executable file can be split into a smaller executable file and overlays (`.OVL`) files, which contain code that is called in (swapped into internal processor memory) as needed. The linker (`linker.exe`) performs this task.

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

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

This chapter includes:

Linker Operation

Figure 2-1 illustrates a basic linking operation. The figure shows several object (.DOJ) files being linked into a single executable (.DXE) file. The Linker Description File (.LDF) directs the linking process.

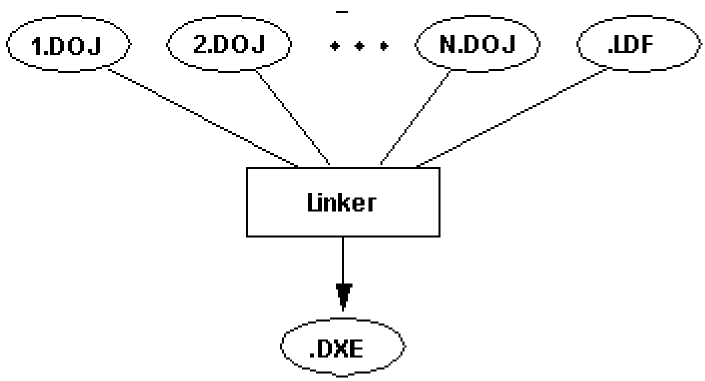


Figure 2-1. Linking Object Files to Produce an Executable File

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

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

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

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

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

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

Directing Linker Operation

Linker operations are directed by these options and commands:

- Linker (`linker.exe`) command-line switches (options). Refer to "Linker Command-Line Reference" on page 2-47.
- Settings (options) on the **Link** page of the **Project Options** dialog box. See "Project Builds" on page 2-6.
- LDF commands. Refer to "LDF Commands" on page 3-29 for a detailed description.

Linker options control how the linker processes object files and library files. These options specify various criteria such as search directories, map file output, and dead code elimination. You select linker options via linker command-line switches or by settings on the **Link** page of the **Project Options** dialog box within the VisualDSP++ environment.

LDF commands in a Linker Description File (`.LDF`) define the target memory map and the placement of program sections within processor memory. The text of these commands provides the information needed to link your code.

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

Using directives in the .LDF file, the linker:

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

Linking Process Rules

The linking process observes these rules:

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

- Contiguous addresses on different-width hardware must reside in different memory segments.
- More than one output section may map to a memory segment if the output sections fit completely within the memory segment.

Linker Description File Overview

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

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

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

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

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

Refer to Chapter 3, "Linker Description File" for further information.

Linking Environment

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

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

Project Builds

The linker runs from an operating system command line, issued from the VisualDSP++ IDDE, or a command prompt window. Figure 2-2 shows the VisualDSP++ environment with the **Project** window and an `.LDF` file open in an **Editor** window.

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

ⓘ For information about the VisualDSP++ environment, refer to the *VisualDSP++ User's Guide* for the appropriate target architecture or online Help. Online Help provides powerful search capabilities. To obtain information on a code item, parameter, or error, select text in an VisualDSP++ IDDE **Editor** window or **Output** window and press the keyboard's **F1** key.

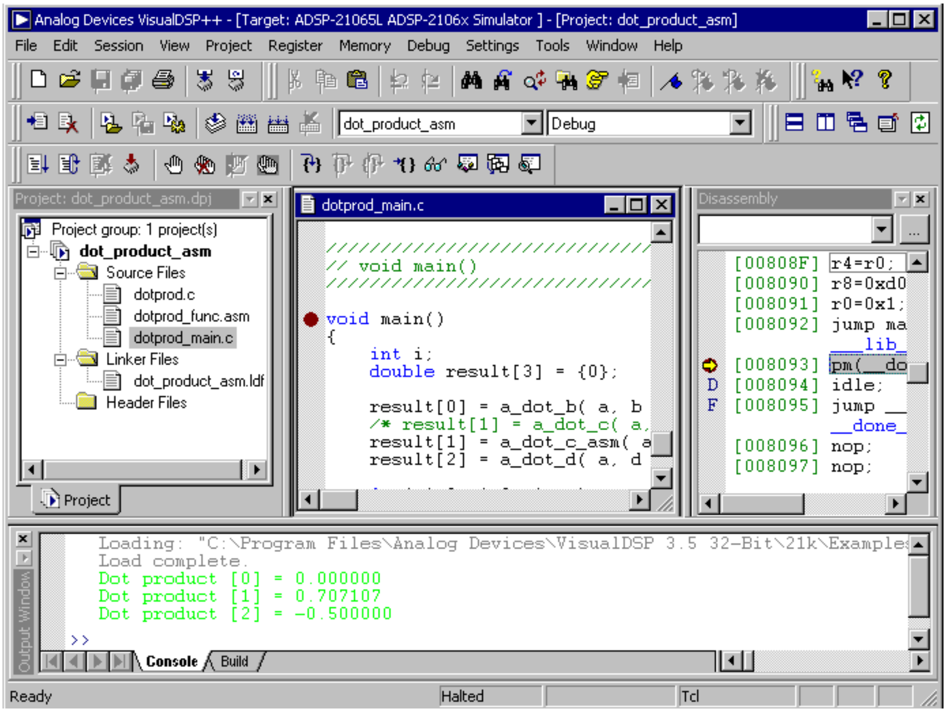


Figure 2-2. VisualDSP++ Environment

Within VisualDSP++, specify tool settings for project builds. Use the **Project** menu to open **Project Options** dialog box.

Figure 2-3, Figure 2-4, and Figure 2-5 show the project option selections for SHARC, TigerSHARC, and Blackfin processors, respectively.

These dialog boxes allow you to select the target processor, type and and name of the executable file, as well as VisualDSP++ tools available for use with the selected processor.

When using the VisualDSP++ IDDE, use the **Link** option from the **Project Options** dialog box (Figure 2-6) to select and/or set linker functional options.

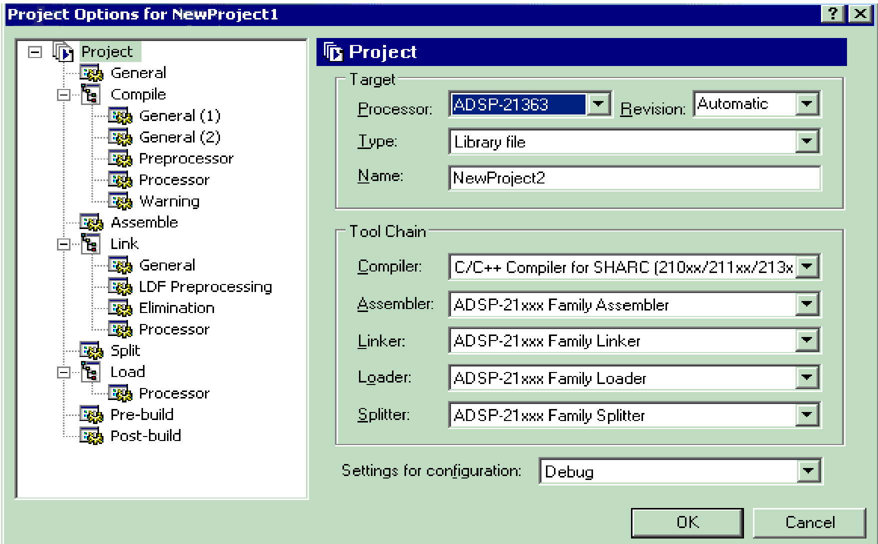


Figure 2-3. Project Options Dialog Box (SHARC Processors)

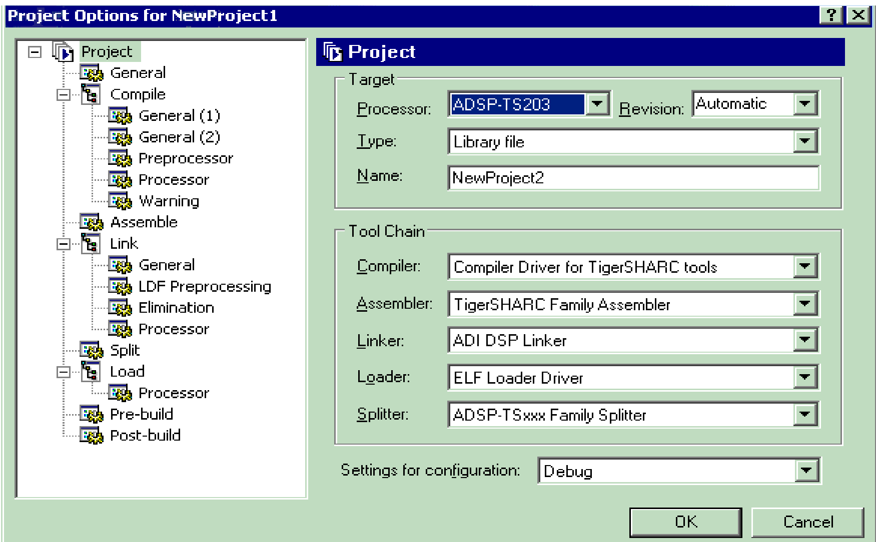


Figure 2-4. Project Options Dialog Box (TigerSHARC Processors)

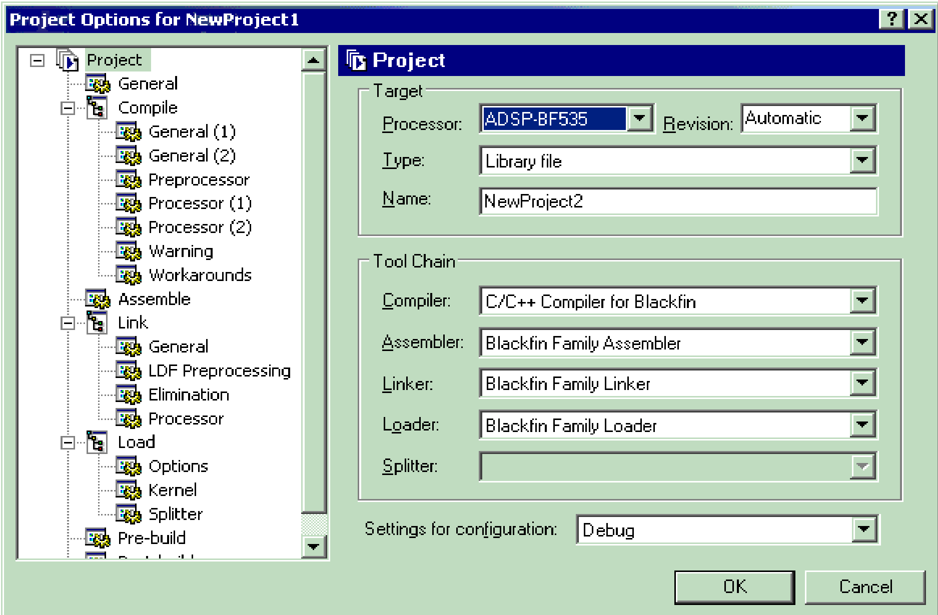


Figure 2-5. Project Options Dialog Box (Blackfin Processors)

There are four subpages you can access—**General**, **LDF Preprocessing**, **Elimination**, and **Processor**. Almost every setting option has a corresponding compiler command-line switch described in "Linker Command-Line Switches" on page 2-51.

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

Due to different processor architectures, the processors may provide different **Link** tab selection options. For example, Figure 2-7 shows Blackfin processor options page.

Figure 2-8 shows SHARC processor options page.

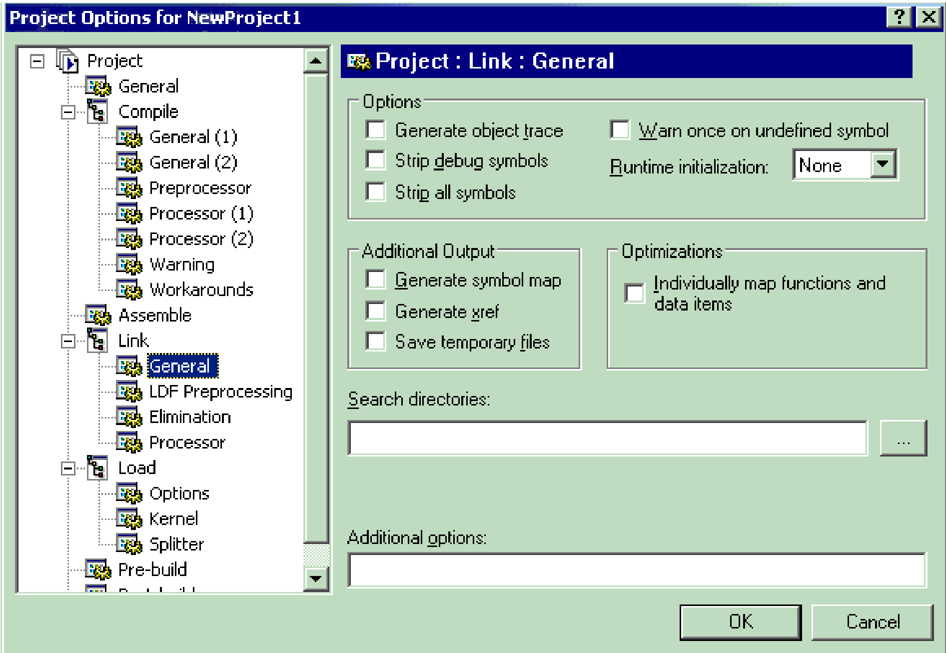


Figure 2-6. Project Options – Link Page

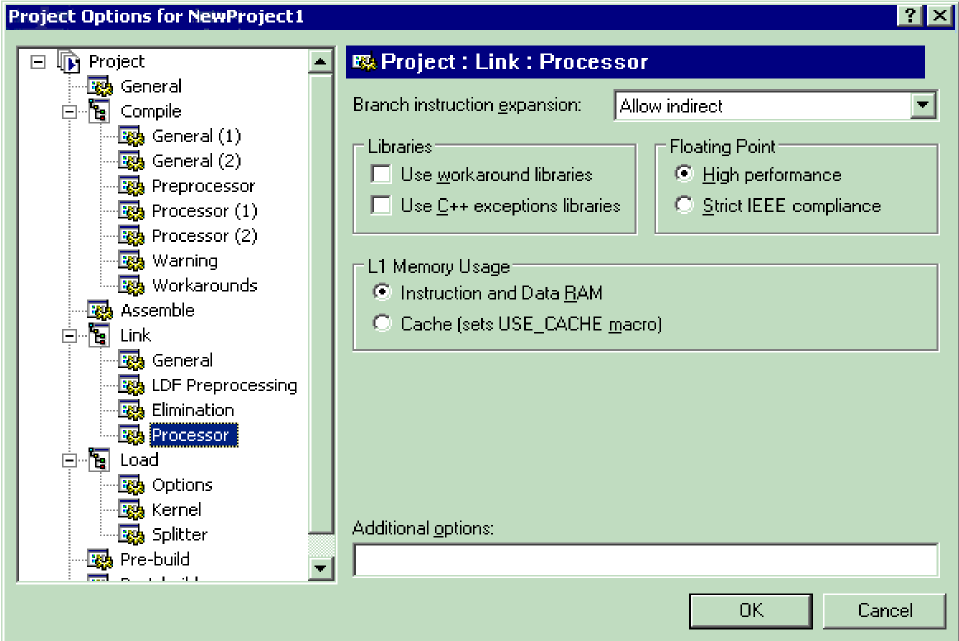


Figure 2-7. Project Options – Link (Processor) Blackfin Page

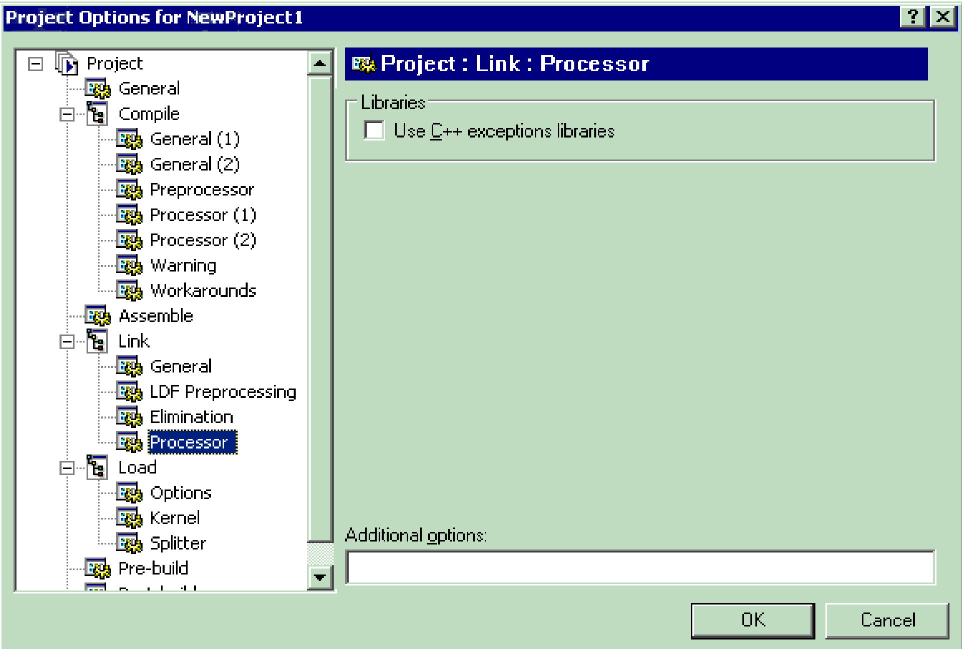


Figure 2-8. Project Options – Link (Processor) SHARC Page

Figure 2-9 shows TigerSHARC processor options page.

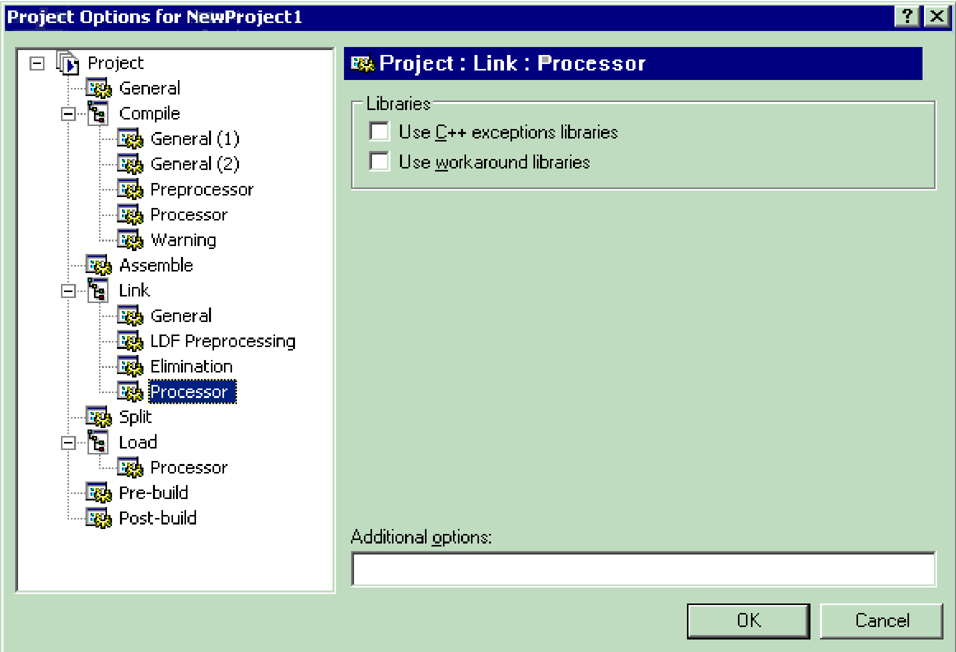


Figure 2-9. Project Options – Link (Processor) TigerSHARC Page

Use the VisualDSP++ context-sensitive online Help for each target architecture to obtain information on linker options you can specify in VisualDSP++. To do that, click on the **?** button and then click in a field or box you need information about.

Expert Linker

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

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

Figure 2-10 shows the Expert Linker window, which comprises two panes: **Input Sections** and **Memory Map** (output sections). Refer to Chapter 4, "Expert Linker", for detailed information.

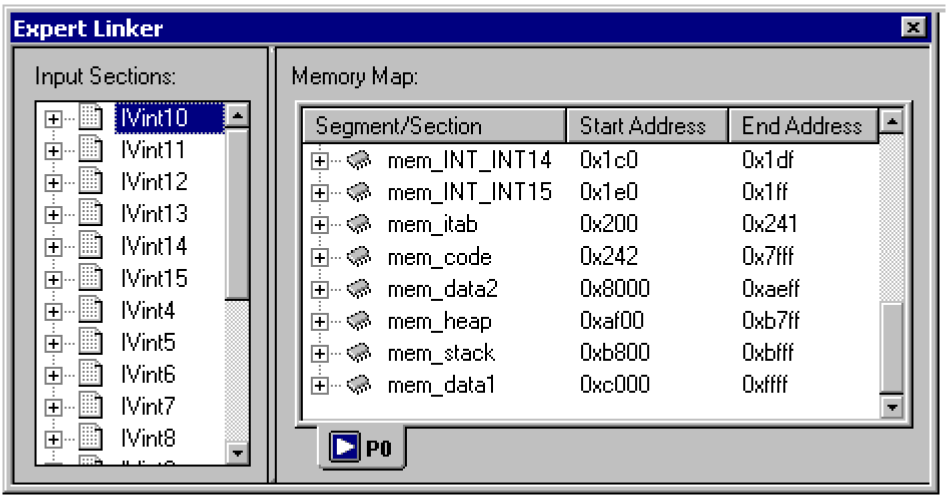


Figure 2-10. Expert Linker Window

Linker Warning and Error Messages

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

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

Example

```
>linker -proc ADSP-unknown a.doj

[Error li1010]    The processor 'ADSP-unknown' is
                  unknown or unsuppported.
```

Interpreting Linker Messages

Within VisualDSP++, the **Output** window's **Build** tab displays project build status and error messages. In most cases, double-clicking a message displays the line in the source file causing the problem. You can access descriptions of linker messages from VisualDSP++ online Help by selecting a six-character code (for example, `li1010`) and pressing the **F1** key.

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

For example, if an input section from the object file is not placed by the `.LDF` file, a cross-reference error occurs at every object that refers to labels in the missing section. Fix this problem by reviewing the `.LDF` file and specifying all sections that need placement. For more information, refer to the *VisualDSP++ 4.0 User's Manual* or online Help.

Link Target Description

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

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

> If the project does not include an .LDF file, the linker uses a default .LDF file for the processor that matches the -proc <processor> switch on the linker's command line (or the **Processor** selection specified on the **Project** page of the **Project Options** dialog box in the VisualDSP++ IDDE).

Be sure to understand the processor's memory architecture, which is described in the appropriate processor's hardware reference manual and in its data sheet.

This section contains:

- "Representing Memory Architecture" on page 2-17
- "Specifying the Memory Map" on page 2-17
- "Placing Code on the Target" on page 2-40
- "Profile-Guided Optimization Support" on page 2-45
- "Passing Arguments for Simulation or Emulation" on page 2-46

Representing Memory Architecture

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

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

- **Memory Usage.** List the ways your program uses memory in your system. Typical uses for memory segments include interrupt tables, initialization data, program code, data, heap space, and stack space. Refer to "Specifying the Memory Map" on page 2-17.
- **Memory Characteristics.** List the types of memory in your processor system and the address ranges and word width associated with each memory type. Memory type is defined as RAM or ROM.
- **MEMORY{} Command**. Construct a MEMORY{} command to combine the information from the previous two lists and to declare your system's memory segments.

For complete information, refer to "MEMORY{}" on page 3-35.

Specifying the Memory Map

A DSP program must conform to the constraints imposed by the processor's data path (bus) widths and addressing capabilities. The following information describes an .LDF file for a hypothetical project. This file specifies several memory segments that support the SECTIONS{} command, as shown in "SECTIONS{}" on page 3-48.

The three topics are important when allocating memory:

- "Memory Usage and Default Memory Sections" on page 2-18
- "Memory Characteristics Overview" on page 2-31
- "Linker MEMORY{} Command in .LDF File" on page 2-38

Memory Usage and Default Memory Sections

Input section names are generated automatically by the compiler or are specified in the assembly source code. The `.LDF` file defines memory section names and output section names. The default `.LDF` file handles all compiler-generated input sections (refer to the "Input Section" column in Table 2-1, Table 2-2, and Table 2-3). The produced `.DXE` file has a corresponding output section for each input section. Although programmers typically do not use output section labels, the labels are used by downstream tools.

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

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

ⓘ Refer to your processor's default `.LDF` file and to the hardware reference manual for details.

Typical uses for memory segments include interrupt tables, initialization data, program code, data, heap space, and stack space, etc. For detailed processor-specific information, refer to:

- "Default Memory Sections for SHARC Processors"
- "Default Memory Sections for TigerSHARC Processors"
- "Default Memory Sections for Blackfin Processors"
- "Special "Table" Memory Sections"

Default Memory Sections for SHARC Processors

Table 2-1 shows section mapping in the default .LDF file for ADSP-21161 processor (as an example for SHARC processors)

Table 2-1. Section Mapping in the Default SHARC LDF

Input Section	Output Section	Memory Section
seg_pmco	dxe_pmco	seg_pmco
seg_dmda	dxe_dmda	seg_dmda
seg_pmda	dxe_pmda	seg_pmda
heap	dxe_heap	seg_heap
seg_stak	dxe_stak	seg_stak
sec_rth	dxe_rth	seg_rth
seg_init	seg_init	seg_init
seg_init_code	seg_init_code	seg_init_code
seg_argv	seg_argv	seg_argv
seg_ctdm	seg_ctdm	seg_ctdm
seg_vtbl	seg_vtbl	seg_vtbl

There are several input/memory sections used in the default .LDF files for ADSP-210xx/211xx/212xx/213xx processors, which must be present in user's own .LDF files. These sections are described in detail below.

seg_rth

This section contains the interrupt vector table (by default, this is located in the start-up file (for example, 060_hdr.doj).

seg_init

This section contains location and size information about the stack and heap; also contains compressed data created by the memory initialization tool (see “-meminit” on page 2-62 for more information).

seg_int_code

Code that modifies interrupt latch registers must not be executed from external memory. To minimize the impact of this restriction, the library functions that modify the latch registers are located in the `seg_init_code` section, which should be located in internal memory.

seg_pmco

This section is the default location for program code.

seg_pmda

This section is the default location for global program data that is qualified with the “`pm`” keyword. For example,

```
int pm xyz[100];   // Located in seg_pmda
```

seg_argv

This section contains the command-line arguments that are used as part of Profile-Guided Optimization (PGO).

seg_ctdm

This section (see also `seg_ctdml` on page 2-21) contains the addresses of constructors that are called before the start of a C++ program (such as constructors for global and static objects). This section must be terminated with the symbol “`___ctor_NULL_marker`” (the default `.ldf` files ensure this). It is required if compiling with C++ code.

seg_dmda

This section is the default location for global data, and also for data that is qualified with the "dm" keyword. For example,

```
int abc[100];      // Located in seg_dmda
int dm def[100];   // Located in seg_dmda
```

seg_heap

This section is the area from which memory allocation functions and operators (new, malloc(), and so on) allocate memory.

seg_stak

This section is the area where the run-time stack is located. Local variables, function parameters, and so on are stored here.

seg_vtbl

This section contains C++ virtual function tables. The default .LDF files place the virtual function tables into the default data memory area but this can be re-mapped as required. You can also direct the compiler to use a different section for C+ virtual function tables, by using the -section compiler switch.

Other Memory Sections

The compiler and libraries also use other data sections that are linked into one of the above memory sections. These data sections include:

seg_ctdml

The symbol "___ctor_NULL_marker" (located in the C++ run-time library) marks the end of the list of global and static constructors and is placed in this data section. The linker ensures that the contents of this data section are the last items in seg_ctdm.

.gdt .gdtl .frt .frtl .cht .chtl .edt .edtl

These data sections are used to hold data used during the handling of exceptions. The linker places the contents of these data sections in seg_dmda. See "Special "Table" Memory Sections" on page 2-30.

Default Memory Sections for TigerSHARC Processors

Table 2-2 shows section mapping in the default .LDF file for ADSP-TS101 processor (as an example for TigerSHARC processors)

Table 2-2. Section Mapping in the Default TigerSHARC LDF

Input Section	Output Section	Memory Section
program	code	M0Code
data1	data1	M1Data
data2	data2	M2Data
heaptab	heaptab	M1Data
mem_argv	mem_argv	M1Data
bsz_init	bsz	M1Data
bsz	bsz_init	M1Data
	jstackseg	M1Stack
	kstackseg	M2Stack
	defheapseg	M1Heap
ctor	ctor	M1Data
ctor0	ctor	M1Data
ctor1	ctor	M1Data
ctor2	ctor	M1Data
ctor3	ctor	M1Data

Table 2-2. Section Mapping in the Default TigerSHARC LDF (Cont'd)

Input Section	Output Section	Memory Section
ctor4	ctor	M1Data
vtbl	vtbl	vtbl

There are several input/memory sections used in the default .LDF files for ADSP-TSxxx processors, which must be present in user's own .LDF files. These sections are described in detail below.

program

This section is the default location for program code.

data1

This section is the default location for global program data.

data2

This section is the default location for global program data specified with the pm memory qualifier.

mem_argv

This section contains the command-line arguments that are used as part of Profile-Guided Optimization (PGO).

ctor

This section contains the addresses of constructors that are called before the start of a C++ program (such as constructors for global and static objects). This section must be terminated with the symbol "___ctor_NULL_marker" (the default .ldf files ensure this). It is required if compiling with C++ code.

When all ctor sections are merged, they form a table containing a list of all constructors for all global C++ objects. The table is only used at startup and can be placed in ROM. When linking, it is important that all ctor sections are merged in sequence (no other sections in between) and the run-time library or the VDK run-time library is placed with the first ctor section. Note that the default LDF's "___ctor_NULL_marker" symbol is placed in a section named "ctorl" which must the the last of the ctor sections to be used as input. The final letter in this name is a lower-case "L".

heaptab

This section is the area from which memory allocation functions and operators (new, malloc(), etc.) allocate memory.

bsz

This section is a BSS-style section for global zero-initialized data.

bsz_init

This section contains run-time initialization data (see "-meminit" on page 2-62 for more information).

M1Stack

This section is the area where the run-time stack is located. Local variables, function parameters, and so on are stored here.

M2Stack

This section is the second area where the run-time stack is located.

M1Heap

This section is the area where the heap is located. Dynamically allocated data is placed here.

vtbl

This section contains C++ virtual function tables. The default .LDF files place the virtual function tables into the default data memory area but this can be re-mapped as required. You can also direct the compiler to use a different section for C+ virtual function tables, by using the -section compiler switch.

Default Memory Sections for Blackfin Processors

Table 2-2 shows section mapping in the default .LDF file for ADSP-BF535 processor (as an example for Blackfin processors)

Table 2-3. Section Mapping in the Default ADSP-BF535 LDF

Input Section	Output Section	Memory Section
program	program	MEM_PROGRAM
data1	data1	MEM_PROGRAM
cplb_code	cplb_code	MEM_PROGRAM
L1_code	L1_code	MEM_L1_CODE
constdata	constdata	MEM_PROGRAM
cplb_data	cplb_data	MEM_PROGRAM
L1_DATA_A	L1_DATA_A	MEM_L1_DATA_A
L1_DATA_B	L1_DATA_B	MEM_L1_DATA_A
voldata	voldata	MEM_PROGRAM
bsz	bsz	MEM_L1_CODE
bsz_init	bsz_init	MEM_PROGRAM
heap	heap	MEM_HEAP
stack	stack	MEM_STACK
ctor	program	MEM_PROGRAM
argv	argv	MEM_ARGV

Table 2-3. Section Mapping in the Default ADSP-BF535 LDF (Cont'd)

Input Section	Output Section	Memory Section
noncache_code	noncache_code	noncache_code
sdram	sdram	MEM_SDRAM
vtbl	sdram	vtbl

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

There are several input/memory sections used in the default .LDF files for ADSP-BFxxx processors, which must be present in user's own .LDF files. These sections are described in detail below.

program

This section is the default location for program code. Code in this section is mapped into L1 Instruction memory if possible, but will spill into slower memory if there is not enough space.

data1

This section is the default location for global program data.

cplb_code

This section stores the run-time library's CPLB management routines. It is usually mapped into L1 Instruction SRAM. In particular, if CPLB-replacement is a possibility, this section must be mapped to memory that is guaranteed to always available; this means that it must be addressed by a locked CPLB.

constdata

This section is used for global data which is declared as constant, and for literal constants such as strings and array initializers. The compiler's default behavior is to assume that "`const`" data remains constant, and therefore can be mapped into read-only memory (although the default LDFs do not do this). Refer to the `-const-read-write` switch in the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for Blackfin Processors* for more details.

cplb_data

This section stores CPLB configuration tables In particular, the cplbtabx.doj files (where x indicates the target) mapped by the LDFs are placed into this section. It must be mapped to a data area that is always available. If CPLB replacement is a possibility, this section must be mapped into memory that is covered by a locked CPLB.

L1_DATA_A

This section is used to allow data to be mapped explicitly into L1 Data A SRAM using the `SECTION` directive. By default, the compiler does not generate data here. This section is analogous to `L1_code`.

L1_DATA_B

This section is similar to `L1_DATA_A`, except that it is used to map data into L1 Data B SRAM.

voldata

This section is used for data that may change due to external influences (such as DMA), and should not be placed into cached data areas.

argv

This section contains the command-line arguments that are used as part of Profile-Guided Optimization (PGO).

ctor

This section contains the addresses of constructors that are called before the start of a C++ program (such as constructors for global and static objects). This section must be terminated with the symbol "`___ctor_NULL_marker`" (default `.LDF` files ensure this). It is required if compiling with C++ code.

When all `ctor` sections are merged, they form a table containing a list of all constructors for all global C++ objects. The table is only used at startup and can be placed in ROM. When linking, it is important that all `ctor` sections are merged in sequence (no other sections in between) and the run-time library or the VDK run-time library is placed with the first `ctor` section. Note that the default LDF's "`___ctor_NULL_marker`" symbol is placed in a section named "`ctorl`" which must the the last of the `ctor` sections to be used as input. The final letter in this name is a lower-case "`L`".

bsz

This section is a BSS-style section for global zero-initialized data. This section does not actually contain data; it is zero-filled upon loading via the VisualDSP++ IDDE, or when processed by the loader.

bsz_init

This section contains run-time initialization data (see "-meminit" on page 2-62 for more information). It is expected that this section is mapped into read-only memory. When a `.DXE` file has been processed by the Memory Initializer utility and the program starts running, other data sections (such as `data1` and `constdata`) are initialized by data copied from this section.

stack

This section is the area where the run-time stack is located. Local variables, function parameters, and so on are stored here.

heap

This section is the area where the heap is located. Dynamically allocated data is placed here.

noncache_code

This section is mapped to areas of memory that cannot be cache and take program code. This section would be used if you have a function that turns on the cache to ensure that the function itself does not reside in cache (as executing code from a cache memory address causes a hardware exception).

sdram0

This section allows code or data to be mapped explicitly into external memory by using the `SECTION` directive. This can be used to place large, infrequently-used data or functions into external memory to free up valuable internal memory.

vtbl

This section contains C++ virtual function tables. The default `.LDF` files place the virtual function tables into the default data memory area but this can be re-mapped as required. You can also direct the compiler to use a different section for C+ virtual function tables, by using the `-section` compiler switch.

Other Memory Sections

The compiler and libraries also use other data sections that are linked into one of the above memory sections. These data sections include:

ctorl

This section contains the terminator for the `ctor` table section. It must be mapped immediately after the `ctor` section.

.gdt .gdtl .frt .frtl .cht .chtl .edt .edtl

These data sections are used to hold data used during the handling of exceptions. See "Special "Table" Memory Sections" on page 2-30.

Special "Table" Memory Sections

The following "table" data sections are used to hold data used during the handling of exceptions. The linker is generally mapping these sections into read-only memory.

.gdt

This section (Global Dispatch Table) is used by the C++ Exception Library to determine which area of code to which a particular address belongs. This section must be contiguous in memory.

.gdtl

This section contains the terminator for the `.gdt` table section. It must be mapped immediately after the `.gdt` section.

.edt

This section (Exception Dispatch Table) is used by the C++ Exception Library to map from try blocks to catch blocks.

.edtl

This section contains the terminator for the `.edt` table section. It must be mapped immediately after the `.edt` section.

.cht

This section (Catch Handler Types Table) is used to map to the RTTI type information. The C++ Exception Library uses it to determine the types that correspond to catch entries for a try block.

.chtl

This section contains the terminator for the `.cht` table section. It must be mapped immediately after the `.cht` section.

.frt

This section (Function Range Table) is used by the C++ Exception Library during exception processing to unwind the stack of active functions.

.frtl

This section contains the terminator for the `.frt` table section. It must be mapped immediately after the `.frt` section.

Memory Characteristics Overview

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

ⓘ Some portions of the processor memory are reserved. Refer to the hardware reference manual for target processor for more information.

SHARC Memory Characteristics

As an example of the SHARC memory architecture, the ADSP-21161 processor contains a large, dual-ported internal memory for single-cycle, simultaneous, independent accesses by the core processor and I/O processor. The dual-ported memory in combination with three separate on-chip buses allow two data transfers from the core and one transfer from the I/O processor in a single cycle. Using the IO bus, the I/O processor provides data transfers between internal memory and the processor's communication ports (link ports, serial ports, and external port) without hindering the processor core's access to memory. The processor provides access to external memory through the processor's external port.

The processor contains one megabit of on-chip SRAM, organized as two blocks of 0.5 Mbits. Each block can be configured for different combinations of code and data storage. All of the memory can be accessed as 16-bit, 32-bit, 48-bit, or 64-bit words. The memory can be configured in each block as a maximum of 16 Kwords of 32-bit data, 8 Kwords of 64-bit data, 32K words of 16-bit data, 10.67 Kwords of 48-bit instructions (or 40-bit data), or combinations of different word sizes up to 0.5 Mbit. This gives a total for the complete internal memory: a maximum of 32 Kwords of 32-bit data, 16 Kwords of 64-bit data, 64 Kwords of 16-bit data, and 21 Kwords of 48-bit instructions (or 40-bit data).

The processor features a 16-bit floating-point storage format that effectively doubles the amount of data that may be stored on-chip. A single instruction converts the format from 32-bit floating-point to 16-bit floating-point.

While each memory block can store combinations of code and data, accesses are most efficient when one block stores data using the DM bus, (typically, Block 1) for transfers, and the other block (typically, Block 0) stores instructions and data using the PM bus. Using the DM bus and PM bus with one dedicated to each memory block assures single-cycle execution with two data transfers. In this case, the instruction must be available in the cache.

Internal Memory

The ADSP-21161 processor has 2 MBits of internal memory space; 1 Mbit is addressable. The 1 Mbit of memory is divided into two 0.5 Mbit blocks: Block 0 and Block 1. The additional 1 MBit of the memory space is reserved on the ADSP-21161 processor. Table 2-4 shows the maximum number of data or instruction words that can fit in each 0.5 MBit internal memory block.

Table 2-4. Words Per 0.5 MBit Internal Memory Block

Word Type	Bits Per Word	Maximum Number of Words Per 0.5 Mbit Block
Instruction	48-bits	10.67 Kwords
Long Word Data	64-bits	8 Kwords
Extended Precision Normal Word Data	40-bits	10.67 Kwords
Normal Word Data	32-bits	16 Kwords
Short Word Data	16-bits	32 Kwords

External Memory

While the processor's internal memory is divided into blocks, the processor's external memory spaces are divided into banks. The internal memory blocks and the external memory spaces may be addressed by either data address generator. External memory banks are fixed sizes that can be configured for various waitstate and access configurations.

There are 254 Mwords of external memory space that the processor can address. External memory connects to the processor's external port, which extends the processor's 24-bit address and 32-bit data buses off the processor. The processor can make 8, 16, 32, or 48-bit accesses to external memory for instructions and 8,16, or 32-bit accesses for data. Table 2-5 shows the access types and words for processor's external memory accesses. The processor's DMA controller automatically packs external data into the appropriate word width during data transfer.

The external data bus can be expanded to 48-bits if the link ports are disabled and the corresponding full width instruction packing mode (IPACK) is enabled in the SYSCON register. Ensure that link ports are disabled when executing code from external 48-bit memory.

Table 2-5. Internal-to-External Memory Word Transfers

Word Type	Transfer Type
Packed Instruction	32, 16, or 8- to 48-bit packing
Normal Word Data	32-bit word in 32-bit transfer
Short Word Data	Not supported

The total addressable space for the fixed external memory bank sizes depends on whether SDRAM or Non-SDRAM (such as SRAM, SBSRAM) is used. Each external memory bank for SDRAM can address 64 Mwords. For Non-SDRAM memory, each bank can address up to 16 Mwords. The remaining 48 Mwords are reserved. These reserved addresses for non-SDRAM accesses are aliased to the first 16 Mspaces within the bank.

TigerSHARC Memory Characteristics

As an example of the TigerSHARC memory architecture, the ADSP-TS101 processor has three internal memory blocks: M0, M1, and M2. Each memory block consists of 2 Mbits of memory space, and is configured as 64 kwords each 32 bits in width. There are three separate internal 128-bit data buses, each connected to one of the memory blocks. Memory blocks can store instructions and data interchangeably, with one access per memory block per cycle. If the programmer ensures that program and data are in different memory blocks, then data access can occur at the same time as program fetch. Therefore, in one cycle, up to three 128-bit transfers can occur within the core (two data transfers and one program instruction transfer).

The I/O Processor can use only one internal bus at a time, and the I/O Processor competes with the core for use of the internal bus. Therefore, in one cycle, the processor can fetch four 32-bit instructions, and load or store 256 bits of data (four 64-bit words or eight 32-bit words or sixteen 16-bit words or thirty-two 8-bit words).

The TigerSHARC processor 32-bit address bus provides an address space of four gigawords. This address space is common to a cluster of TigerSHARC processors that share the same cluster bus.

The zones in the memory space are made up of the following regions.

- External memory bank space—the region for standard addressing of off-chip memory (including SDRAM, MB0, MB1, and Host)
- External multiprocessor space—the on-chip memory of all other TigerSHARC processors connected in a multiprocessor system
- Internal address space—the region for standard internal addressing

In the example system, the ADSP-TS101 processor has internal memory addresses from `0x0` to `0x17FFFF`. Refer to Table 2-6.

Table 2-6. ADSP-TS101 Processor Memory Structure

Block	Range	Word Size
M0 memory block	`0x0000 0000 - 0x0000 FFFF`	32-bit instructions
	`0x0001 0000 - 0x0007 FFFF`	Reserved
M1 memory block	`0x0008 0000 - 0x0008 FFFF`	32-bit instructions
	`0x0009 0000 - 0x0009 FFFF`	Reserved
M2 memory block	`0x0010 0000 - 0x0010 FFFF`	32-bit instructions
	`0x0011 0000 - 0x0017 FFFF`	Reserved

Table 2-6. ADSP-TS101 Processor Memory Structure (Cont'd)

Block	Range	Word Size
Internal registers	`0x0018 0000 - 0x0018 07FF`	Control, status, and I/O registers. This cannot be used in `.LDF` files. Internal registers are memory accessible in MP space only.
	`0x0018 0800 - 0x01BF FFFF`	Reserved
	`0x01C0 0000 - 0x03FF FFFF`	Broadcast and multiprocessor (not used in `.LDF` file)
SDRAM	`0x0400 0000 - 0x07FF FFFF`	32-bit instructions

Blackfin Processor Memory Characteristics

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

ⓘ Other processors in the Blackfin processor family (ADSP-BF531, ADSP-BF532, ADSP-BF533 and ADSP-BF561 processors) have very different memory architectures. Refer to hardware reference manuals of target processors for appropriate information.

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

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

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

Table 2-7 lists memory ranges for the ADSP-BF535 processors. Blackfin processors have a 32-bit address range to support memory addresses from 0x0 to 0xFFFF FFFF. Address ranges that are not listed are reserved.

Table 2-7. ADSP-BF535 Processor Memory Map Addresses

Memory Range	Range Description
0xFFE00000 -0xFFFFFFFF	Core MMR registers (2MB)
0xFFC00000 -0xFFDFFFFF	System MMR registers (2MB)
0xFFB00000 -0xFFB00FFF	Scratchpad SRAM (4K)
0xFFA00000 -0xFFA03FFF	Instruction SRAM (16K)
0xFF900000 -0xFF903FFF	Data Memory Bank 2 SRAM (16K)
0xFF800000 -0xFF803FFF	Data Memory Bank 1 SRAM (16K)
0xF0040000 -0xFF7FFFFF	Reserved
0xF0000000 -0xF003FFFF	L2 Memory Bank SRAM (256K)
0xEF000400 -0xEFFFFFFF	Reserved
0xEF000000 -0xEF0003FF	Boot ROM (1K)
0x00000000 -0xEEFFFFFF	External memory

The MEMORY section in Listing 2-1 on page 2-38 assumes that only L1 and L2 SRAMs are available and that L1 is unused. Refer to the *VisualDSP++ C/C++ Compiler and Library Manual for Blackfin Processors* and the appropriate hardware reference for information about cache configuration.

ⓘ See the *Memory* chapter in an appropriate hardware reference for information about your target processor's memory organization.

Linker MEMORY{} Command in .LDF File

Referring to information in sections "Memory Usage and Default Memory Sections" and "Memory Characteristics Overview", you can specify the target's memory with the MEMORY{} command for any of target processor architectures (Listing 2-1, Listing 2-2, and Listing 2-3 provide code examples for specific processors).

Listing 2-1. ADSP-21161 MEMORY{} Command Code

```
MEMORY
{
seg_rth      { TYPE(PM RAM) START(0x00040000) END(0x000400ff)
               WIDTH(48) }
seg_init     { TYPE(PM RAM) START(0x00040100) END(0x000401ff)
               WIDTH(48) }
seg_int_code { TYPE(PM RAM) START(0x00040200) END(0x00040287)
               WIDTH(48) }
seg_pmco     { TYPE(PM RAM) START(0x00040288) END(0x000419ff)
               WIDTH(48) }
seg_pmda     { TYPE(PM RAM) START(0x00042700) END(0x00043fff)
               WIDTH(32) }
seg_dmda     { TYPE(DM RAM) START(0x00050000) END(0x00051fff)
               WIDTH(32) }
seg_heap     { TYPE(DM RAM) START(0x00052000) END(0x00052fff)
               WIDTH(32) }
seg_stak     { TYPE(DM RAM) START(0x00053000) END(0x00053fff)
               WIDTH(32) }
```

Listing 2-2. ADSP-TS101 MEMORY{} Command

```
MEMORY
{
      /*  Internal memory blocks are 0x10000 (64K bytes) */
      /*  Start of TS101_memory.ldf */

M0Code  {TYPE(RAM) START(0x00000000) END(0x0000FFFF) WIDTH(32)}
M1Data  {TYPE(RAM) START(0x00080000) END(0x0008BFFF) WIDTH(32)}
M1Stack {TYPE(RAM) START(0x0008C000) END(0x0008FFFF) WIDTH(32)}
```

```
M2Data  {TYPE(RAM) START(0x00100000) END(0x0010BFFF) WIDTH(32)}
M2Heap  {TYPE(RAM) START(0x0010C000) END(0x0010C7FF) WIDTH(32)}
M2Stack {TYPE(RAM) START(0x0010C800) END(0x0010FFFF) WIDTH(32)}
SDRAM   {TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)}
MS0     {TYPE(RAM) START(0x08000000) END(0x0BFFFFFF) WIDTH(32)}
MS1     {TYPE(RAM) START(0x0C000000) END(0x0FFFFFFF) WIDTH(32)}

      /*  end of TS101_memory.ldf file  */
}
```

Listing 2-3. ADSP-BF533 MEMORY{} Command Code

```
MEMORY        /* Define/label system memory     */
{             /* List of global Memory Segments */
   MEM_L2_CODE
        { TYPE(RAM) START(0xF0000000) END(0xF002FFFF) WIDTH(8) }
   MEM_L1_DATA_A
        { TYPE(RAM) START(0xFF800000) END(0xFF803FFF) WIDTH(8) }
   MEM_L1_DATA_B
        { TYPE(RAM) START(0xFF900000) END(0xFF903FFF) WIDTH(8) }
   MEM_HEAP
        { TYPE(RAM) START(0xF0030000) END(0xF0037FFF) WIDTH(8) }
   MEM_STACK
        { TYPE(RAM) START(0xF0038000) END(0xF003DFFF) WIDTH(8) }
   MEM_ARGV
        { TYPE(RAM) START(0xF003FE00) END(0xF003FFFF) WIDTH(8) }
   MEM_SDRAM0
        { TYPE(RAM) START(0x00000004) END(0x07FFFFFF) WIDTH(8) }
}
```

The above examples apply to the preceding discussion of how to write a `MEMORY{}` command and to the following discussion of the `SECTIONS{}` command. The `SECTIONS{}` command is not atomic; it can be interspersed with other directives, including location counter information. You can define new symbols within the `.LDF` file. These examples define the starting stack address, the highest possible stack address, and the heap's starting location and size. These newly-created symbols are entered in the executable's symbol table.

Placing Code on the Target

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

To write a SECTIONS{} command:

1. List all input sections defined in the source files.
   - **Assembly files.** List each assembly code .SECTION directive, identify its memory type (PM or CODE, or DM or DATA), and note when location is critical to its operation. These .SECTIONS portions include interrupt tables, data buffers, and on-chip code or data. (see "Specifying Two Buffers in Different Memory Segments" on page 2-44 for TigerSHARC-specific information.)
   - **C/C++ source files.** The compiler generates sections with the name "program" or "code" for code, and the names "data1" and "data2" for data. These sections correspond to your source when you do not specify a section by means of the optional section() extension.
2. Compare the input sections list to the memory segments specified in the MEMORY{} command. Identify the memory segment into which each .SECTION must be placed.
3. Combine the information from these two lists to write one or more SECTIONS{} commands in the .LDF file.

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

Listing 2-4 presents a SECTIONS{} command that would work with the MEMORY{} command in Listing 2-1.

Listing 2-4. ADSP-21161 SECTIONS{} Command in the .LDF File

```
SECTIONS
{
/*  Begin output sections  */
     seg_rth { // run-time header and interrupt table
         INPUT_SECTIONS( $OBJS(seg_rth) $LIBS(seg_rth))
         } >seg_rth
     seg_init { // Initialization
         ldf_seginit_space = . ;
         INPUT_SECTIONS( $OBJS(seg_init) $LIBS(seg_init))
         } >seg_init
     seg_init_code { // Initialization data
         INPUT_SECTIONS( $OBJS(seg_init_code) $LIBS(seg_init_code))
         } >seg_init_code
     seg_pmco { // PM code
         INPUT_SECTIONS( $OBJS(seg_pmco) $LIBS(seg_pmco))
         } >seg_pmco
     seg_pmda { // PM data
         INPUT_SECTIONS( $OBJS(seg_pmda) $LIBS(seg_pmda))
         } >seg_pmda
      .bss ZERO_INIT {
         INPUT_SECTIONS( $OBJS(.bss) $LIBS(.bss))
         } >seg_dmda
     seg_dmda { // DM data
         INPUT_SECTIONS( $OBJS(seg_dmda) $LIBS(seg_dmda))
         } >seg_dmda

     heap {
         // allocate a heap for the application
         ldf_heap_space = .;
         ldf_heap_length = MEMORY_SIZEOF(seg_heap);
         ldf_heap_end = ldf_heap_space + ldf_heap_length - 1;
     } > seg_heap;
}        // end sections
```

Listing 2-5 presents a `SECTIONS{}` command that would work with the `MEMORY{}` command in Listing 2-2.

Listing 2-5. ADSP-TS101 SECTIONS{} Command in the .LDF File

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

Listing 2-6 presents a SECTIONS{} command that would work with the MEMORY{} command in Listing 2-3.

Listing 2-6. ADSP-BF535 SECTIONS{} Command in the .LDF File

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

      L1_code
      {
         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_CODE

      program
      {
      // Align all code sections on 2 byte boundary
         INPUT_SECTION_ALIGN(4)
         INPUT_SECTIONS( $OBJECTS(program) $LIBRARIES(program))
         INPUT_SECTIONS( $OBJECTS(data1) $LIBRARIES(data1))
         INPUT_SECTIONS( $OBJECTS(cplb) $LIBRARIES(cplb))
```

```
        INPUT_SECTIONS( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
        INPUT_SECTIONS( $OBJECTS(cplb_data) $LIBRARIES(cplb_data))
        INPUT_SECTIONS( $OBJECTS(constdata) $LIBRARIES(constdata))
        INPUT_SECTIONS( $OBJECTS(voldata) $LIBRARIES(voldata))
      } >MEM_PROGRAM

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

     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 */
```

Specifying Two Buffers in Different Memory Segments

On TigerSHARC processors, the linker is enhanced to support efficient programming using the .SEPARATE_MEM_SEGMENTS assembler directive.

- The .SEPARATE_MEM_SEGMENTS assembler directive (or the compiler pragma #pragma separate_mem_segments) specifies two buffers directing the linker to place the buffers into different memory segments. For example,

```
.SECTION data1;
.VAR buf1;
.VAR buf2;
.EXTERN buf3;
.SEPARATE_MEM_SEGMENTS(buf1, buf2)
.SEPARATE_MEM_SEGMENTS(buf1, buf3)
```

- The set of available memory segments for each buffer is defined by using the linker's "one-to-many" feature—mapping the input section(s) that contain the buffer into multiple memory segments. For example,

```
data2 {
   INPUT_SECTIONS( $OBJECTS(data1) )
} >M2DataA

data4 {
   INPUT_SECTIONS( $OBJECTS(data1) )
} >M4DataA
```

- Linker tries to satisfy placement constraint requirements by allocating the buffers to different memory segments.

  a. If the linker fails to satisfy some or all of the requirements, the linker produces a warning.

  b. All symbols mentioned in .SEPARATE_MEM_SEGMENTS are mapped before anything else by the linker (with the exception of absolute placement).

c. Reference to symbol in .SEPARATE_MEM_SEGMENTS is a weak reference. If such symbol is defined in a library, linker does NOT bring the symbol from the library (unless the symbol is referenced directly or indirectly from an object file.

d. The linker ignores the cases where the symbol is mentioned in .SEPARATE_MEM_SEGMENTS assembler directive is undefined or is not defined in an input section (for example, as an LDF symbol).

See "*Pragmas*" in Chapter 1 of the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for TigerSHARC Processors* for more information.

Profile-Guided Optimization Support

The SHARC, TigerSHARC and Blackfin processor architectures support Profile-Guided Optimization (PGO). PGO is the process of gathering information about a running application over many invocations of the executable with different input data, and then re-optimizing it using that gathered information.

The process relies upon the same application being run with different data sets, which often means that the application acts upon sample data sets stored in files. More specifically, it means that the application is instructed to process each file via command-line options passed to main().

The .LDF files and the VisualDSP++ IDDE collaborate to provide support for command-line arguments. Under normal circumstances, a typical embedded program is not interested in command-line arguments, and receives none. In these normal cases, the run-time header invokes a function to parse a global string __argv_string[] and finds it empty.

To support PGO, the LDF option IDDE_ARGS can be used to define a memory section called MEM_ARGV, and __argv_string[] is mapped directly to the start of this section. The VisualDSP++ IDDE follows the convention that command-line arguments can be passed to an application by writing the argument string into memory starting at the beginning of MEM_ARGV.

Refer to *VisualDSP++ 4.0 C/C++ Compiler and Library Manual* for the appropriate processor architecture for more information on Profile-Guided Optimization.

Passing Arguments for Simulation or Emulation

This feature applies to Blackfin processors ONLY.

To support simulation and emulation in Blackfin processors, the linker should obtain the start address and buffer length of the argument list from the ARGV memory segment of the .LDF file (refer to "Example 1 – Basic .LDF File for Blackfin Processors" on page 3-6).

To set the address:

1. In the MEMORY{} section, add a line to define the MEM_ARGV section.

2. Add a command to define the ARGV section and the values for ldf_argv_space, ldf_argv_length, and ldf_argv_end.

Refer to the *VisualDSP++ 4.0 User's Manual* or the online Help for information about the simulator and command-line arguments.

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

Linker Command-Line Reference

This section provides reference information, including:

- "Linker Command-Line Syntax" on page 2-47
- "Linker Command-Line Switches" on page 2-51

When you use the linker via the VisualDSP++ IDDE, the settings on the **Link** tab of the **Project Options** dialog box correspond to linker command-line switches. VisualDSP++ calls the linker with these switches when linking your code. For more information, refer to the *VisualDSP++ 4.0 User's Manual* and the VisualDSP++ online Help.

Linker Command-Line Syntax

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

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

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

Use `-proc` *`processor`* instead of the deprecated `-Darchitecture` on the command line to select the target processor. See Table 2-9 on page 2-53 for more information.

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

The following are the example linker commands.

```
linker -proc ADSP-21161 p0.doj -T target.ldf -t -o program.dxe
linker -proc ADSP-TS201 p0.doj -T target.ldf -t -o program.dxe
linker -proc ADSP-BF535 p0.doj -T target.ldf -t -o program.dxe
```

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

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

- "Command-Line Object Files" on page 2-48
- "Command-Line File Names" on page 2-49
- "Object File Types" on page 2-51

Command-Line Object Files

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

- Standard object (`.DOJ`) files produced by the assembler
- One or more libraries (archives), each with a `.DLB` extension. Examples include the C run-time libraries and math libraries included with VisualDSP++. You may create libraries of common or specialized objects. Special libraries are available from DSP algorithm vendors. For more information, see Chapter 6, "Archiver".
- An executable (`.DXE`) file to be linked against. Refer to `$COMMAND_LINE_LINK_AGAINST` in "Built-In LDF Macros" on page 3-27.

Object File Names

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

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

If the file exists before the link begins, the linker opens the file to verify its type before processing the file. Table 2-8 lists valid file extensions used by the linker.

Command-Line File Names

Some linker switches take a file name as a parameter. Table 2-8 lists the types of files, names, and extensions that the linker expects on file name arguments. The linker follows the conventions for file extensions in Table 2-8.

Table 2-8. File Extension Conventions

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

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

1. Specified path – If the command line includes relative or absolute path information, the linker searches that location for the file.

2. Specified directories – If you do not include path information on the command line and the file is not in the default directory, the linker searches for the file in the search directories specified with the `-L` (path) command-line switch, and then searches directories specified by `SEARCH_DIR` commands in the `.LDF` file. Directories are searched in order of appearance on the command line or in the `.LDF` file.

3. Default directory – If you do not include path information in the `.LDF` file named by the `-T` switch, the linker searches for the `.LDF` file in the current working directory. If you use a default `.LDF` file (by omitting LDF information in the command line and instead specifying `-proc <processor>`), the linker searches in the processor-specific `LDF` directory; for example, ...`\$ADI_DSP\21161\ldf`.

For more information on file searches, see "Built-In LDF Macros" on page 3-27.

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

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

Object File Types

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

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

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

For more information on objects, see the `$COMMAND_LINE_OBJECTS` macro. For information on executables, see the `$COMMAND_LINE_LINK_AGAINST` macro. Both are described in "Built-In LDF Macros" on page 3-27.

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

Linker Command-Line Switches

This section describes the linker's command-line switches. Table 2-9 on page 2-53 briefly describes each switch with regard to case sensitivity, equivalent switches, switches overridden or contradicted by the one described, and naming and spacing constraints for parameters.

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

```
-switch [argument]
```

Rules

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

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

Example

```
linker -proc ADSP-BF535 p0.doj p1.doj p2.doj -T target.ldf -t -o program.dxe
```

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

- `-proc ADSP-BF535`
  Specifies the processor.
- `p0.doj`, `p1.doj`, and `p2.doj`
  Links three object files into an executable file.
- `-T target.ldf`
  Uses a secondary `.LDF` file to specify executable program placement.
- `-t`
  Turns on trace information, echoing each link object's name to `stdout` as it is processed.
- `-o program.dxe`
  Specifies a name of the linked executable file.

Typing `linker` without any switches displays a summary of command-line options. Using no switches is the same as typing `linker -help`.

Linker Switch Summary and Descriptions

Table 2-9 briefly describes each linker switch. Each individual switch is described in detail following this table. See "Project Builds" on page 2-6 for information on the VisualDSP++ **Project Options** dialog box.

Table 2-9. Linker Command-Line Switches – Summary

Switch	Description	More Info
`@file`	Uses the specified file as input on the command line	on page 2-55
`-DprocessorID`	Specifies the target processor ID. The use of `-proc processorID` is recommended.	on page 2-55
`-L path`	Adds the path name to search libraries for objects	on page 2-56
`-M`	Produces dependencies	on page 2-56
`-MM`	Builds and produces dependencies	on page 2-56
`-Map file`	Outputs a map of link symbol information to a file	on page 2-56
`-MDmacro[=def]`	Defines and assigns value `def` to a preprocessor macro	on page 2-57
`-MUDmacro`	Undefines the preprocessor macro	on page 2-57
`-S`	Omits debugging symbols from the output file	on page 2-57
`-T filename`	Names the LDF	on page 2-58
`-Wwarn number`	Demotes the specified error message to a warning	on page 2-58
`-Wnumber`	Selectively disables warnings by one or more message numbers. For example, `-W1010` disables warning message `li1010`.	on page 2-58
`-e`	Eliminates unused symbols from the executable	on page 2-59
`-ek secName`	Specifies a section name in which elimination should not take place	on page 2-59

Table 2-9. Linker Command-Line Switches – Summary (Cont'd)

Switch	Description	More Info
`-es` *secName*	Names input sections (*secName* list) to which elimination algorithm is applied	on page 2-59
`-ev`	Eliminates unused symbols verbosely	on page 2-59
`-flag-meminit`	Passes each comma-separated option to the Memory Initializer utility	on page 2-60
`-flag-pp`	Passes each comma-separated option to the preprocessor	on page 2-60
`-h` `-help`	Outputs the list of command-line switches and exits	on page 2-60
`-i` *path*	Includes search directory for preprocessor `include` files	on page 2-60
`-ip`	Fills fragmented memory with individual data objects that fit. **Note:** Not used with Blackfin processors.	on page 2-60
`-jcs21`	Converts out-of-range short calls and jumps to the longer form. **Note:** Blackfin processors only.	on page 2-61
`-jcs21+`	Enables `-jcs21` and allows the linker to convert out-of-range branches to indirect calls and jumps sequences **Note:** Blackfin processors only.	on page 2-62
`-keep` *symName*	Keeps symbols from being eliminated	on page 2-62
`-meminit`	Causes post-processing of the executable file	on page 2-62
`-nonmemcheck`	Turns off LDF memory checking	on page 2-62
`-o` *filename*	Outputs the named executable file	on page 2-63
`-od` *filename*	Specifies the output directory	on page 2-63
`-pp`	Stops after preprocessing	on page 2-63
`-proc` *processor*	Selects a target processor	on page 2-63
`-s`	Strips symbol information from the output file	on page 2-64

Table 2-9. Linker Command-Line Switches – Summary (Cont'd)

Switch	Description	More Info
`-save-temps`	Saves temporary output files	on page 2-64
`-si-revision` *version*	Specifies silicon revision of the specified processor	on page 2-64
`-sp`	Skips preprocessing	on page 2-65
`-t`	Outputs the names of link objects	on page 2-65
`-v` `-verbose`	Verbose: Outputs status information	on page 2-65
`-version`	Outputs version information and exits	on page 2-66
`-warnonce`	Warns only once for each undefined symbol	on page 2-66
`-xref` *filename*	Produces a cross-reference file	on page 2-66

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

@filename

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

-D*processor*

The `-D`*processor* (define processor) switch specifies the target processor (architecture); for example, `-DADSP-21062` or `-DADSP-TS201`.

ⓘ The `-proc` *processor* switch (on page 2-63) is a preferred option to be used as a replacement for the `-D`*processor* command line entry to specify the target processor.

White space is not permitted between -D and *processor*. The architecture entry is case sensitive and must be available in your VisualDSP++ installation. This switch must be used if no .LDF file is specified on the command line (see -T on page 2-58). This switch must be used if the specified .LDF file does not specify ARCHITECTURE(). Architectural inconsistency between this switch and the .LDF file causes an error.

-L *path*

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

-M

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

-MM

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

-Map *filename*

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

Opening an .xml map file in a Web browser provides an organized view of the map file. By using hyperlinks, it becomes easy to quickly find any relevant information. Since the format of .xml files can be extended between VisualDSP++ releases, the map file is dependant on particular installation of VisualDSP++. Thus, the .xml map file can be used only on a machine it was generated. In order to view the map file on a different machine, the file should be transformed to HTML format using the "xmlmap2html.exe" command-line utility. The utility makes it possible to view the map on virtually any machine with any browser.

-MD*macro*[=*def*]

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

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

-MUD*macro*

The -MUD*macro* (undefine macro) switch undefines the preprocessor macro where *macro* specifies a name. For example, -MUDTEST undefines macro TEST. The switch is processed after all -MDmacro switches have been processed. The -MUD*macro* switch may be repeated on the command line.

-S

The -S (strip debug symbol) switch directs the linker to omit debugging symbol information (*not* all symbol information) from the output file. Compare this switch with the -s switch on page 2-64.

-T *filename*

The `-T` *`filename`* (linker description file) switch directs the linker to use *`filename`* to name an `.LDF` file. The `.LDF` file specified following the `-T` switch must contain an `ARCHITECTURE()` command if the command line does not have `-proc` *`<processor>`*. The linker requires the `-T` switch when linking for a processor for which no VisualDSP++ support has been installed. In such cases, the processor ID does not appear in the **Target processor** field of the **Project Options** dialog box.

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

-Wwarn [*number*]

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

-W*number*[,*number*]

The `-Wnumber` or *`-wnumber`* (warning suppression) switch selectively disables warnings specified by one or more message numbers. For example, `-W1010` disables warning message `li1010`. This switch optionally accepts a list, such as `[,number ...]`.

-e

The -e switch directs the linker to eliminate unused symbols from the executable file.

> In order for the C and C++ run-time libraries to work properly, the following symbols should be retained with the "KEEP()" LDF command (described on page 3-33):
> ___ctor_NULL_marker and ___lib_end_of_heap_descriptions

-ek *sectionName*

The -ek *sectionName* (no elimination) switch specifies a section to which the elimination algorithm is not applied. Both this switch and the KEEP_SECTIONS() LDF command (on page 3-34) may be used to specify a section name in which elimination should *not* take place.

-es *sectionName*

The -es *sectionName* (eliminate listed section) switch specifies a section to which the elimination algorithm is to be applied. This switch restricts elimination to the named input sections. The -es switch may be used on a command line more than once. The default is all sections. Both this switch and the ELIMINATE_SECTIONS() LDF command (on page 3-32) may be used to specify sections from which unreferenced code and data are to be eliminated.

> In order for the C and C++ run-time libraries to work properly, the following symbols should be retained with the "KEEP()" LDF command (described on page 3-33):
> ___ctor_NULL_marker and ___lib_end_of_heap_descriptions

-ev

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

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

The `-flags-meminit` switch passes each comma-separated option to the Memory Initializer utility (see "Memory Initializer" for more information).

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

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

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

-h[elp]

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

-i|I *directory*

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

-ip

This switch is not used with Blackfin processors.

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

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

The assembler's `-noip` option turns off individual placement option.??????

-jcs2l

Used with Blackfin processors only.

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

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

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

Refer to the instruction set reference for target architecture for more information on jump and call instructions.

-jcs2l+

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

The `-jcs2l+` switch enables the `-jcs2l` switch and allows the linker to convert out-of-range branches (`0x800000` to `0x7FFFFF`) to indirect calls/jumps sequences using the `P1` register. This is used, for example, when a call from a function in L2 memory is made to a function in L1 memory.

-keep *symbolName*

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

-meminit

The `-meminit` (post-processing executable file) switch directs the linker to post-process the `.DXE` file through the Memory Initializer utility (see "Memory Initializer" for more information). This action causes the sections specified in the `.LDF` file to be "run-time" initialized by the C run-time library. By default, if this flag is not specified, all sections are initialized at "load" time (for example, via the VisualDSP++ IDDE or the boot loader). Refer to "SECTIONS{}" on page 3-48 for more information on section initialization.

-nonmemcheck

The `-nonmemcheck` (memory checking off) switch allows you to turn off memory checking in any `.LDF` file.

-o *filename*

The `-o` *`filename`* (output file) switch sets the value of `$COMMAND_LINE_OUTPUT_FILE` macro which is normally used as a parameter in LDF `OUTPUT()` command which specifies the output file name. If no `-o` is present on command line, the `$COMMAND_LINE_OUTPUT_FILE` macro gets a value of "`a.dxe`".

-od *directory*

The `-od` *`directory`* switch directs the linker to specify the value of the `$COMMAND_LINE_OUTPUT_DIRECTORY` LDF macro. This switch allows you to make a command-line change that propagates to many places without changing the `.LDF` file. Refer to "Built-In LDF Macros" on page 3-27.

-pp

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

-proc *processor*

The `-proc` *`processor`* (target processor) switch directs the linker to produce code suitable for the specified processor. For example,

```
linker -proc ADSP-TS201 p0.doj p1.doj p2.doj -o program.dxe
```

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

For example,

```
[proc]
Architecture: ADSP-TS201
```

See also “-si-revision version” for more information on silicon revision of the specified processor.

-s

The `-s` (strips all symbols) switch directs the linker to omit all symbol information from the output file.

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

-save-temps

The `-save-temps` switch directs the linker to save temporary (intermediate) output files.

-si-revision *version*

The `-si-revision` *`version`* (silicon revision) switch directs the assembler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter “*`version`*” represents a silicon revision of the processor specified by the `-proc` switch (on page 2-63). For example,

```
linker -proc ADSP-TS201 -si-revision 0.1
```

If silicon version “`none`” is used, then no errata workarounds are enabled, whereas specifying silicon version “`any`” will enable all errata workarounds for the target processor.

If the `-si-revision` switch is not used, the linker builds for the latest known silicon revision for the target processor and any errata workarounds which are appropriate for the latest silicon revision will be enabled.

If the silicon revision is set to "`any`", the `__SILICON_REVISION__` macro is set to `0xffff` and if the `-si-revision` switch is set to "`none`", the linker will not set the `__SILICON_REVISION__` macro.

The linker will pass the `-si-revision <silicon version>` switch when invoking another VisualDSP++ tool, for example when the linker invokes the assembler.

Example:

The TigerSHARC linker invoked as

```
linker -proc ADSP-TS201 -si-revision 0.1 …
```

invokes the assembler with

```
easmts -proc ADSP-TS201 -si-revision 0.1
```

-sp

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

-t

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

-v[erbose]

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

-version

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

-warnonce

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

-xref *filename*

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

3 LINKER DESCRIPTION FILE

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

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

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

This chapter contains:

The linker runs the preprocessor on the .LDF file, so you can use preprocessor commands (such as #defines) within the file. For information about preprocessor commands, refer to a *VisualDSP++ 4.0 Assembler and Preprocessor Manual* for an appropriate target processor architecture.

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

Refer to example DSP programs shipped with VisualDSP++ for sample .LDF files supporting typical system models.

LDF File Overview

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

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

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

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

The linker may output warning messages and error messages. You must resolve the error messages to enable the linker to produce valid output. See "Linker Warning and Error Messages" on page 2-15 for more information.

Default LDFs

The name of each .LDF file indicates the intended processor (for example, ADSP-BF531.ldf) and may include an optional suffix (for example, ADSP-BF533_C.ldf). If the .LDF file name has no suffix, it is the "default .LDF file". That is, when no .LDF file is explicitly specified, the default file is used to link an application when building for that processor. Therefore, ADSP-BF531.ldf is the default .LDF file for the ADSP-BF531 processor.

If no .LDF file is specified explicitly via the -T command-line switch, the compiler driver selects the default .LDF file for the target processor. For example, the first of the following commands uses the default .LDF file, and the second uses a user-specified file:

```
ccblkfn -proc ADSP-BF531 hello.c # uses default ADSP-BF531.ldf

ccblkfn -proc ADSP-BF531 hello.c -T ./my.ldf # uses ./my.ldf
```

For each processor, there are three .LDF files: with suffixes _C, _CPP, and _ASM, respectively (for example, ADSP-BF533_C.ldf).

These .LDF files are templates for the Expert Linker. If you use the Expert Linker to create a custom .LDF file for your project, the Expert Linker will do so by querying you for the kind of .LDF file you want to create (asm, C, or C++) and then copying one of the above templates. The suffixes indicate the kind of .LDF files they support.

The CPP template is a superset of the C template, and the C template is a superset of the ASM template. The differences are as follows:

- The CPP template links against C++ run-time libraries, C++ exception libraries, and the run-time headers built to initialize C++ constructors. It maps data sections that contain information controlling how thrown exceptions are caught.
- The C template is currently identical to the CPP template, since a C project may link against local or system libraries that have been implemented in C++; there may be differences in a future release.
- The ASM template does not include a run-time header, and does not permit command-line arguments to applications. The ASM template is not suitable for use with the compiler's Profile-Guided Optimization. Since the ASM template has no run-time header, it does not mandate a "start" symbol resolved to the Reset address. It does not map the C++ exception sections into memory.

Each .LDF file handles a variety of demands, allowing applications to be built in multiple configurations, merely by supplying a few command-line options. This flexibility is achieved by extensive use of preprocessor macros within the .LDF file. Macros serve as flags to indicate one choice or another, and as variables within the .LDF file to hold the name of a chosen file or other link-time parameter. This reliance on preprocessor operation can make the .LDF file seem an imposing sight.

In simple terms, different .LDF file configurations are selected by defining preprocessor macros on the linker command line. This can be specified from the **Link** tab of the VisualDSP++ IDDE's **Project Options** dialog box or directly from the command line.

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

See Listing 3-1 on page 3-6, Listing 3-2 on page 3-9, and Listing 3-3 on page 3-10 as examples of basic .LDF files for supported processors. See "Notes on Basic .LDF File Examples" on page 3-11 for basic information on LDF structure.

See Appendix C, Appendix D, and Appendix E for code examples for TigerSHARC, SHARC, and Blackfin processors, respectively.

Example 1 – Basic .LDF File for Blackfin Processors

Listing 3-1 is an example of a basic .LDF file for ADSP-BF535 processors (formatted for readability). Note the MEMORY{} and SECTIONS{} commands and refer to "Notes on Basic .LDF File Examples". Other .LDF file examples are provided in "LDF Programming Examples for Blackfin Processors". For more information, refer to EE-237, "*Guide to Blackfin Processor LDF Files*."

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

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

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

PROCESSOR P0 {   /*  the only processor in the system  */
   OUTPUT ( $COMMAND_LINE_OUTPUT_FILE )

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

      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

      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 */
```

Memory Usage in Blackfin Processors

The .LDF files define memory areas for all defined spaces on the processor.[1] Not all of these memory areas are used within the .LDF files. Instead, the .LDF files provide two basic memory configurations:

- The default configuration specifies that only internal memory is available and caching is disabled. Thus, no code or data is mapped to SDRAM unless explicitly placed there, and all of the available L1 space is used for code or data.
- The USE_CACHE option selects the alternative configuration, where code and data caches are enabled and external SDRAM is used. Code and data are mapped into L1 where possible, but the Cache/SRAM areas are left empty; any spill-over goes into the lower 32 MB area of SDRAM.

If USE_CACHE is used, caches may safely be turned on, because doing so will not corrupt code or data. Selecting this option does not actually enable the caches — that must be done separately (for example, through the ___cplb_ctrl configuration variable). Instead, this option ensures that the memory layout allows caches to be enabled later. Note that the SDRAM is not enabled automatically when USE_CACHE is enabled.

A common user error is to enable cache despite not having specified USE_CACHE, which would lead to code or data corruption as cache activity overwrites the contents of SRAM. Therefore, the .LDF files use the following "guard symbols":

```
___l1_code_cache
___l1_data_cache_a
___l1_data_cache_b
```

[1] With the exception of the core MMRs, which the linker considers "out of bounds".

These symbols are defined by the .LDF files and are given values (that is, resolved to addresses 0 or 1), depending on whether USE_CACHE is defined. The run-time library examines these symbols when cache configuration is requested, and refuses to enable a cache if the corresponding guard symbol is zero, indicating that valid information already occupies this space.

For more information, refer to *VisualDSP++ 4.0 C/C++ Compiler and Library Manual*, section "*Caching and Memory Protection*".

Example 2 – Basic .LDF File for TigerSHARC Processors

Listing 3-2 is an example of a basic .LDF file for the ADSP-TS101 processor (formatted for readability). Note the MEMORY{} and SECTIONS{} commands and refer to "Notes on Basic .LDF File Examples" on page 3-11. Other .LDF file examples are provided in "LDF Programming Examples for TigerSHARC Processors".

Listing 3-2. Example .LDF File for ADSP-TS201 Processor

```
ARCHITECTURE(ADSP-TS101)
SEARCH_DIR($ADI_DSP\TS\lib)
$OBJECTS = main.doj, $COMMAND_LINE_OBJECTS;

MEMORY {     /*  Define and label system memory  */
             /*  List of global memory segments  */
   M0Code   {TYPE(RAM) START(0x000000) END(0x00FFFF) WIDTH(32)}
   M1Data   {TYPE(RAM) START(0x080000) END(0x08FFFF) WIDTH(32)}
   M2Data   {TYPE(RAM) START(0x100000) END(0x10FFFF) WIDTH(32)}
}
PROCESSOR P0 {   /*  the only processor in the system  */
   OUTPUT ( $COMMAND_LINE_OUTPUT_FILE )
      SECTIONS{
         code  { INPUT_SECTIONS ( $OBJECTS(program))}   > M0Code
         data1 { INPUT_SECTIONS ( $OBJECTS(data1))}     > M1Data
         data2 { INPUT_SECTIONS ( $OBJECTS(data2))}     > M2Data
      }           /*  End of SECTIONS command for processor P0  */
}        /*  End of PROCESSOR command.  /*
```

Example 3 – Basic .LDF File for SHARC Processors

Listing 3-3 is an example of a basic .LDF file for the ADSP-21161 processor (formatted for readability). Note the MEMORY{} and SECTIONS{} commands and refer to "Notes on Basic .LDF File Examples" on page 3-11. Other examples for assembly and C source files are in "LDF Programming Examples for SHARC Processors".

Listing 3-3. Example .LDF File for ADSP-21161 Processor

```
//  Link for the ADSP-21161
ARCHITECTURE(ADSP-21161)
SEARCH_DIR ( $ADI_DSP\211xx\lib )
MAP (SINGLE-PROCESSOR.XML)   // Generate a MAP file

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

   // lib161.dlb is an ADSP-2116x-specific library and must precede
   // precede libc.dlb, C library to link 2116x-specific routines

$LIBS = lib161.dlb, libc.dlb;

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

   //  161_hdr.doj is the standard initialization file for 2116x
$OBJS = $COMMAND_LINE_OBJECTS, 161_hdr.doj;

   //  A linker project to generate a .DXE file
PROCESSOR P0
{
   OUTPUT ( .\SINGLE.DXE )   // The name of the output file

   MEMORY                   // Processor-specific memory command
```

```
      { INCLUDE("21161_memory.h") }

   SECTIONS                    // Specify the output sections
      {
      INCLUDE( "21161_sections.h" )
      }   // end P0 sections
}         // end P0 processor
```

Notes on Basic .LDF File Examples

In the following description, the `MEMORY{}` and `SECTIONS{}` commands connect the program to the target processor. For complete syntax information on LDF commands, see "LDF Commands" on page 3-29.

These notes describe features of a typical `.LDF` file (as presented in Listing 3-1, Listing 3-2, and Listing 3-3).

- `ARCHITECTURE(ADSP-xxxxx)` specifies the target architecture (processor). For example, `ARCHITECTURE(ADSP-TS201)`. The architecture dictates possible memory widths and address ranges, the register set, and other structural information for use by the debugger, linker, and loader. The target architecture must be installed in VisualDSP++.
- `SEARCH_DIR()` specifies directory paths searched for libraries and object files (on page 3-47). For example, the argument `$ADI_DSP\TS\lib` specifies one search directory for TigerSHARC libraries and object files.

  The linker supports a sequence of search directories presented as an argument list (directory1, directory2, ...). The linker follows this sequence and stops at the first match.
- `$LIBRARIES` is a list of the library and object files searched to resolve references, in the required order. Some of the options specify the selection of one library over another.

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

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

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

- $COMMAND_LINE_OBJECTS (on page 3-27) is an LDF *command-line macro*, which expands at the linker command line into the list of input files. Each linker invocation from the VisualDSP++ IDDE has a command-line equivalent. In the VisualDSP++ IDDE, $COMMAND_LINE_OBJECTS represents the .DOJ file of every source file in the VisualDSP++ **Project** window.

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

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

  The following (SHARC processor) examples are functionally identical.

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

$DOJS = main.doj, fft.doj;
dxe_program {
            INPUT_SECTIONS ($DOJS(program))

            } >mem_program;
```

- The `MEMORY{}` command (on page 3-35) defines the target system's physical memory and connects the program to the target system. Its arguments partition the memory into memory segments. Each memory segment is assigned a distinct name, memory type, a start and end address (or segment length), and a memory width. These names occupy different namespaces from input section names and output section names. Thus, a memory segment and an output section may have the same name.

- Each `PROCESSOR{}` command (on page 3-45) generates a single executable file.

- The `OUTPUT()` command (on page 3-46) produces an executable (`.DXE`) file and specifies its file name.

  In basic example, the argument to the `OUTPUT()` command is the `$COMMAND_LINE_OUTPUT_FILE` macro (on page 3-27). The linker names the executable file according to the text following the `-o` switch (which corresponds to the name specified in the **Project Options** dialog box when the linker is invoked via the VisualDSP++ IDDE).

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

- `SECTIONS{}` (on page 3-48) specifies the placement of code and data in physical memory. The linker maps input sections (in object files) to output sections (in executable files), and maps the output sections to memory segments specified by the `MEMORY{}` command.

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

For example, in TigerSHARC processors, the following INPUT_SECTIONS() statement directs the linker to place the program input section in the code output section and to map it to the M0Code memory segment.

```
code   { INPUT_SECTIONS ( $OBJECTS(program))}  > M0Code
```

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

```
dxe_isr{ INPUT_SECTIONS ( $OBJECTS (isr_tbl) ) } > mem_isr
```

For Blackfin processors, the following two input sections (program and data1) are mapped into one memory segment (L2), as shown below.

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

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

"DXE_L2", mapping them to the memory segment "MEM_L2". The two pieces of code follow each other in the program memory segment.

The INPUT_SECTIONS() commands are processed in the same order as object files appear in the $OBJECTS macro. You may intersperse INPUT_SECTIONS() statements within an output section with other directives, including location counter information.

LDF Structure

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

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

  Declare common (shared) memory definitions in the global scope before the PROCESSOR{} commands. See "Command Scoping" for more information.

Comments in the .LDF File

C-style comments may cross "newline" boundaries until a */ terminator is encountered.

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

For more information on LDF structure, see:

- "Link Target Description" on page 2-16
- "Placing Code on the Target" on page 2-40

Also see Appendix C, Appendix D, and Appendix E for code and LDF structure examples for TigerSHARC, SHARC and Blackfin processors, respectively.

Command Scoping

The two LDF scopes are *global* and *command*. A *global scope* occurs outside commands. Commands and expressions that appear in the global scope are always available and are visible in all subsequent scopes. LDF macros are available globally, regardless of the scope in which the macro is defined (see “LDF Macros” on page 3-26).

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

Figure 3-1 illustrates some scoping issues. For example, the MEMORY{} command that appears in the LDF’s global scope is available in all command scopes, but the MEMORY{} command that appear in command scopes is restricted to those scopes.

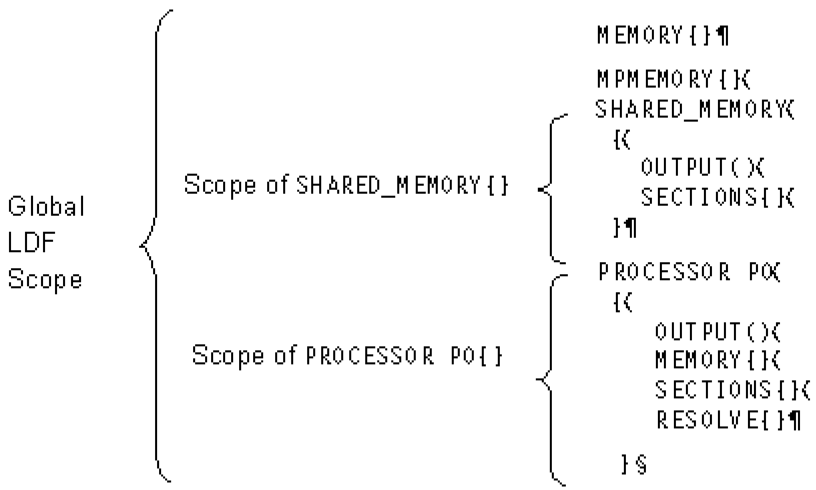


Figure 3-1. LDF Command Scoping Example

LDF Expressions

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

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

Table 3-1. Valid Items in Expressions

Convention	Description
.	Current location counter (a period character in an address expression). See "Location Counter (.)" on page 3-25.
`0x`*`number`*	Hexadecimal number (a 0x prefix)
`number`	Decimal number (a number without a prefix)
number`k` or *number*`K`	A decimal number multiplied by 1024
`B#`number or `b#`number	A binary number

LDF Keywords, Commands, and Operators

Table 3-2 lists .LDF file keywords (used in Blackfin, SHARC and TigerSHARC processor families). Descriptions of LDF keywords, operators, macros, and commands are provided in the following sections.

- "Miscellaneous LDF Keywords" on page 3-20
- "LDF Operators" on page 3-21
- "LDF Macros" on page 3-26
- "LDF Commands" on page 3-29

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

Table 3-2. LDF File Keywords Summary

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

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

PLIT_SYMBOL_ADDRESS	PLIT_SYMBOL_OVERLAYID	PM[1]
PROCESSOR	RAM	RESOLVE
RESOLVE_LOCALLY	ROM	SEARCH_DIR
SECTIONS	SHARED_MEMORY	SHT_NOBITS
SIZE	SIZEOF	SROM[1]
START	TYPE	VERBOSE
WIDTH	XREF	

1 Supported on ADSP-21xxx processors only.

Miscellaneous LDF Keywords

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

Table 3-3. Miscellaneous LDF File Keywords

Keyword	Description
FALSE	A constant with a value of 0
TRUE	A constant with a value of 1
XREF	A cross-reference option setting. See "-xref filename" on page 2-66.

For more information about other .LDF file keywords, see "LDF Operators" on page 3-21, "LDF Macros" on page 3-26, and "LDF Commands" on page 3-29.

LDF Operators

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

Each LDF operator is described in the following sections.

ABSOLUTE() Operator

Syntax:

```
ABSOLUTE(expression)
```

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

- A symbolic expression in parentheses; for example:

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

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

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

- An integer constant in one of these forms: hexadecimal, decimal, or decimal optionally followed by "K" (kilo [x1024]) or "M" (Mega [x1024x1024])

- A period, indicating the current location (see "Location Counter (.)" on page 3-25)

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

  ```
  ldf_stack_space = .;
  ```

can also be written as:

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

- A symbol name

ADDR() Operator

Syntax:

```
ADDR(section_name)
```

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

Blackfin Code Example:
If an .LDF file defines output sections as,

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

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

the .LDF file may contain the command:

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

The linker generates the constant ldf_start_L2 and assigns it the start address of the dxe_L2 output section.

SHARC Code Example:
If an .LDF file defines output sections as,

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

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

the .LDF file may contain the command:

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

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

DEFINED() Operator

Syntax:

```
DEFINED(symbol)
```

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

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

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

MEMORY_SIZEOF() Operator

Syntax:

```
MEMORY_SIZEOF(segment_name)
```

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

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

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

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

SIZEOF() Operator

Syntax:

```
SIZEOF(section_name)
```

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

SHARC Code Example:
The following LDF fragment defines the _sizeofdata1 constant to the size of the seg_dmda section.

```
seg_dmda
{
   INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
```

```
        _sizeofdata1 = SIZEOF(seg_dmda);
    } > seg_dmda
```

TigerSHARC Code Example:
The following LDF fragment defines the `_sizeofdata1` constant to the size of the `data1` section.

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

Location Counter (.)

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

Observe these rules:

- Use a period anywhere a symbol is allowed in an expression.
- Assign a value to the period operator to move the location counter and to leave voids or gaps in memory.
- The location counter may not be decremented.

LDF Macros

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

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

The linker:

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

LDF macros funnel input from the linker command line into predefined macros and provide support for user-defined macro substitutions. Linker macros are available globally in the `.LDF` file, regardless of where they are defined. For more information, see "Command Scoping" on page 3-17 and "LDF Macros and Command-Line Interaction" on page 3-28.

ⓘ LDF macros are independent of preprocessor macro support, which is also available in the `.LDF` file. The preprocessor places preprocessor macros (or other preprocessor commands) into source files. Preprocessor macros repeat instruction sequences in your source code or define symbolic constants. These macros facilitate text replacement, file inclusion, and conditional assembly and compilation. For example, the assembler's preprocessor uses the `#define` command to define macros and symbolic constants.

For more information, refer to the *VisualDSP++ 4.0 Compiler and Library Manual* and the *VisualDSP++ 4.0 Assembler and Preprocessor Manual* for appropriate target processors.

Built-In LDF Macros

The linker provides the following built-in LDF macros.

- `$COMMAND_LINE_OBJECTS`

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

- `$COMMAND_LINE_LINK_AGAINST`

  This macro expands into the list of executable (`.DXE` or `.SM`) files that one input on the linker's command line. This macro provides a comprehensive list of executable file input that the linker searches to resolve external symbols.

- `$COMMAND_LINE_OUTPUT_FILE`

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

- `$COMMAND_LINE_OUTPUT_DIRECTORY`

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

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

- `$ADI_DSP`

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

User-Declared Macros

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

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

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

LDF Macros and Command-Line Interaction

The linker receives commands through a command-line interface, regardless of whether the linker runs automatically from the VisualDSP++ IDDE or explicitly from a command window. Many linker operations, such as input and output, are controlled through the command-line entries. Use LDF macros to apply command-line inputs within the LDF.

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

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

LDF Commands

Commands in the .LDF file (called LDF commands) define the target system and specify the order in which the linker processes output for that system. LDF commands operate within a scope, influencing the operation of other commands that appear within the range of that scope. For more information, see "Command Scoping" on page 3-17.

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

ALIGN()

The `ALIGN(number)` command aligns the address of the current location counter to the next address that is a multiple of *number*, where *number* is a power of 2. The *number* is a word boundary (address) that depends on the word size of the memory segment in which the `ALIGN()` takes action.

ARCHITECTURE()

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

The command's syntax is:

```
ARCHITECTURE(processor)
```

The `ARCHITECTURE()` command is case sensitive. For example, valid entry may be `ADSP-TS201`. Thus, `ADSP-TS201` is valid, but `adsp-TS201` is not.

If the `ARCHITECTURE()` command does not specify the target processor, you must identify the target processor via the linker command line (`linker -proc processor ...`). Otherwise, the linker cannot link the program.

If processor-specific `MEMORY{}` commands in the `.LDF` file conflict with the processor type, the linker issues an error message and halts.

> ⓘ Test whether your VisualDSP++ installation accommodates a particular processor by typing the following linker command.
>
> ```
> linker -proc processor
> ```
>
> If the architecture is not installed, the linker prints a message to that effect.

ELIMINATE()

The ELIMINATE() command enables object elimination, which removes symbols from the executable file if they are not called. Adding the VERBOSE keyword, ELIMINATE(VERBOSE), reports on objects as they are eliminated. This command performs the same function as the -e command-line switch (see on page 2-59).

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

> ⓘ For the C and C++ run-time libraries to work properly, retain the following symbols with "KEEP()" (on page 3-33):
> ___ctor_NULL_marker and ___lib_end_of_heap_descriptions

In order to allow efficient elimination, the structure of the assembly source has to be such that the linker can unambiguously identify the boundaries of each "source object" in the input section (a "source object" is a function or a data item). Specifically, an input section must be fully covered by non-overlapping source objects with explicit boundaries. The boundary of a function item is specified by the function label and its corresponding ".end" label. If an input section layout does not conform to the rule described above, no elimination is performed in the section. See the *VisualDSP++ 4.0 Assembler and Preprocessor Manual* for more details on using ".end" labels.

ELIMINATE_SECTIONS()

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

The *`sectionList`* is a comma-delimited list of input sections. Both this LDF command and the linker's `-es` command-line switch (on page 2-59) may be used to specify sections where unreferenced code and data should be eliminated.

INCLUDE()

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

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

INPUT_SECTION_ALIGN()

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

The linker fills empty spaces created by `INPUT_SECTION_ALIGN()` commands with zeros (by default), or with the value specified with the preceding `FILL` command valid for the current scope. See `FILL` under "SECTIONS{}" on page 3-48.

The INPUT_SECTION_ALIGN() command is valid only within the scope of an output section. For more information, see "Command Scoping" on page 3-17. For more information on output sections, see the syntax description for "SECTIONS{}" on page 3-48.

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

```
SECTIONS
{
     program
     {
     INPUT_SECTION_ALIGN(2)

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

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

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

     } >MEM_PROGRAM
 }
```

KEEP()

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

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

> For the C and C++ run-time libraries to work properly, retain the following symbols with KEEP:
> ___ctor_NULL_marker and ___lib_end_of_heap_descriptions
> A symbol specified in *keeplist* must be a global symbol.

KEEP_SECTIONS()

The linker uses the KEEP_SECTIONS() command to specify a section name in which elimination *should not* take place. This command can appear anywhere the ELIMINATE_SECTION command appears. You may either use the KEEP_SECTIONS() command or the -ek linker switch (on page 2-59).

LINK_AGAINST()

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

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

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

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

where:

- Pn is the processor name; for example, P0 or P1.
- *executable_file_names* is a list of one or more executable (.DXE) or shared memory (.SM) files. Separate multiple file names with commas. However, Expert Linker allows the use of white spaces to separate multiple file names.

The linker searches the executable files in the order specified in the LINK_AGAINST() command. When a symbol's definition is found, the linker stops searching. Override the search order for a specific variable or label by using the RESOLVE() command (see "RESOLVE()" on page 3-47), which directs the linker to use the specified resolver, thus ignoring LINK_AGAINST() for a specific symbol. The LINK_AGAINST() command for other symbols still applies.

MAP()

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

The MAP(*filename*) command corresponds to and may be overridden by the linker's -Map <*filename*> command-line switch (on page 2-56). In VisualDSP++, if project options (**Link** tab of the **Project Options** dialog box) specify the generation of a symbol map, the linker runs with -Map <*projectname*>.xml asserted and the LDF's MAP() command generates a warning.

MEMORY{}

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

The LDF must contain a `MEMORY{}` command for global memory on the target system and may contain a `MEMORY{}` command that applies to each processor's scope. There is no limit to the number of memory segments you can declare within each `MEMORY{}` command. For more information, see "Command Scoping" on page 3-17.

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

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

The syntax for the `MEMORY{}` command appears in Figure 3-2, followed by a description of each part of a *`segment declaration`*.

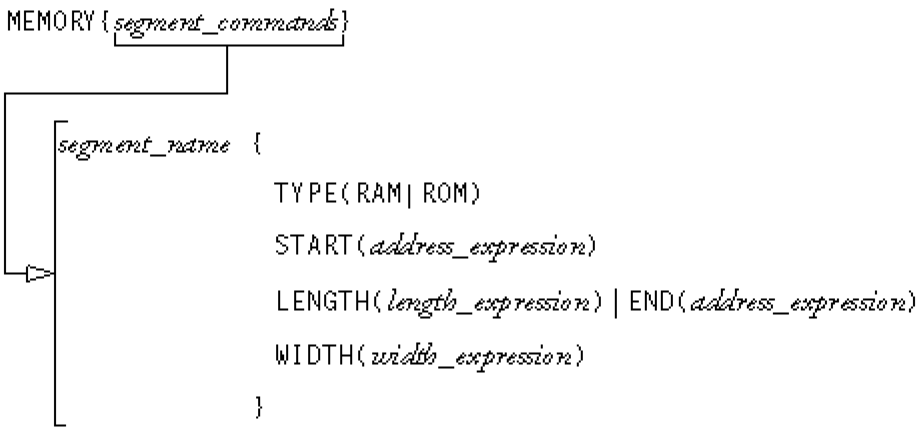


Figure 3-2. MEMORY{} Command Syntax Tree

Segment Declarations

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

Each *segment declaration* must contain a *segment_name*, TYPE(), START(), LENGTH() or END(), and a WIDTH(). Parts of a segment declaration are described as:

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

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

- TYPE()
  Identifies the architecture-specific type of memory within the memory segment.

  Note: Not all target processors support all types of memory. The linker stores this information in the executable file for use by other development tools.

  - For Blackfin and TigerSHARC processors, use TYPE() to specify the functional or hardware locus (RAM or ROM). The RAM declarator specifies segments that need to be booted. ROM segments are not booted; they are executed/loaded directly from off-chip PROM space.

- For SHARC (ADSP-21xxx) processors, use `TYPE()`to specify two parameters: memory usage (PM for program memory or DM for data memory), and functional or hardware locus (RAM or ROM, as described above).

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

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

  For Blackfin processors, width must be 16 (bits); for TigerSHARC processors, width must be 32 (bits); and for SHARC processors, width may be 8, 16, 32, 48 or 64 (bits).

MPMEMORY{}

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

Refer to "MPMEMORY{}" on page 5-30 for a detailed description of the `MPMEMORY{}` command.

OVERLAY_GROUP{}

The `OVERLAY_GROUP{}` command is deprecated. This command provides support for defining a set of overlays that share a block of run-time memory.

For detailed command description, refer to "OVERLAY_GROUP{}" on page 5-32. Refer to "Memory Management Using Overlays" on page 5-4 for a detailed description of overlay functionality.

PACKING()

In VisualDSP++ 4.0, the `PACKING()` command is used with ADSP-21xxx (SHARC) processors (as described in "Packing in SHARC Processors" on page 3-41) .

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

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

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

Syntax

The syntax of the `PACKING()` command is:

```
PACKING (number_of_bytes byte_order_list)
```

where:

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

For example, in SHARC processors, the first byte is B1 (not B0). The second byte is B2, and so on.

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

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

The order used to place bytes in memory correlates to the order the processor may use while unpacking the data when the processor transfers data from external memory into its internal memory. The processor's unpacking order can relate to the transfer method.

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

Packing in SHARC Processors

On SHARC processors, PACKING() applies to the processor's external port. Each external port buffer contains data packing logic that allows the packing of 8-, 16-, or 32-bit external bus words into 32- or 48-bit internal words. This logic is fully reversible.

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

In some Direct Memory Access (DMA) modes, SHARC processors unpack three 32-bit words to build two 48-bit instruction words when the processor receives data from 32-bit memory. For example, the unpacked order and storage order (Table 3-4) could apply to a DMA mode.

Table 3-4. DMA Packing Order

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

The order of unpacked bytes does **not** match the transfer (stored) order. Because the processor uses two bytes per short word, the above transfer translates into the format in Table 3-5.

Table 3-5. Storage Order vs. Unpacked Order

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

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

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

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

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

Overlay Packing Formats in SHARC Processors

Use the `PACKING()` command when:

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

The linker word-aligns the packing instruction as needed.

Table 3-6 indicates packing format combinations for SHARC DMA overlays available under each of the two operations.

Table 3-7 indicates packing format combinations for ADSP-21161N overlays available for storage in 8-bit-wide memory; 8-bit packing is available on ADSP-2106x and ADSP-21160 processors during EPROM booting only.

Table 3-6. Packing Formats for SHARC DMA Overlays

Execution Memory type	Storage Memory type	Packing Instruction
32-bit PM	16-bit DM	`PACKING(6 B0 B0 B1 B2 B5 B0 B0 B3 B4 B6)`
32-bit DM	16-bit DM	`PACKING(4 B0 B0 B1 B2 B0 B0 B3 B4 B5)`
48-bit PM	16-bit DM	`PACKING(6 B0 B0 B1 B2 B0 B0 B0 B3 B4 B0 B0 B0 B5 B6 B0)`
48-bit DM	32-bit DM	`PACKING(12 B1 B2 B3 B4 B0 BB5 B6 B11 B12 B0 B7 B8 B9 B10 B0)`

Table 3-7. Additional Packing Formats for DMA Overlays

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

External Execution Packing in SHARC Processors

The only two processors that require packed memory for external execution are the ADSP-21161N and the ADSP-21065L chips. The ADSP-21161N processor supports 48-, 32-, 16-, and 8-bit-wide external memory. The ADSP-21065L processor supports 32-bit external memory only.

ⓘ Previous to VisualDSP++ 3.5, it was required to use "packing" commands in the .LDF file to cause the code to be placed properly. In VisualDSP++ 3.5 and latter releases, the VisualDSP++ tools are enhanced to perform packing automatically.

In order for the VisualDSP++ tools to execute packing directly from external memory on ADSP-21065L and ADSP-21161N processors, the tools "pack" the code into the external memory providing the following conditions are met:

1. Ensure the "type" of the external memory is PM (Program Memory)
2. Ensure the data width matches the "real/actual" memory width: ADSP-21065L processor – 32 bits; ADSP-21161N processor – 48, 32, 16 and 8 bits
3. If the .LDF file has the PACKING() command for the particular section, remove the command.

When defining memory segments (required for external memory), the "type" of a memory section is recommended to be:

- PM – code or 40-bit data (data requires PX register to access)
- DM – all other sections

Width should be the "actual/physical" width of the external memory.

PLIT{}

The PLIT{} (procedure linkage table) command in an .LDF file inserts assembly instructions that handle calls to functions in overlays. The PLIT{} commands provide a template from which the linker generates assembly code when a symbol resolves to a function in overlay memory.

Refer to "PLIT{}" on page 5-36 for a detailed description of the PLIT{} command. Refer to "Memory Management Using Overlays" on page 5-4 for a detailed description of overlay and PLIT functionality.

PROCESSOR{}

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

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

The syntax for the PROCESSOR{} command appears in Figure 3-3.

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

Figure 3-3. PROCESSOR{} Command Syntax Tree

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

- *`processor_name`*

  Assigns a name to the processor. Processor names follow the same rules as linker labels. For more information, see "LDF Expressions" on page 3-18.

- `OUTPUT(`*`file_name`*`.DXE)`

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

- `MEMORY{`*`segment_commands`*`}`

  Defines memory segments that apply only to this specific processor. Use command scoping to define these memory segments outside the `PROCESSOR{}` command. For more information, see "Command Scoping" on page 3-17 and "MEMORY{}" on page 3-35.

- `PLIT{`*`plit_commands`*`}`

  Defines procedure linkage table (PLIT) commands that apply only to this specific processor. For more information, see "PLIT{}" on page 3-45.

- `SECTIONS{`*`section_commands`*`}`

  Defines sections for placement within the executable (`.DXE`) file. For more information, see "SECTIONS{}" on page 3-48.

- `RESOLVE{`*`symbol, resolver`*`}`

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

RESOLVE()

Use the RESOLVE(*symbol_name*, *resolver*) command to ignore a LINK_AGAINST() command for a specific symbol. This command overrides the search order for a specific variable or label. Refer to the "LINK_AGAINST()" on page 3-34 for more information.

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

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

- When the symbol is not defined in the current processor scope, the <*resolver*> supplies a file name, overriding any LINK_AGAINST().
- When the symbol is defined in the current processor scope, the <*resolver*> supplies to the linker the symbol location address.

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

SEARCH_DIR()

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

The search order follows the order of the listed directories. This command appends search directories to the directory selected with the linker's -L command-line switch (on page 2-56). Place this command at the beginning of the .LDF file to ensure that the linker applies the command to all file searches.

Example

```
ARCHITECTURE (ADSP-21161)
MAP (SINGLE-PROCESSOR.XML)          // Generate a MAP file

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

SECTIONS{}

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

An `.LDF` file may contain one `SECTIONS{}` command within each of the `PROCESSOR{}` commands. The `SECTIONS{}` command must be preceded by a `MEMORY{}` command, which defines the memory segments in which the linker places the output sections. Though an `.LDF` file may contain only one `SECTIONS{}` command within each processor command scope, multiple output sections may be declared within each `SECTIONS{}` command.

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

`expressions`
  or
`section_declarations`

Use *`expressions`* to manipulate symbols or to position the current location counter. Refer to "LDF Expressions" on page 3-18.

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

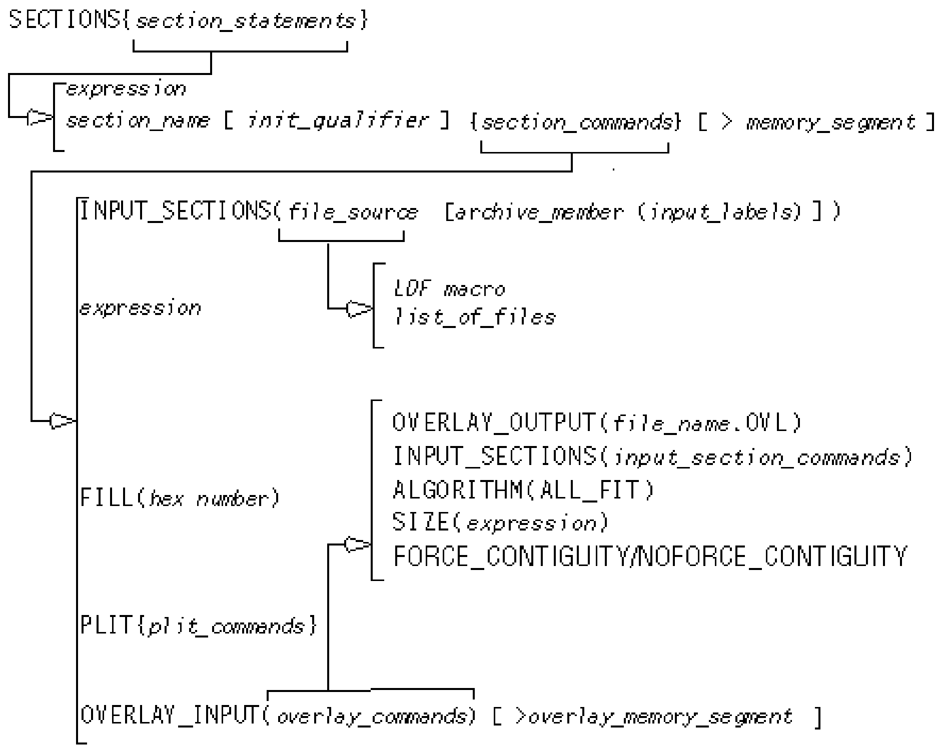


Figure 3-4. SECTIONS{} Command Syntax Tree

Parts of a `SECTION` declaration are:

- *`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 LDF keywords.

  The special section name `.PLIT` indicates the procedure linkage table (PLIT) section that the linker generates when resolving symbols in overlay memory. Place this section in non-overlay memory to manage references to items in overlay memory.

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

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

  The qualifiers are:

  - NO_INIT – Contains un-initialized data. There is no data stored in the .DXE file for this section (equivalent to SHT_NOBITS legacy qualifier).
  - ZERO_INIT – Contains only "zero-initialized" data. If invoked with the -meminit switch (on page 2-62), the "zeroing" of the section is done at runtime by the C run-time library. If -meminit is not specified, the "zeroing" is done at "load" time.
  - RUNTIME_INIT – If the linker is invoked with the -meminit switch, this section fills at runtime. If -meminit is not specified, the section fills at "load" time.

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

  "INPUT_SECTIONS()" on page 3-51
  "expression" on page 3-52
  "FILL(hex number)" on page 3-52
  "PLIT{plit_commands}" on page 3-52
  "OVERLAY_INPUT{overlay_commands}" on page 3-53

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

  The *memory_segment* is optional. Some sections, such as those for debugging, need not be included in the memory image of the executable file, but are needed for other development tools that read the executable file.

  By omitting a memory segment assignment for a section, you direct the linker to generate the section in the executable, but prevent section content from appearing in the memory image of the executable file.

INPUT_SECTIONS()

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

The command syntax is:

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

Note that spaces are significant in this syntax.

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

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

- *input_labels* are derived from run-time .SECTION names in assembly programs (for example, program). Delimit the list of names with commas.

Example

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

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

expression

In a *section_command*, an *expression* manipulates symbols or positions the current location counter. See "LDF Expressions" on page 3-18 for details.

FILL(hex number)

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

The FILL() command is used only within a section declaration.

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

```
FILL (0x0)
```

or

```
FILL (0xFFFF)
```

PLIT{plit_commands}

In a *section_command*, a PLIT{} command declares a locally-scoped procedure linkage table (PLIT). It contains its own labels and expressions. For more information, see "PLIT{}" on page 5-36.

OVERLAY_INPUT{overlay_commands}

In a *section_command*, OVERLAY_INPUT{} identifies the parts of the program to place in an overlay executable (.OVL) file. For more information on overlays, see "Memory Management Using Overlays" on page 5-4 and "OVERLAY_GROUP{}" on page 5-32. For overlay code examples, see "Linking for Overlay Memory" on page C-12 and "Linking for Overlay Memory" on page D-13.

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

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

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

- DEFAULT_OVERLAY()
  When the DEFAULT_OVERLAY() command is used, the linker initially places the overlay in the run-time space (that is, without running the overlay manager). .

- OVERLAY_OUTPUT()
  Outputs an overlay (.OVL) file for the overlay with the specified name. The OVERLAY_OUTPUT() in an OVERLAY_INPUT{} command must appear before any INPUT_SECTIONS() for that overlay.

- INPUT_SECTIONS()
  Has the same syntax within an OVERLAY_INPUT{} command as when it appears within an output_section_command, except that a .PLIT section may not be placed in overlay memory. For more information, see "INPUT_SECTIONS()" on page 3-51.
- OVERLAY_ID()
  Returns the overlay ID.
- NUMBER_OF_OVERLAYS()
  Returns the number of overlays that the current link generates when the FIRST_FIT or BEST_FIT overlay placement for ALGORITHM() is used.

  Note: Not currently available.
- ALGORITHM()
  Directs the linker to use the specified overlay linking algorithm. The only currently available linking algorithm is ALL_FIT.

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

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

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

- SIZE()
  Sets an upper limit to the size of the memory that may be occupied by an overlay.
- FORCE_CONTIGUITY
  Forces contiguous placement of sections. NOFORCE_CONTIGUITY suppresses a linker warning about non-contiguous placement of sections in the operating system.

SHARED_MEMORY{}

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

For more information, see "SHARED_MEMORY{}" on page 5-40.

4 EXPERT LINKER

The linker (`linker.exe`) combines object files into a single executable object module. Using the linker, you can create a new Linker Description File (LDF), modify an existing `.LDF` file, and produce an executable file (files). The linker is described in Chapter 2, "Linker", of this manual.

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

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

This chapter contains:

Expert Linker Overview

Expert Linker is a graphical tool that allows you to:

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

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

ⓘ Use default .LDF files that come with VisualDSP++, or use the Expert Linker interactive wizard to create new .LDF files.

When opening Expert Linker in a project that has an existing .LDF file, Expert Linker parses the .LDF file and graphically displays the target's memory map and the object mappings. The memory map displays in the **Expert Linker** window (Figure 4-1).

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

Expert Linker is able to show graphically how much space is allocated for program heap and stack. After you load and run the program, Expert Linker can show how much of the heap and stack has been used. You can interactively reduce the amount of space allocated to heap or stack if they are using too much memory. Freeing up memory enables you to store other things like processor code or data.

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

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

The **Expert Linker** window appears.

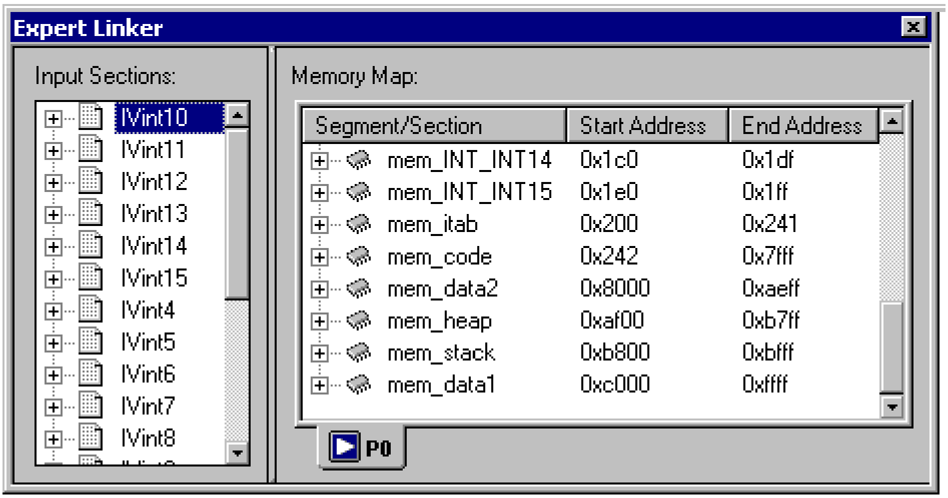


Figure 4-1. Expert Linker Window

Launching the Create LDF Wizard

From the VisualDSP++ main menu, choose **Tools -> Expert Linker -> Create LDF** to invoke a wizard for creating and customizing a new `.LDF` file. Use the **Create LDF** option when creating a new project.

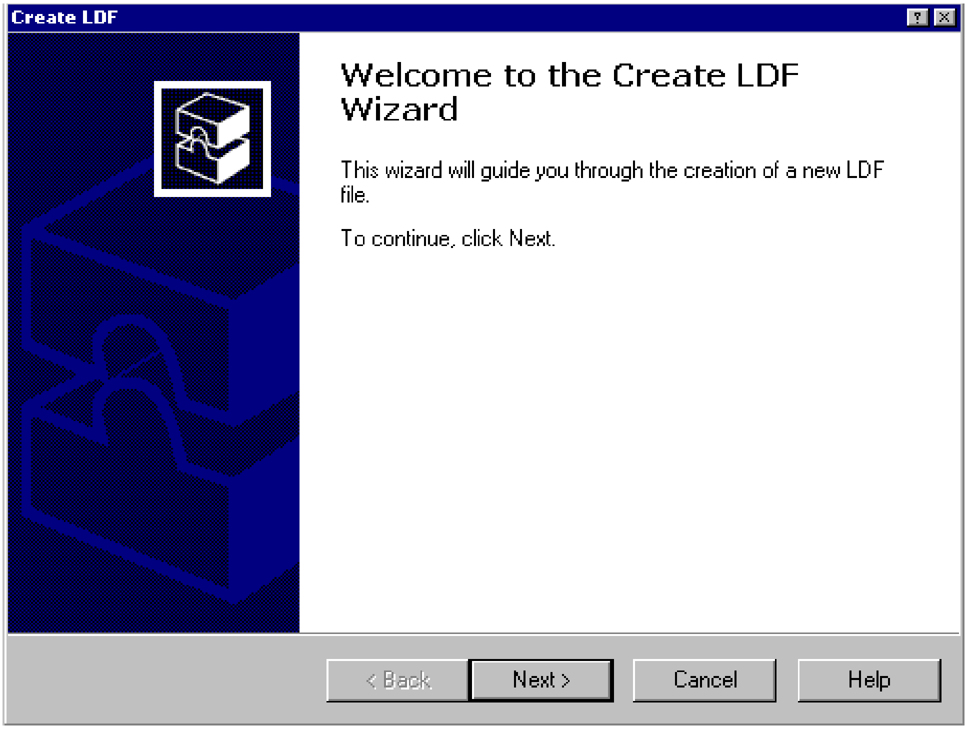


Figure 4-2. Welcome Page of the Create LDF Wizard

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

Step 1: Specifying Project Information

The first wizard window is displayed.

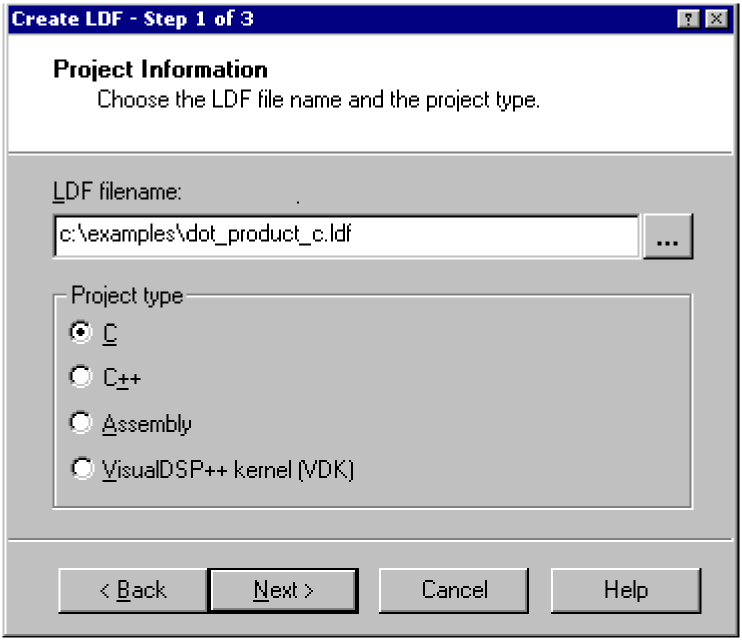


Figure 4-3. Selecting File Name and Project Type

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

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

For a case where there is a mix of assembly and C files (or any other file combination), the most abstract programming language should be selected. For example, for a project with C and assembly files, a C LDF should be selected. Similarly, for a C++ and C project, the C++ LDF should be selected.

Press **Next**.

Step 2: Specifying System Information

Choose whether the project is for a single-processor system or a multiprocessor (MP) system.

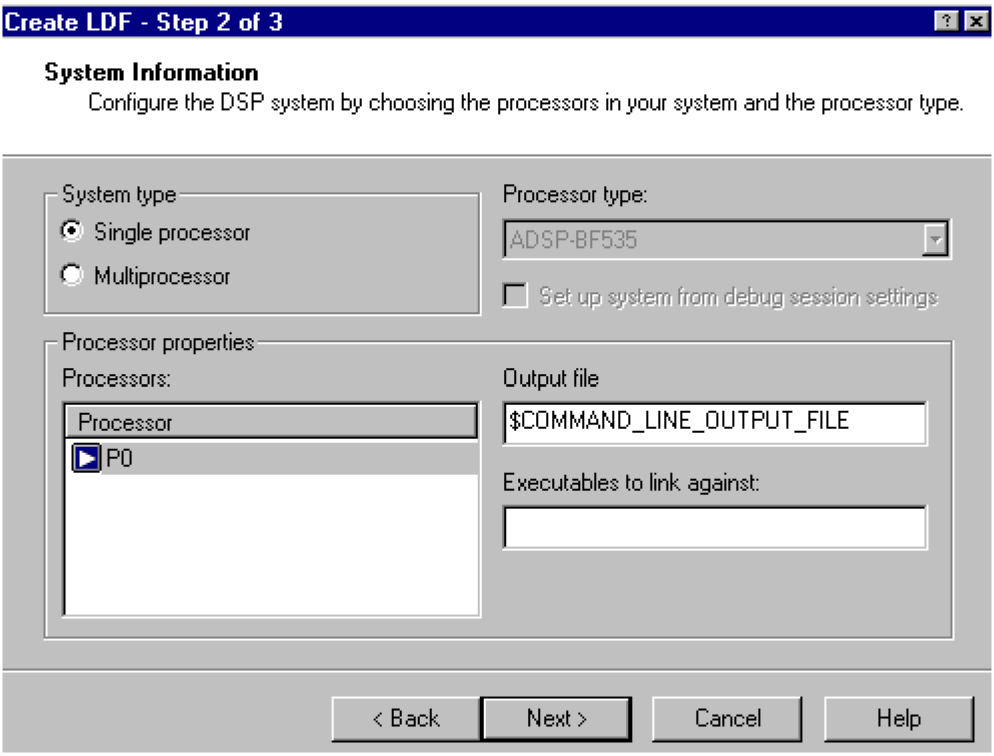


Figure 4-4. Selecting System and Processor Types

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

- For a single-processor system, the **Processors** list shows only one processor and the MP address columns do not appear.
- For a multiprocessor system, right-click in the **Processor Properties** box to add the desired number of processors included in the .LDF file, name each processor, and set the processor order (which will determine each processor's MP memory address range).

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

By selecting **Set up system from debug session settings**, the processor information (number of processors and the processor names) is filled automatically from the current settings in the debug session. This field is grayed out when the current debug session is not supported by the Expert Linker.

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

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

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

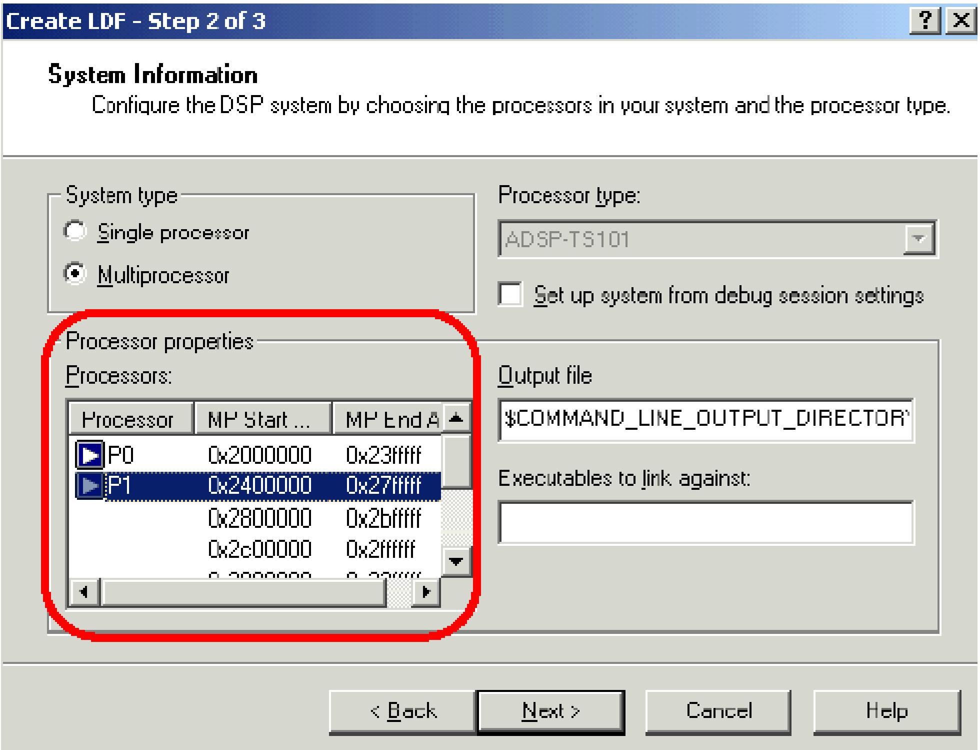


Figure 4-5. Processors and MMS Offset

The MP address range is available only for processors that have MP memory space..

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

Step 3: Completing the LDF Wizard

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

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

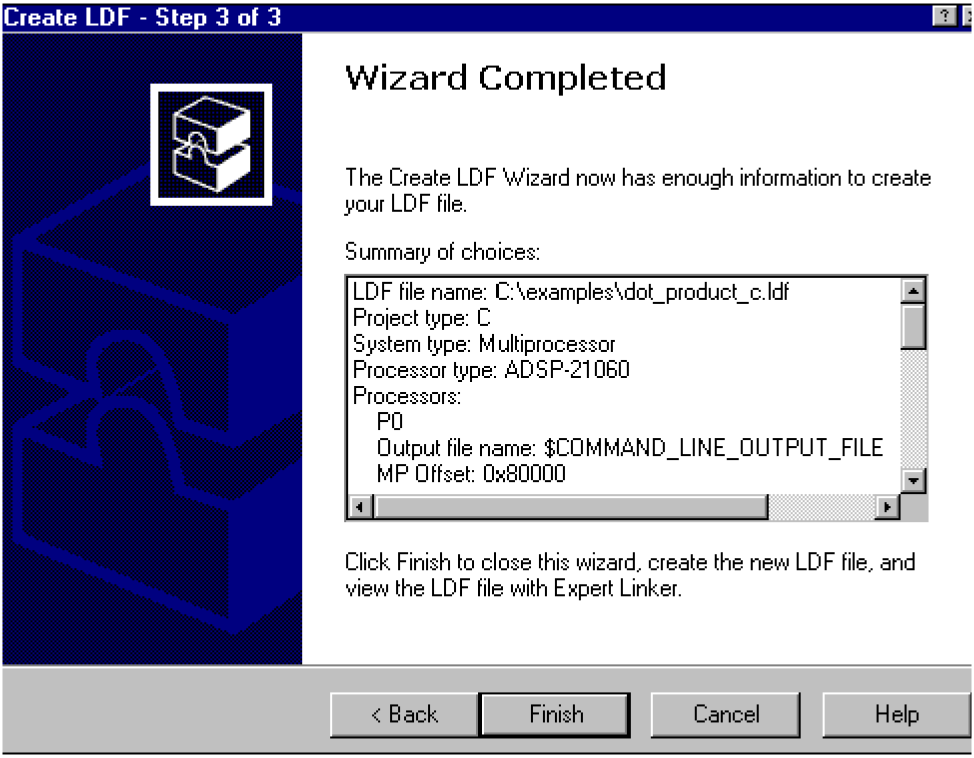


Figure 4-6. Wizard Completed Page of the Create LDF Wizard

Expert Linker Window Overview

The Expert Linker window contains two panes:

- The **Input Sections** pane (Figure 4-7) provides a tree display of the project's input sections (see "Input Sections Pane" on page 4-12).
- The **Memory Map** pane displays each memory map in a tree or graphical representation (see "Memory Map Pane" on page 4-19).

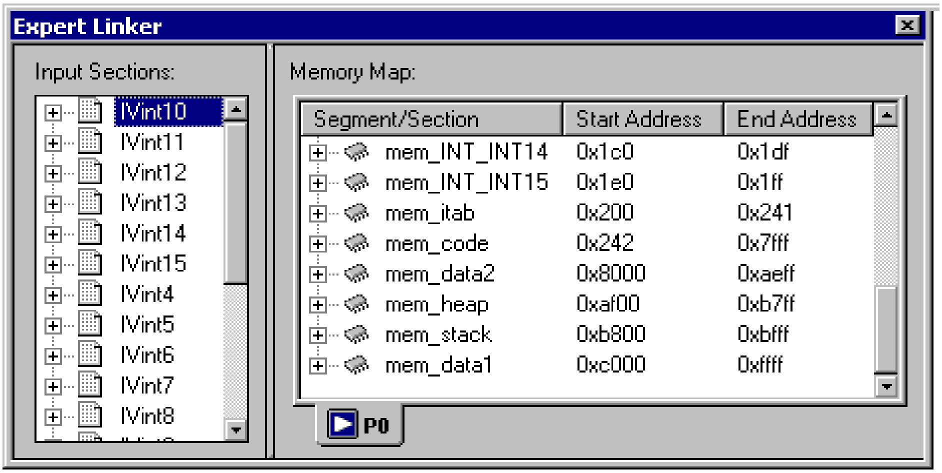


Figure 4-7. Expert Linker Window

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

Using drag-and-drop, you can map an input section to an output section in the memory map. Each memory segment may have one or more output sections under it. Input sections that have been mapped to an output section are displayed under that output section. For more information, refer to “Input Sections Pane” on page 4-12 and “Memory Map Pane” on page 4-19.

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

Input Sections Pane

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

Input Sections Menu

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

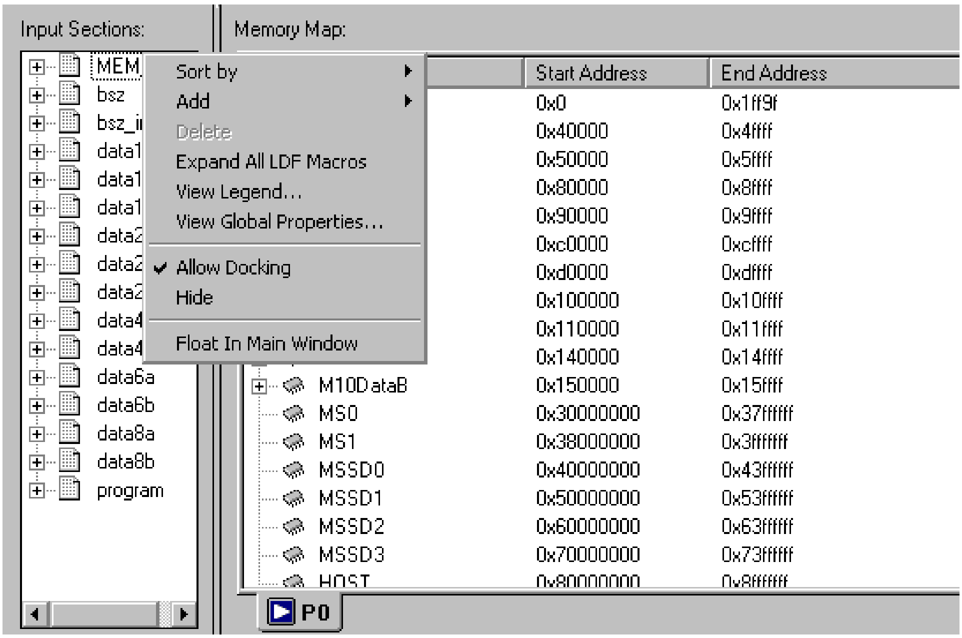


Figure 4-8. Input Sections Right-Click Menu

The main menu functions include:

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

  Create an input section as a shell, without object/library files or LDF macros in it. You can even map this section to an output section. However, input sections without data are grayed out.
- **Delete** – Deletes the selected object (input section, object/library file, or LDF macro).
- **Remove** – Removes an LDF macro from another LDF macro but does not delete the input section mappings that contain the removed macro. The difference between **Delete** and **Remove** is that **Delete** completely deletes the input section macros that contain the deleted macro.

  NOTE: The **Remove** option becomes available only if you right-click on an LDF macro that is part of another LDF macro.
- **Expand All LDF Macros** – Expands all the LDF macros in the input sections pane so that the contents of all the LDF macros are visible.
- **View Legend** – Displays the **Legend** dialog box which shows icons and colors used by the Expert Linker.

- **View Section Contents** – Opens the **Section Contents** dialog box, which displays the section contents of the object file, library file, or `.DXE` file. This command is available only after you link or build the project and then right-click on an object or output section.
- **View Global Properties** – Displays the **Global Properties** dialog box which provides the map file name (of the map file generated after linking the project) as well as access to various processor and setup information (see Figure 4-43 on page 4-52).

Mapping an Input Section to an Output Section

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

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

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

A red "**x**" on an icon indicates the object/file is not mapped. Once an input section has been completely mapped (that is, all object files that contain the section are mapped), the icon next to the input section changes to indicate that it is now mapped; the "**x**" disappears. See Figure 4-9.

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

Viewing Icons and Colors

Use the **Legend** dialog box to display all possible icons in the tree pane as well as short descriptions of each icon. (Figure 4-9)

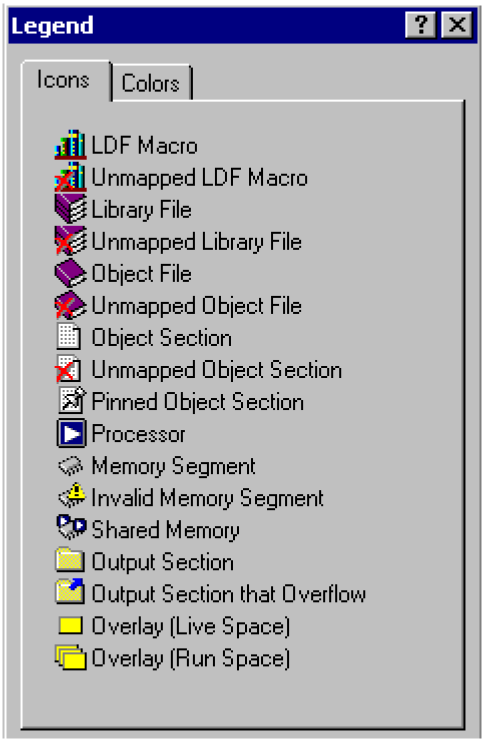


Figure 4-9. Legend Dialog Box – Icons Page

The red "**x**" on an icon indicates this object/file is not mapped.

Click the **Colors** tab to view the **Colors** page (Figure 4-10). This page contains a list of colors used in the graphical memory map view; each item's color can be customized. The list of displayed objects depends on the processor family.

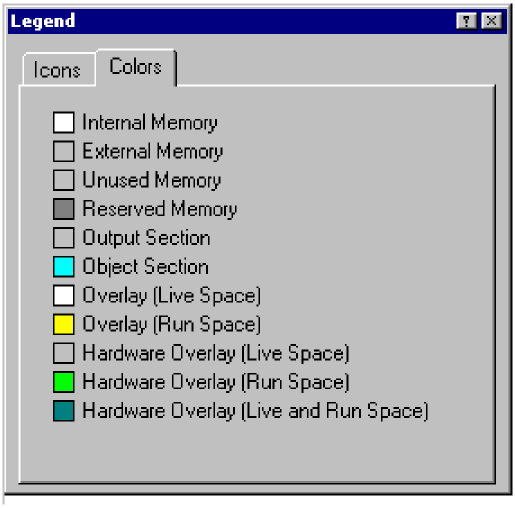


Figure 4-10. Legend Dialog Box – Colors Page

To change a color:

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

   The system displays the **Select a Color** dialog box (Figure 4-11).

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

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

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

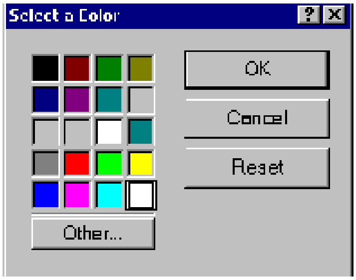


Figure 4-11. Select a Color Dialog Box

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

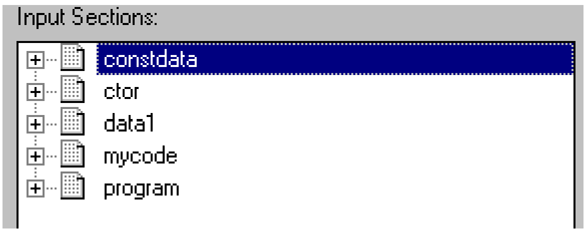


Figure 4-12. Expert Linker Window – Sorted by Input Sections

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

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

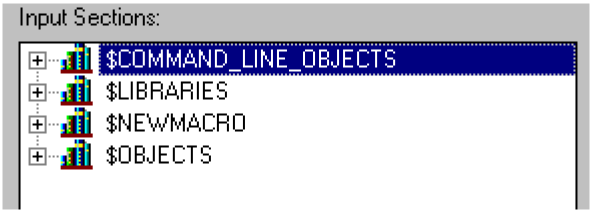


Figure 4-13. Expert Linker Window – Sorted by LDF Macros

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 specifies start and end addresses, memory width, and memory type (such as program, data, stack, and so on). It connects your program to the target system. The OUTPUT() command directs the linker to produce an executable (.DXE) file and specifies its file name. Figure 4-14 shows a typical memory map pane.

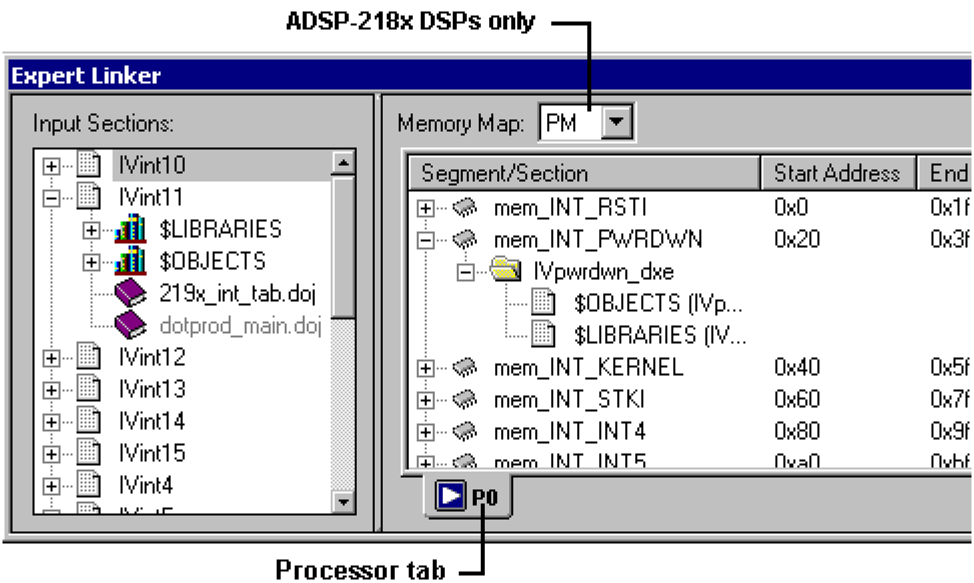


Figure 4-14. Expert Linker Window – Memory Map

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

This section describes:

- "Context Menu" on page 4-22
- "Tree View Memory Map Representation" on page 4-24
- "Graphical View Memory Map Representation" on page 4-25
- "Specifying Pre- and Post-Link Memory Map View" on page 4-31
- "Zooming In and Out on the Memory Map" on page 4-32
- "Adding a Memory Segment" on page 4-33
- "Inserting a Gap Into a Memory Segment" on page 4-35
- "Working With Overlays" on page 4-36
- "Viewing Section Contents" on page 4-38
- "Profiling Object Sections" on page 4-42
- "Adding Shared Memory Segments and Linking Object Files" on page 4-46

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

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

The **Memory Map** pane displays a tooltip when the mouse cursor moves 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.

Use the right-click menu ("Context Menu" on page 4-22) to select and perform major memory map functions.

Invalid Memory Segment Notification:
When a memory segment is invalid (for example, when a memory range overlaps another memory segment or if the memory width is invalid), the tree shows an **Invalid Memory Segment** icon (see Figure 4-15). Move the mouse pointer over the icon and a tooltip displays a message describing why the segment is invalid.

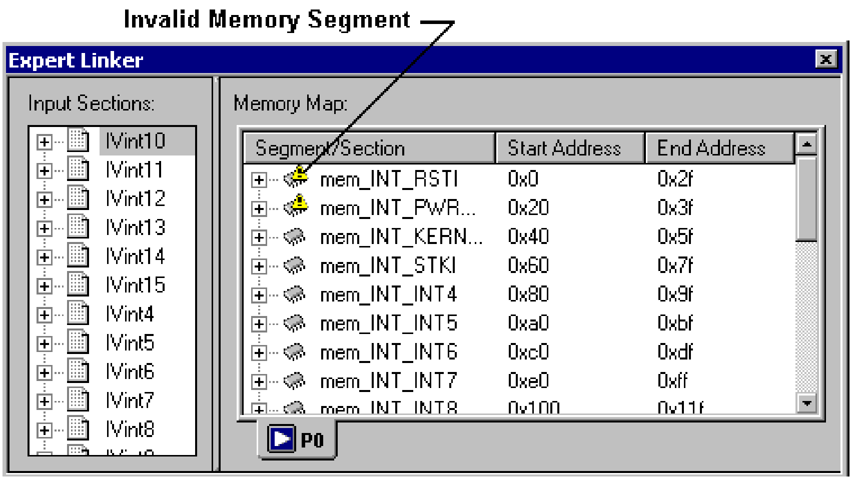


Figure 4-15. Memory Map With Invalid Memory Segments

Context Menu

Display the context menu by right-clicking in the **Memory Map** pane. The menu ([Figure 4-16](#)) allows you to select and perform major functions. The available right-click menu commands are listed below.

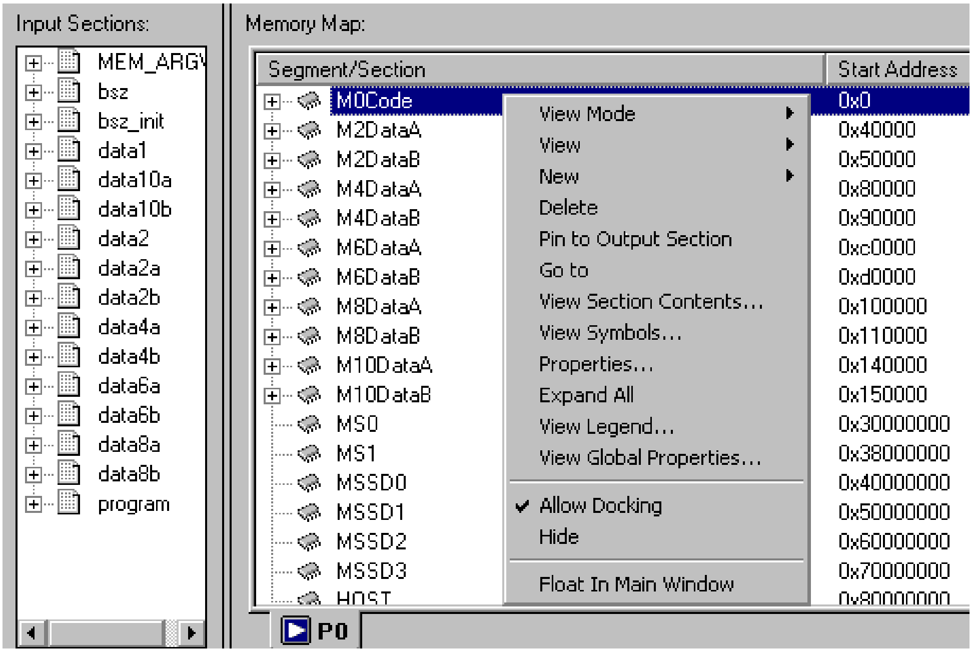


Figure 4-16. Memory Map Main Menu

View Mode

- **Memory Map Tree** – Displays the memory map in a tree representation (see Figure 4-17 on page 4-25)
- **Graphical Memory Map** – Displays the memory map in graphical blocks (see Figure 4-18 on page 4-26)

View

- **Mapping Strategy (Pre-Link)** – Displays the memory map that shows the placement of your object sections.
- **Link Results (Post-Link)** – Displays the memory map that shows the actual placement of the object sections.

New

- **Memory Segment** – Specifies the name, address range, type, size, and so on for memory segments you want to add.
- **Output Section** – Adds an output section to the selected memory segment. (Right-click on the memory segment to access this command.) If you do not right-click on a memory segment, this option is disabled.
- **Shared Memory** – Adds a shared memory to the memory map.
- **Overlay** – Invokes a dialog box that allows adding a new overlay to the selected output section or memory segment. The selected output section is the new overlay's run space (see Figure 4-54 on page 4-69).

Delete – Deletes the selected object.

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

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

View Section Contents – Invokes a dialog box that displays the contents of the input or output section. It is available only after you link or build the project and then right-click on an input or object section (see Figure 4-31 on page 4-39).

View Symbols – Invokes a dialog box that displays the symbols for the project, overlay, or input section. It is available only after you link the project and then right-click on a processor, overlay, or input section (see Figure 4-43 on page 4-52).

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

View Legend – Displays the **Legend** dialog box showing tree view icons and a short description for each icon. The **Colors** page lists the colors used in the graphical memory map. You can customize each object's color. See Figure 4-9 on page 4-15 and Figure 4-10 on page 4-16.

View Global Properties – Displays a **Global Properties** dialog box that lists the map file generated after linking the project. It also provides access to some processor and setup information (see Figure 4-44 on page 4-53).

Tree View Memory Map Representation

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

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

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

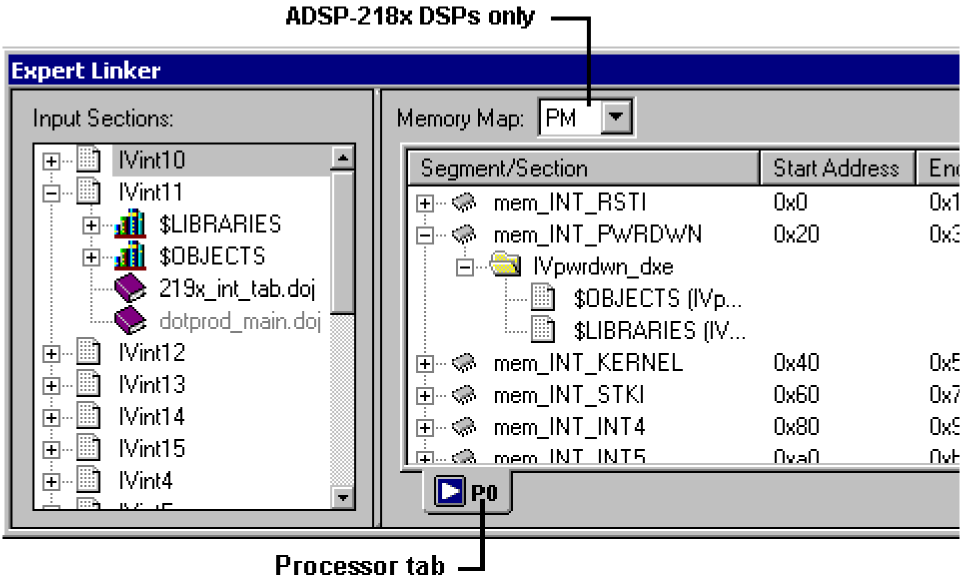


Figure 4-17. Expert Linker Window – Memory Map Tree View

Graphical View Memory Map Representation

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

View the memory map from two perspectives: pre-link view and post-link view (see "Specifying Pre- and Post-Link Memory Map View" on page 4-31). Figure 4-18 through Figure 4-22 show examples of graphical memory map representations.

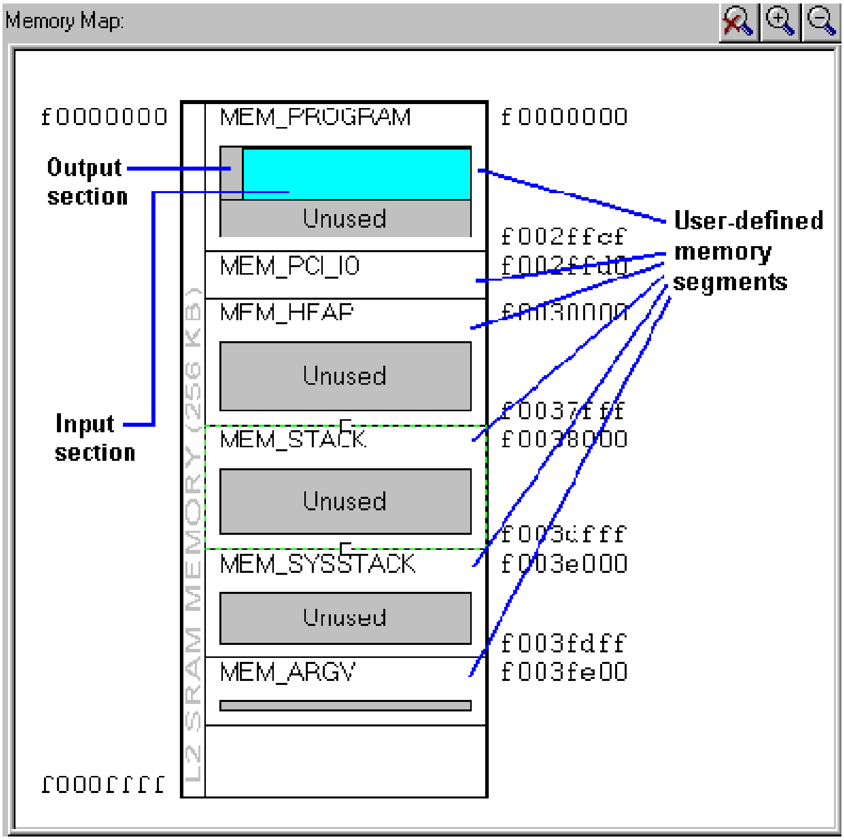


Figure 4-18. Graphical Memory Map Representation

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

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

Three buttons at the top right of the **Memory Map** pane permit zooming. If there is not enough room to display the memory map when zoomed in, horizontal and/or vertical scroll bars allow you to view the entire memory map (for more information, see "Zooming In and Out on the Memory Map" on page 4-32).

You can drag-and-drop any object except memory segments. See Figure 4-19.

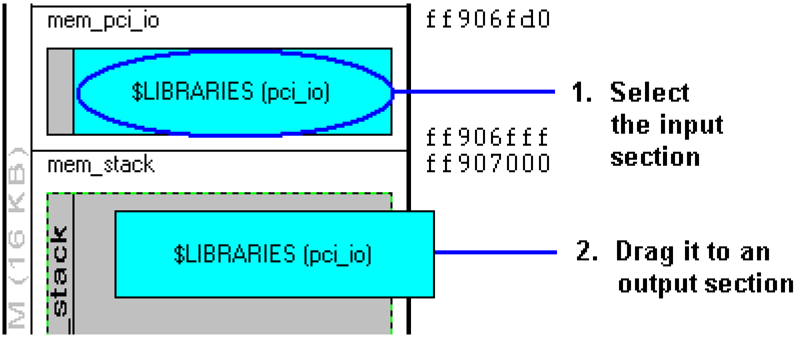


Figure 4-19. Dragging and Dropping an Object (1)

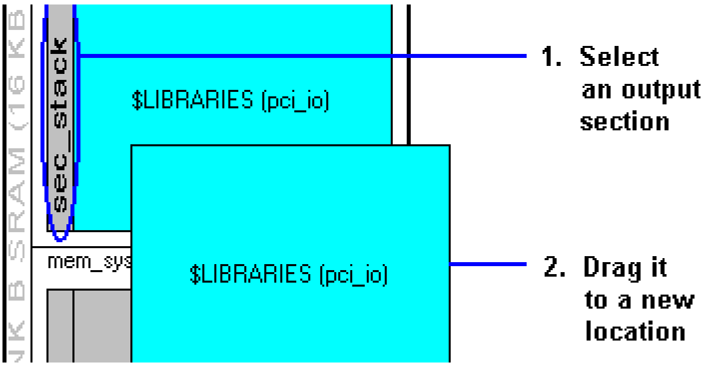


Figure 4-20. Dragging and Dropping an Object (2)

Select a memory segment to display its border. Memory segments, when selected, display a tiny box at their top and bottom borders (Figure 4-21). Drag the border (at this box) to change the memory segment's size. By doing this, the size of the selected and adjacent memory segments change.

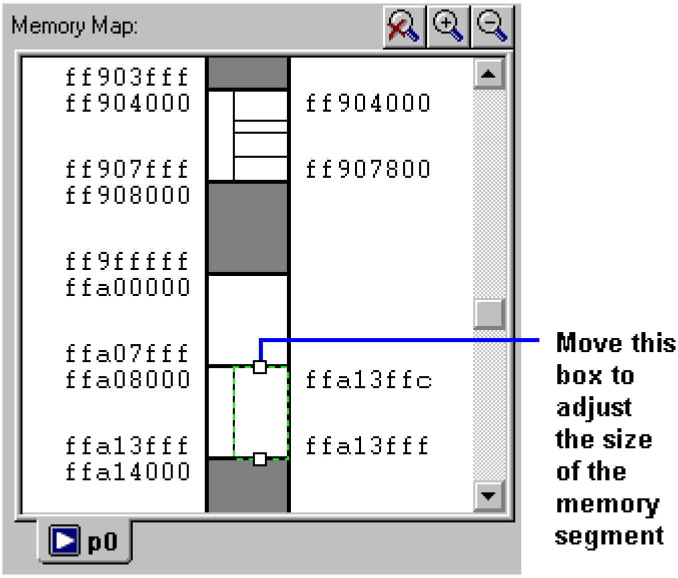
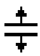


Figure 4-21. Adjusting the Size of a Memory Segment

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



When an object is selected in the memory map, it is highlighted as shown in Figure 4-22 on page 4-30. If you move the mouse pointer over an object in the graphical memory map, a yellow tooltip displays the information about the object (such as name, address, and size).

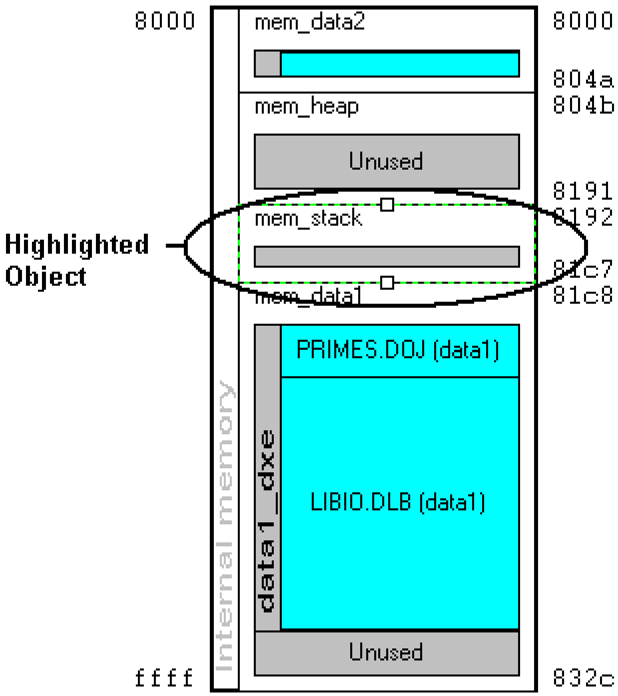


Figure 4-22. A Highlighted Memory Segment in the Memory Map

Specifying Pre- and Post-Link Memory Map View

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

- To enable pre-link view from the **Memory Map** pane, right-click and choose **View** and **Mapping Strategy** (**Pre-Link**). Figure 4-23 on page 4-31 illustrates a memory map before linking.

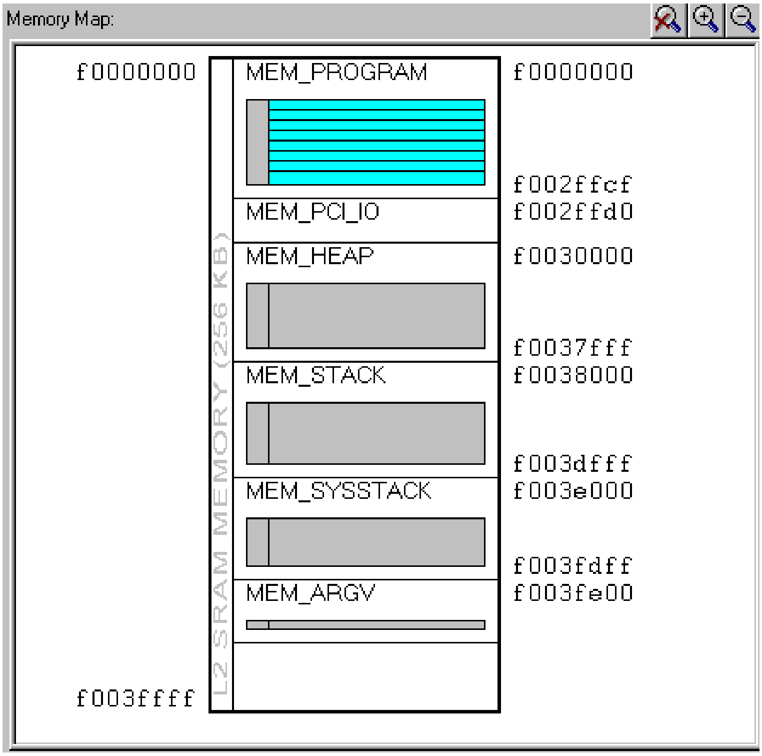


Figure 4-23. Memory Map Pane in Pre-Link View

- To enable post-link view from the **Memory Map** pane, right-click and choose **View** and **Link Results** (**Post-Link**). Figure 4-24 on page 4-32 illustrates a memory map after linking.

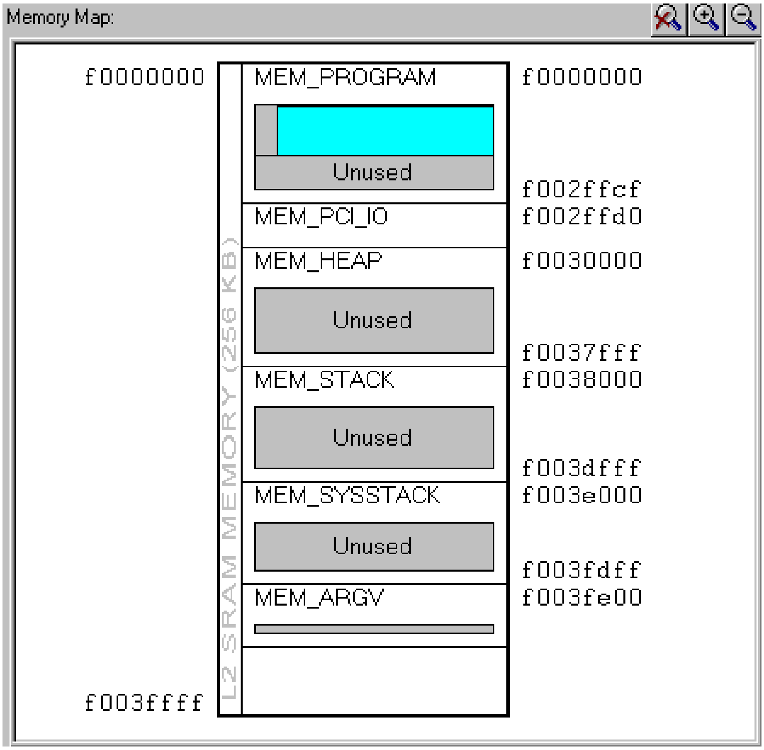


Figure 4-24. Memory Map Pane in Post-Link View

Zooming In and Out on the Memory Map

From the **Memory Map** pane, you can zoom in or out incrementally or zoom in or out completely. Three buttons at the top right of the pane perform zooming operations. Horizontal and/or vertical scroll bars appear when there is not enough room to display a zoomed memory map in the **Memory Map** pane (see Figure 4-25 on page 4-33).

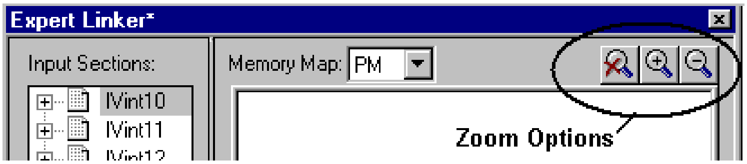


Figure 4-25. Memory Map – Zoom Options

To:

- Zoom in, click on the magnifying glass icon with the + sign above the upper right corner of the memory map window.
- Zoom out, click on the magnifying glass icon with the - sign above the upper right corner of the memory map window.
- Exit zoom mode, click on the magnifying glass icon with the "**x**" above the upper right corner of the memory map window.
- View a memory object by itself by double-clicking on the memory object.
- View the memory object containing the current memory object by double-clicking on the white space around the memory object

Adding a Memory Segment

You can add memory segments to the memory map. This procedure assumes that the Expert Linker window (**Memory Map** pane) is open.

To add a memory segment:

1. Right-click in the **Memory Map** pane.
2. Choose **New** and then choose **Memory Segment**. The **Memory Segment Properties** dialog box appears (Figure 4-26).
3. In **Name**, type a name for the memory segment.
4. Specify the following attributes:

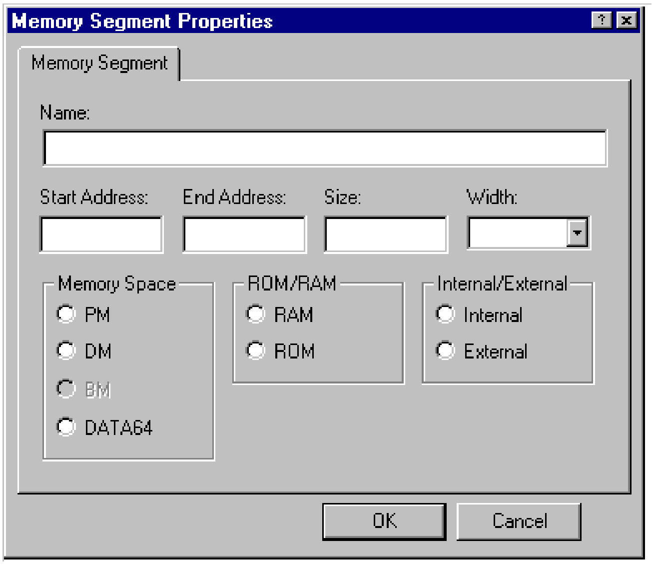


Figure 4-26. Memory Segment Properties Dialog Box

- Start address
- End Address
- Size (hexadecimal)

It is only necessary to specifiy either "End Address" or "Length" and not both.

- Width
- Memory Space
  For Blackfin and TigerSHARC processors, this option is unavailable, because VisualDSP employs a unified memory space.
- ROM/RAM
- Internal/External (memory location)

5. Click **OK**.

Inserting a Gap Into a Memory Segment

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

To insert a gap:

1. Right-click on a memory segment.

2. Choose **Insert gap**. The **Insert Gap** dialog box appears, as shown in Figure 4-27. It displays the start address, end address, and size of the selected memory segment.

Figure 4-27. Insert Gap Dialog Box

You may insert a gap at the start of the memory segment or the end of it.

- If the **Start...** is chosen, the **Start address** for the gap is grayed out and you must enter an **End Address** or **Size** (of the gap).
- If the **End...** is chosen, the **End address** of the gap is grayed out and you must enter a **Start Address** or **Size**.

Working With Overlays

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

Figure 4-28 shows an overlay in "live" space, and Figure 4-29 shows an overlay in "run" space.

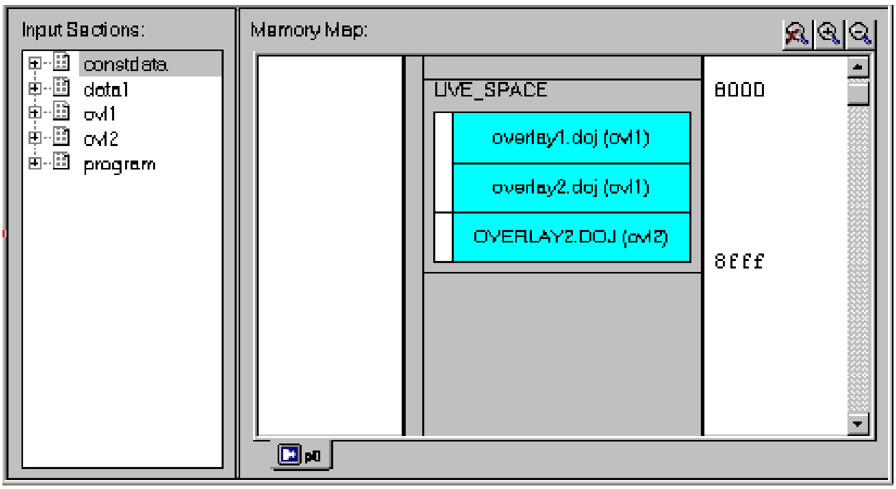


Figure 4-28. Graphical Memory Map Showing an Overlay in "Live" Space

Overlays in a "run" space appear one at a time in the graphical memory map. The scroll bar next to an overlay in "run" space allows you to specify an overlay to be shown on top. Drag the overlay on top to another output section to change the "run" space for an overlay.

Click the Up arrow or Down arrow button in the header to display a previous overlay or next overlay in "run" space. Click the browse button to display the list of all available overlays. The header shows the number of overlays in this "run" space as well as the current overlay number.

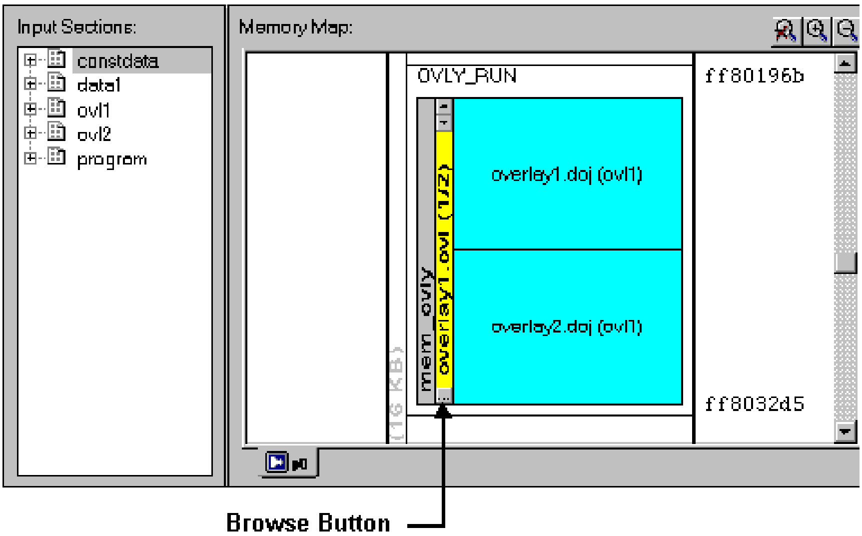


Figure 4-29. Graphical Memory Map Showing an Overlay "Run" Space

To create an overlay in the "run" space:

1. Right-click on an output section.
2. Choose **New -> Overlay**.
3. Select the "live" space from the **Overlay Properties** dialog box (see "Managing Overlay Properties" on page 4-69). The new overlay appears in the "run" and "live" spaces in two different colors in the memory map.

Viewing Section Contents

To view the contents of an input section or an output section, specify the particular memory address and the display's format.

This capability employs the `elfdump` utility (`elfdump.exe`) to obtain the section contents and display it in a window similar to a memory window in VisualDSP++. Multiple **Section Contents** dialog boxes may be displayed. For example, Figure 4-30 shows output section contents in HEX format.

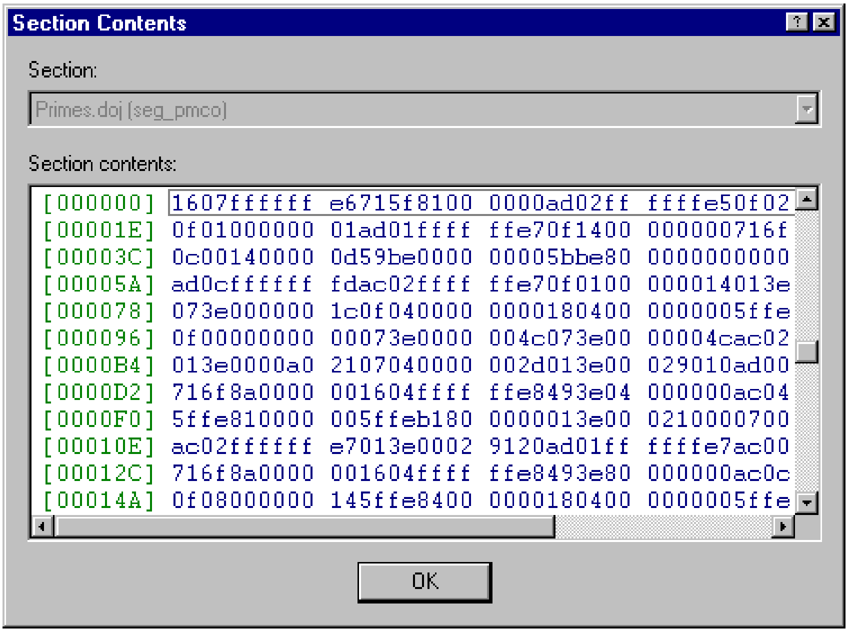


Figure 4-30. Output Section Contents in Hex Format

To display the contents of an output section:

1. In the **Memory Map** pane, right-click an output section.

2. Choose **View Section Contents** from the menu.
   The **Section Contents** dialog box appears.

   By default, the memory section content appears in **Hex** format.

3. Right-click anywhere in the section view to display a menu with these selections:

   - **Go To** – Displays an address in the window.

     - **Select Format** — Provides a list of formats: **Hex**, **Hex and ASCII**, and **Hex and Assembly.** Select a format type to specify the memory format.

Figure 4-31 and Figure 4-32 illustrate memory data formats available for the selected output section.

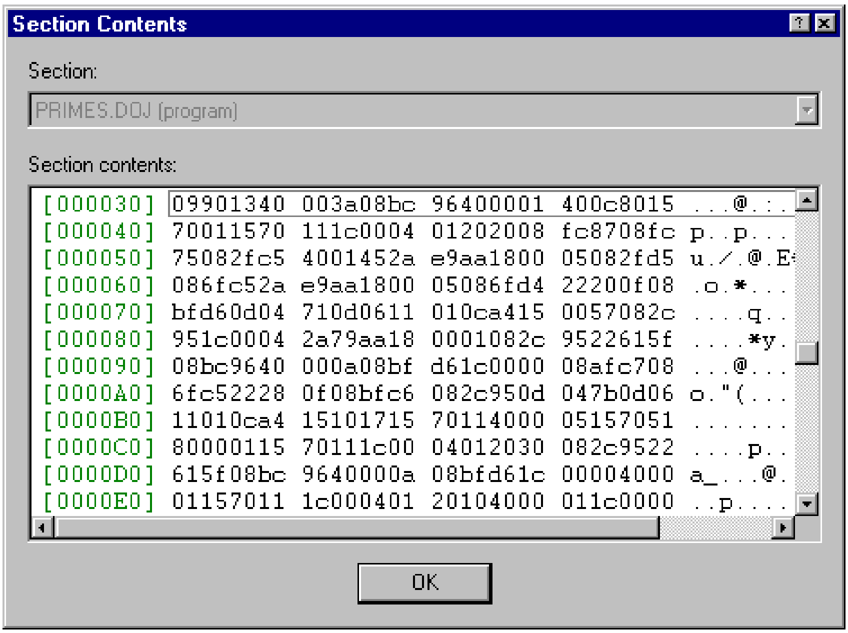


Figure 4-31. Output Section Contents in Hex and ASCII Format

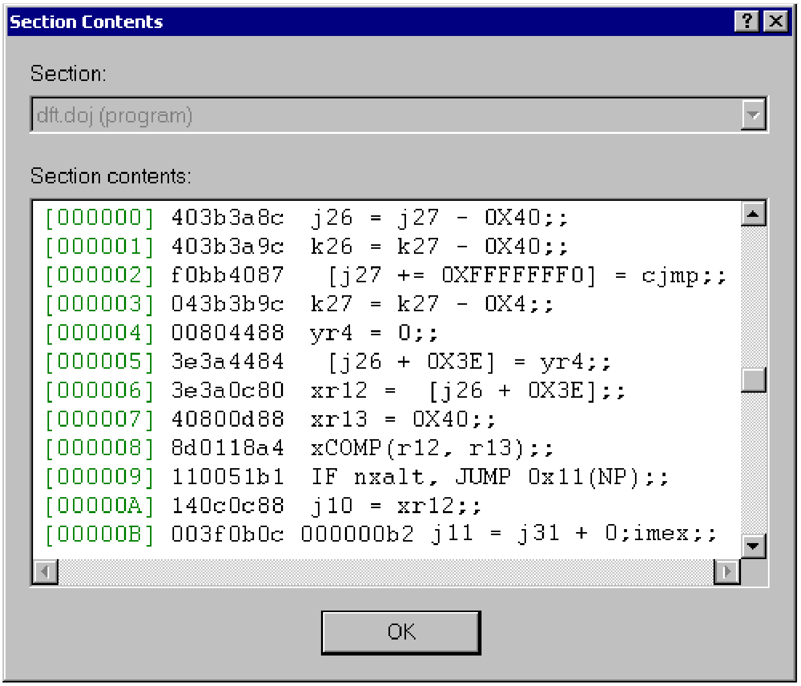


Figure 4-32. Output Section Contents in Hex and Assembly Format

Viewing Symbols

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

To view symbols (Figure 4-33):

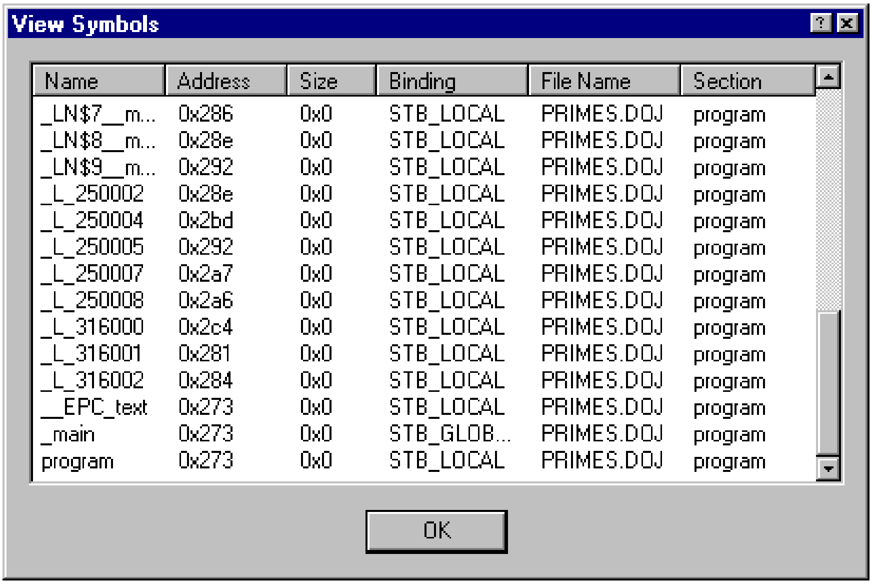


Figure 4-33. View Symbols Dialog Box

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

2. Right-click and choose **View Symbols**.

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

Profiling Object Sections

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

The following is a sample profiling procedure.

Start by selecting **Profile execution of object sections** in the **General** page of the **Global Properties** dialog box (Figure 4-34).

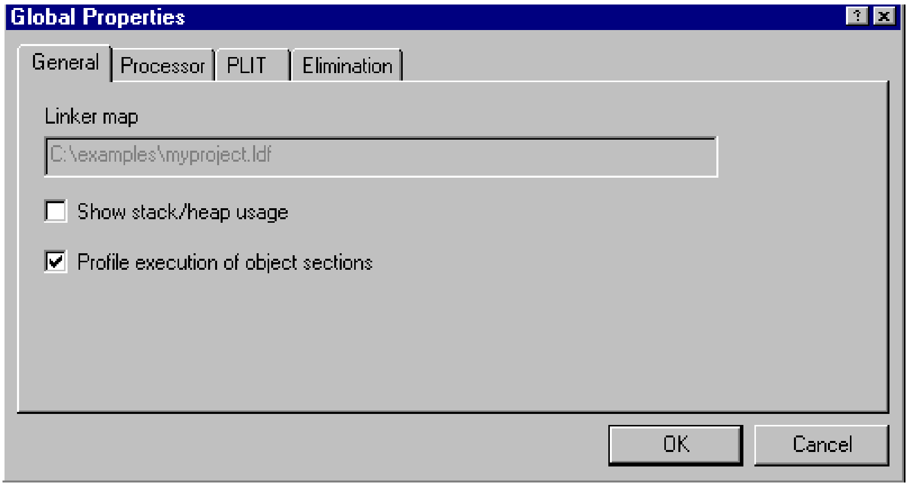


Figure 4-34. General Page of the Global Properties Dialog Box

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

When the program run is complete, Expert Linker colors each object section with a different shade of red to indicate how much time was spent executing that section. For an example, see Figure 4-35.

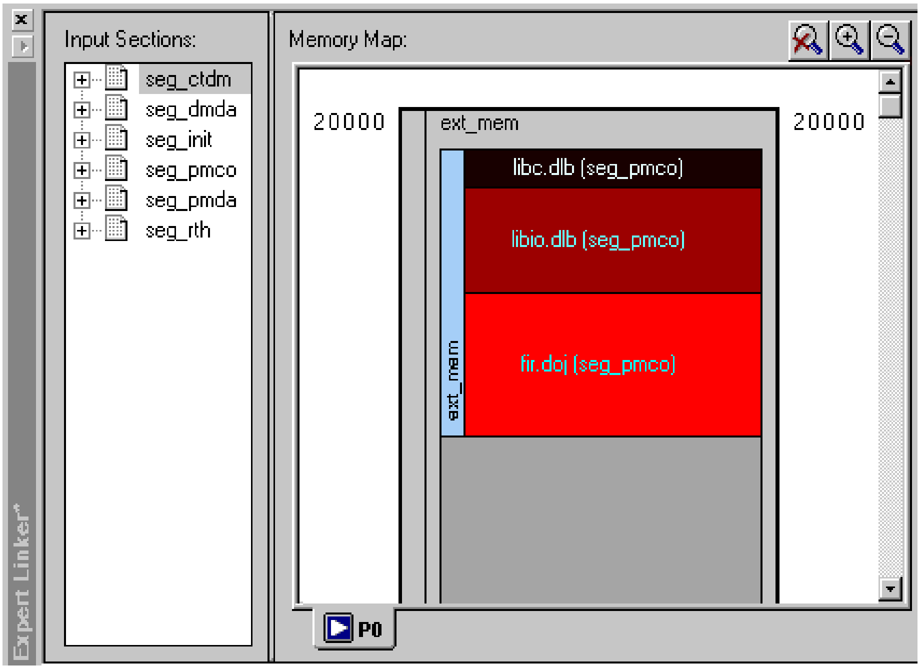


Figure 4-35. Colored Object Sections

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

From Expert Linker, you can view PC sample counts for object sections. To view an actual PC sample count (Figure 4-36), move the mouse pointer over an object section and view the PC sample count.

To view sample counts for functions located within an object section, double-click on the object section (Figure 4-37).

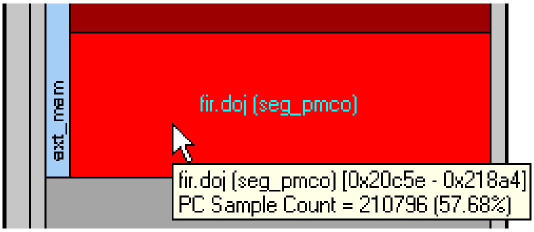


Figure 4-36. PC Sample Count

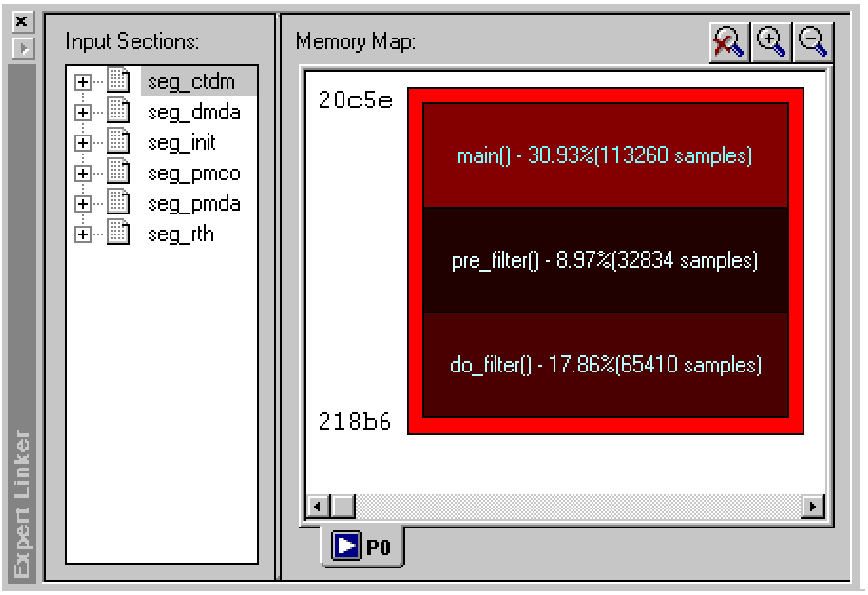


Figure 4-37. Sample Count of Functions Within Object Section

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

You can view detailed profile information such as the sample counts for each line in the function (Figure 4-38). To view detailed profile information, double-click on a function.

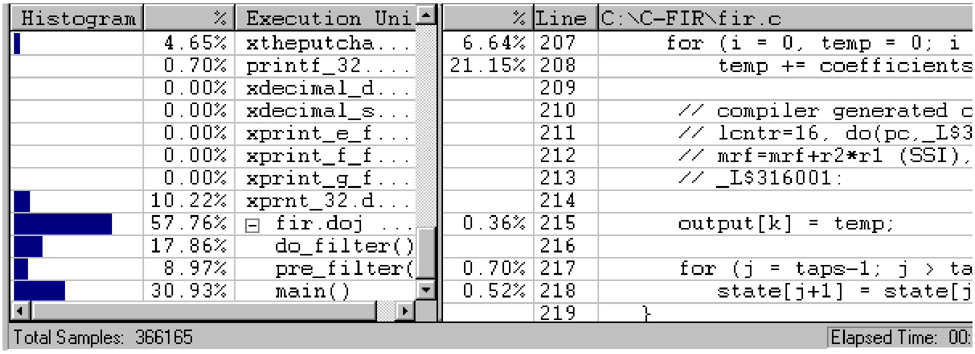


Figure 4-38. Detailed Profile Information

To view PC samples as a percentage of total samples, view the memory map tree (Figure 4-39).

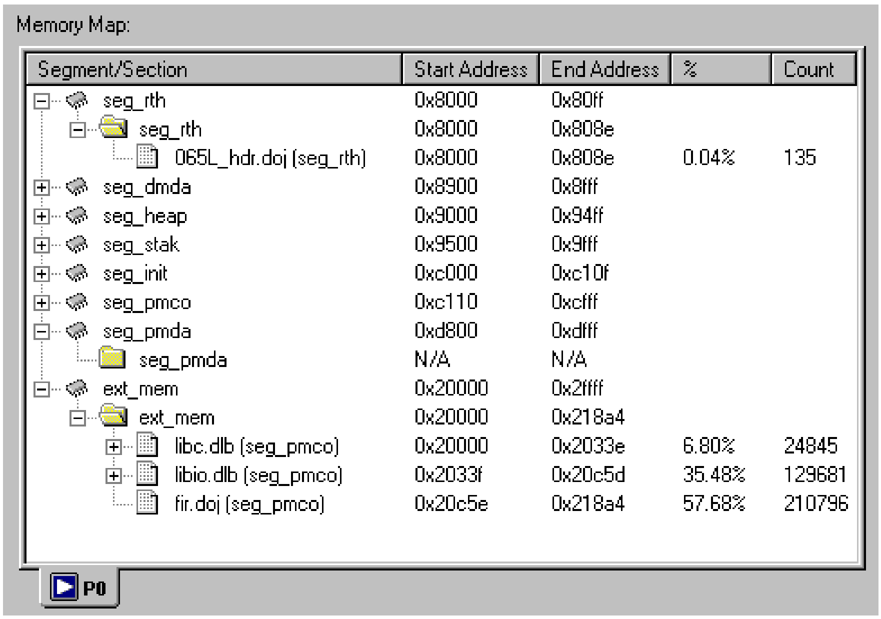


Figure 4-39. Percentage of Total PC Sample Count

Adding Shared Memory Segments and Linking Object Files

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

Refer to Engineer-To- Engineer Note EE-202 "*Using the Expert Linker for Multiprocessor LDF*" for a detailed description and procedure. Find this EE Note on Analog Devices Web site at `http://www.analog.com/ee-notes`.

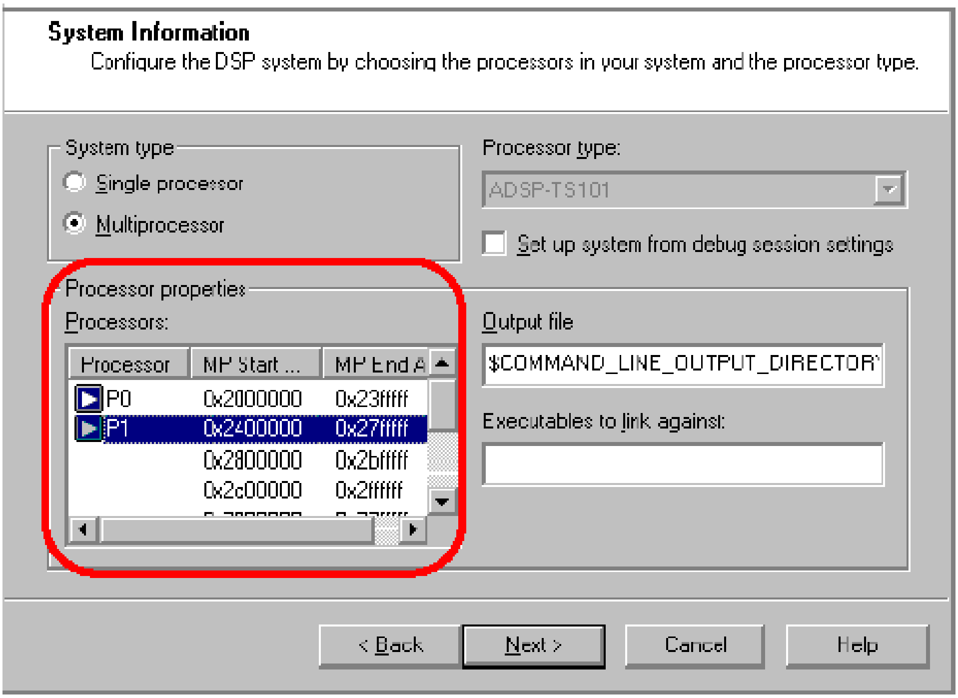


Figure 4-40. Multiprocessor LDF Selection

To add a shared memory section to the .LDF file, right-click in the **Memory Map** pane and select **New/Shared Memory**. Then specify a name for the shared memory segment (.SM) and select the processors that have access to this shared memory segment. Refer to "Managing Shared Memory Properties" on page 4-74 for more information.

As shown in Figure 4-41, a new shared memory segment, visible to processors P0 and P1, has been successfully added to the system. Note that variables declared in the shared memory segment will be accessed by both processors in the system. In order for the linker to be able to correctly resolve these variables, the link against command should be used once again.

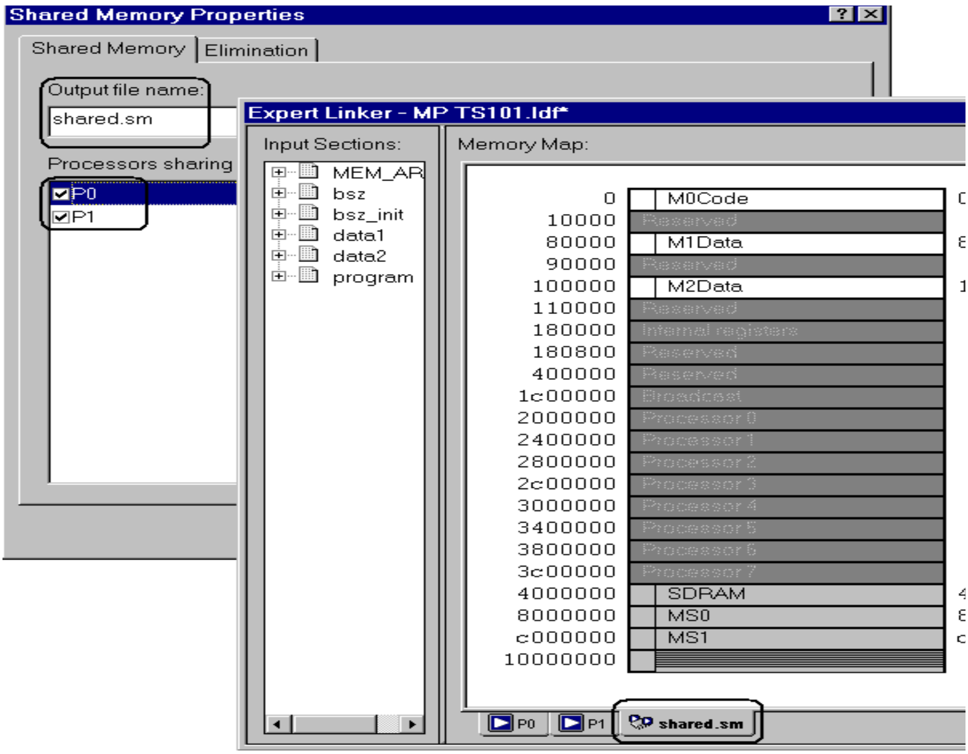


Figure 4-41. Shared Memory Segment

Expert Linker automatically adds shared memory segments, and therefore no any additional modifications to the LDF are needed.

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

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

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

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

In general, the linking process consists of these steps:

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

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

3. Add the object (.DOJ) files that correspond to each processor as well as to the shared memory segment.
   To do this, right-click on each recently created LDF macro and then select **Add/Object/Library File**. The use of LDF macros becomes extremely useful in systems where there is more than one object files, .DOJ files per processor or shared memory segments, in which case the same step previously explained should be followed for each .DOJ file.

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

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

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

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

8. Press **Rebuild All**.

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

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

The LDF is now complete. Figure 4-42 illustrates the generated `.LDF` file in the **Source Code View** mode.

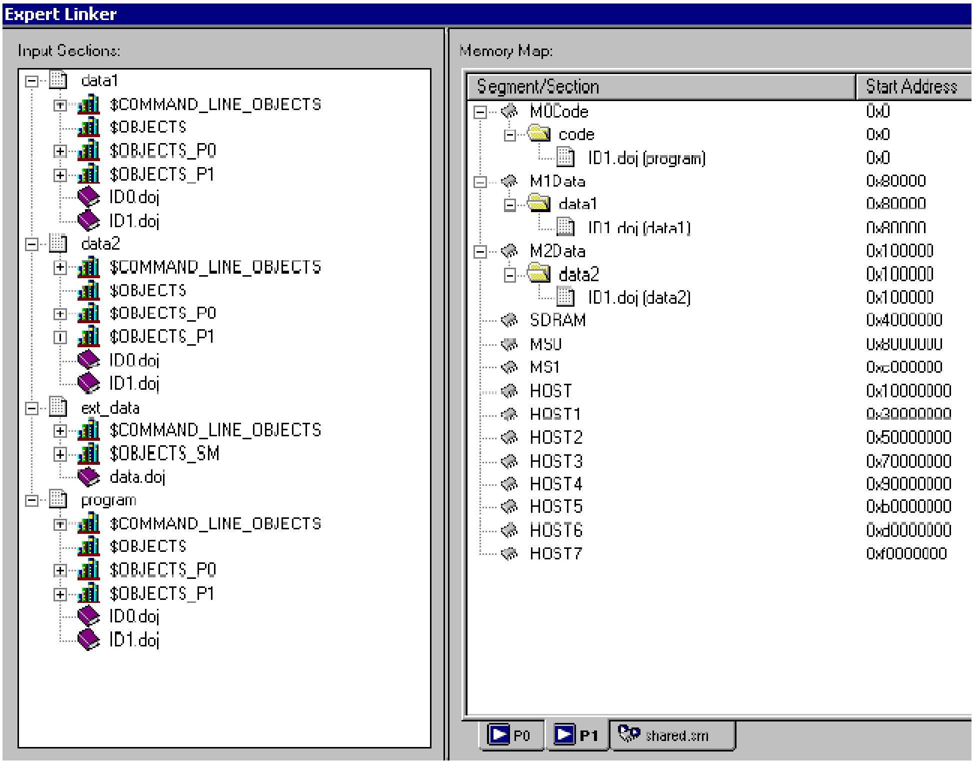


Figure 4-42. Expert Linker Multiprocessor LDF

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

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

Managing Object Properties

You can display different properties for each type of object. Since different objects may share certain properties, their **Properties** dialog boxes share pages.

The following procedures assume the **Expert Linker** window is open.

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

- "Managing General Global Properties" on page 4-52
- "Managing Processor Properties" on page 4-53
- "Managing PLIT Properties for Overlays" on page 4-55
- "Managing Elimination Properties" on page 4-56
- "Managing Symbols Properties" on page 4-58
- "Managing Memory Segment Properties" on page 4-61
- "Managing Output Section Properties" on page 4-62
- "Managing Packing Properties" on page 4-65
- "Managing Alignment and Fill Properties" on page 4-67
- "Managing Overlay Properties" on page 4-69
- "Managing Stack and Heap in Processor Memory" on page 4-71
- "Managing Shared Memory Properties" on page 4-74

Managing General Global Properties

To access **Global Properties**, right-click in the **Input Sections** pane and choose **Properties**.

The **Global Properties** dialog box appears.

The **General** tab of the **Global Properties** dialog box provides these selections (Figure 4-43):

- **Linker map file** displays the map file generated after linking the project. This is a read-only field.
- If **Show stack/heap usage** is selected after you run a project, Expert Linker shows how much of the stack and heap were used.
- If **Profile execution of object sections** is selected, Expert Linker enables the profiling feature that allows you to see "hotspots" in object sections and to fine-tune the placement of object sections.

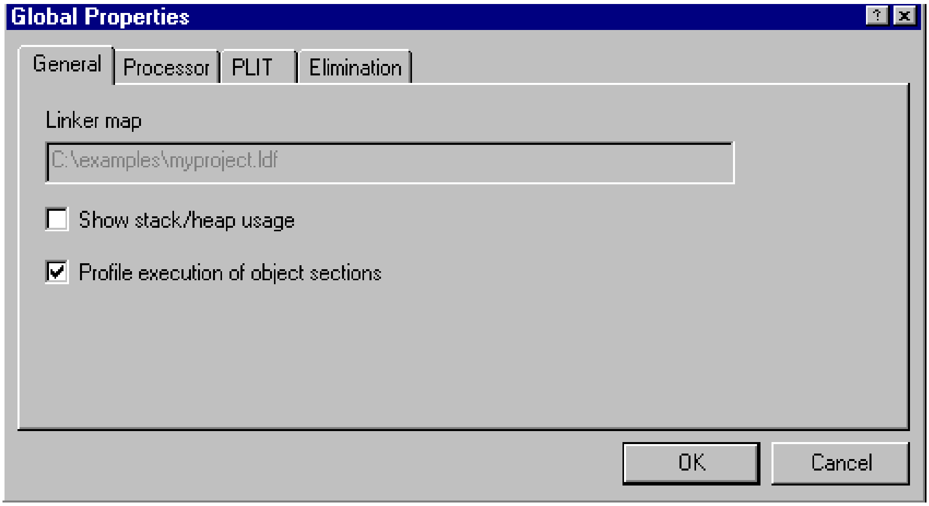


Figure 4-43. General Page of the Global Properties Dialog Box

Managing Processor Properties

To specify processor properties:

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

   The **Processor Properties** dialog box appears.

2. Click the **Processor** tab (Figure 4-44).

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

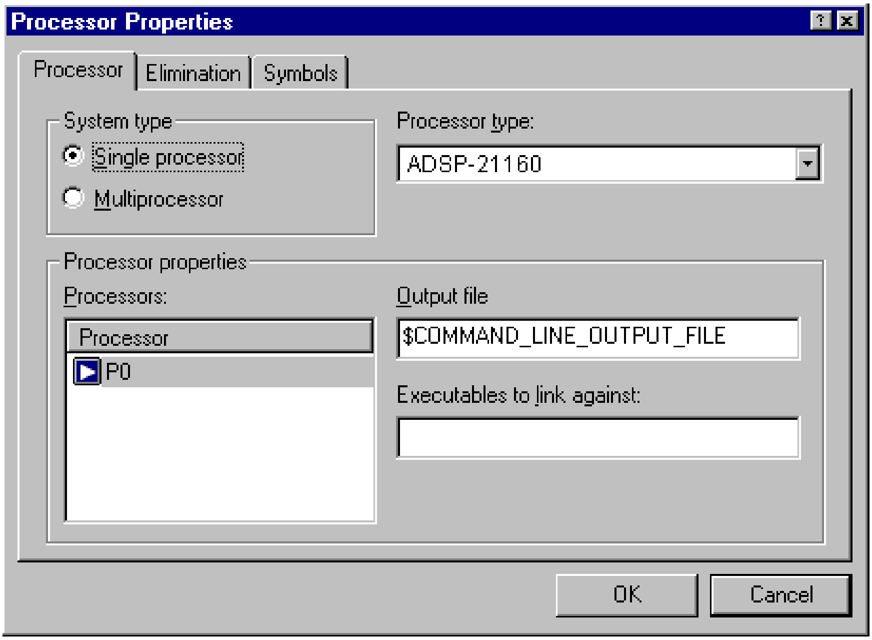


Figure 4-44. Processor Page of the Processor Properties Dialog Box

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

- Specify **System Type** – It may be a **Single processor** or **Multiprocessor** selection. (The **Processors list** displays the names of all the processors in the project and the address range for each processor.)
- Select a **Processor type** (such as `ADSP-21060`).
- Specify an **Output file** name – The file name may include a relative path and/or LDF macro. Specify an output file for each processor.
- Specify **Executables to link against** – Multiple files names are permitted, but must be separated with space characters or commas. Only `.SM`, `.DLB`, and `.DXE` files are permitted. A file name may include a relative path, LDF macro, or both.

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

For multiprocessor systems, you can add, delete, and rearrange processor order. Right-click in the **Processors** box, choose **Add Processor**, and type a name for the new processor, or choose **Delete Processor**. To move a processor, select the processor and drag it to another position in the **Processors** list.

When a processor in a multiprocessor system is moved to a different position, its address range changes. The **MP Start Addr.** and **MP End Addr.** information is static.

Managing PLIT Properties for Overlays

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

Enter assembly code only. Comments are not allowed.

To view and edit PLIT information:

1. Right-click in the **Input Sections** pane.
2. Choose **Properties**.
   The **Global Properties** dialog box appears.
3. Click the **PLIT** tab (Figure 4-45).

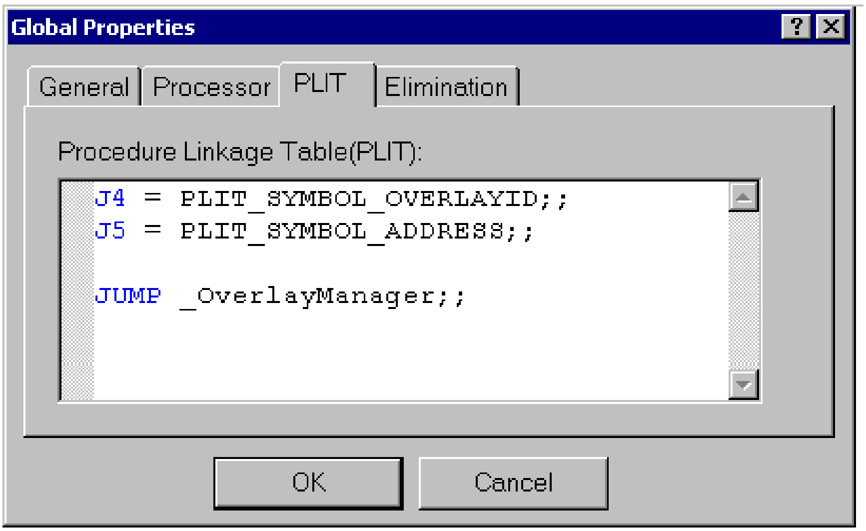


Figure 4-45. PLIT Page of the Global Properties Dialog Box

Managing Elimination Properties

Eliminate unused code from the target .DXE file. Specify the input sections from which to eliminate code and the symbols you want to keep.

Select the **Global Properties** dialog box by right-clicking in the **Input Sections** pane and choosing **Properties**.

Use the **Elimination** tab to perform elimination (Figure 4-46).

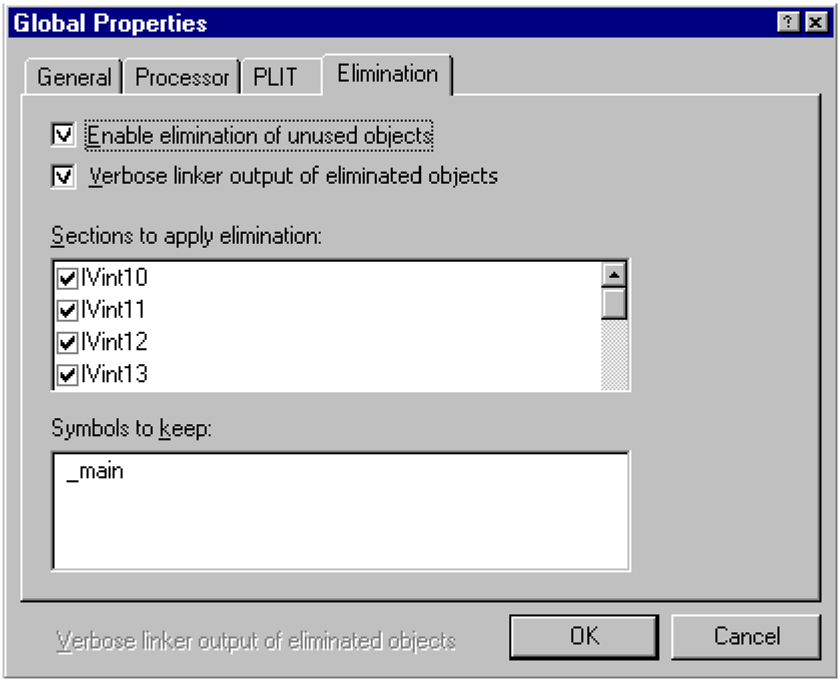


Figure 4-46. Elimination Tab

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

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

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

The **Symbols to keep** box displays a list of symbols to be retained (see "Managing Symbols Properties" on page 4-58 for more information).

Managing Symbols Properties

You can view the list of symbols resolved by the linker. You can also add and remove symbols from the list of symbols kept by the linker. The symbols can be resolved to an absolute address or to a program (.DXE) file. It is assumed that the elimination of unused code is enabled.

To add or remove a symbol:

1. Right-click in the **Input Sections** pane.
2. Choose **Properties**. The **Global Properties** dialog box appears.
3. Click the **Elimination** tab to add or remove a symbol (Figure 4-47).

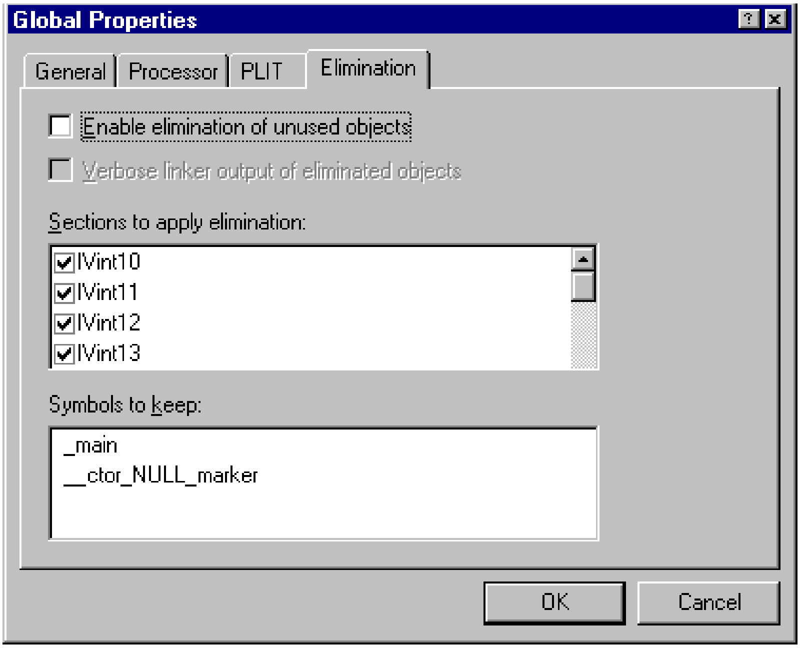


Figure 4-47. Elimination Page of the Global Properties Dialog Box

4. Right-click in the **Symbols to keep** box.

   Using the menu, choose **Add Symbol** to open the dialog box and type a new symbol name (names) at the end of the existing list. To delete a symbol, select the symbol, right-click, and choose **Remove Symbol**.

To specify symbol resolution:

1. In the **Memory Map** pane, right-click a **Processor** tab.

2. Choose **Properties**.
   The **Processor** page of the **Processor Properties** dialog box appears. The **Symbols** tab allows you to specify how symbols are to be resolved by the linker (Figure 4-48).

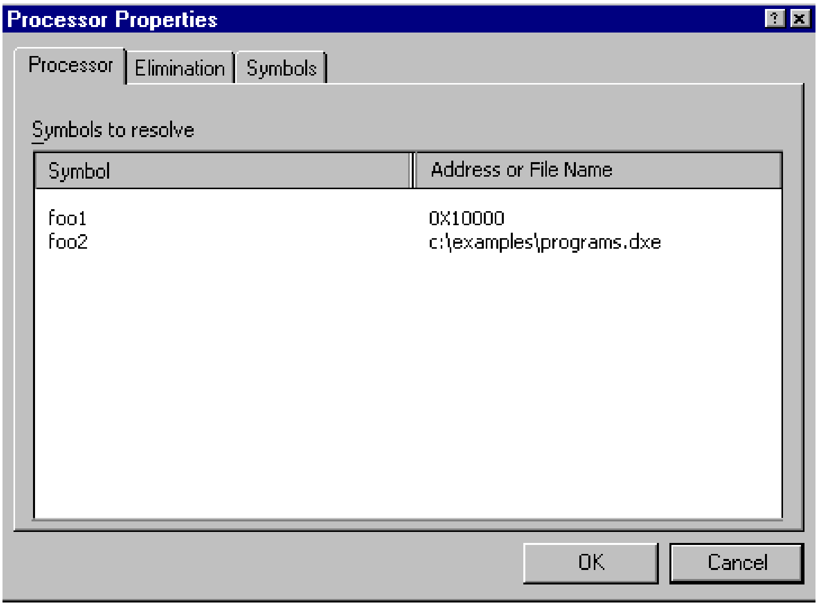


Figure 4-48. Processor Properties Dialog Box – Symbols Tab

The symbols can be resolved to an absolute address or to a program file. Right-clicking in the **Symbols** field allows you to add or remove symbols.

Choosing **Add Symbol** from the menu invokes the **Add Symbol to Resolve** dialog box (Figure 4-49), 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.

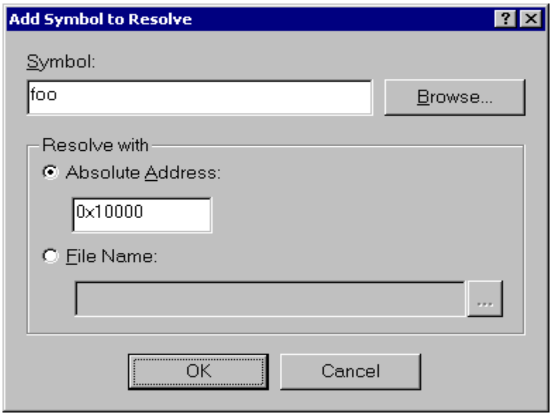


Figure 4-49. Add Symbol to Resolve Dialog Box

The **Browse** button is grayed out when no symbol list is available; for example, if the project has not been linked. When this button is active, click it to display the **Browse Symbols** dialog box, which shows a list of all the symbols.

Selecting a symbol from that list places it in the **Symbol** box of the **Edit Symbol to Resolve** dialog box.

To delete a symbol from the resolve list:

1. Click **Browse** to display the **Symbols to resolve** list (Figure 4-49).
2. Select the symbol to delete.
3. Right-click and choose **Remove Symbol**.

Managing Memory Segment Properties

Specify or change the memory segment's name, start address, end address, size, width, memory space, memory type, and internal/external flag.

To display the **Memory Segment Properties** dialog box (Figure 4-50):

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

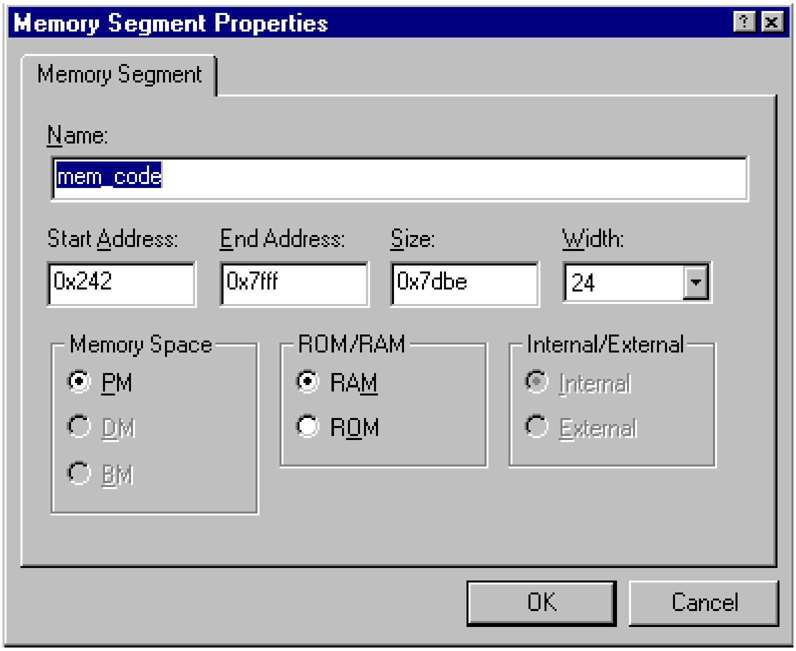


Figure 4-50. Memory Segment Properties Dialog Box

Managing Output Section Properties

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

To specify output section properties:

1. Right-click an output section (for example, `PROGRAM_DXE` or `CODE_DXE`) in the **Memory Map** pane.
2. Choose **Properties** (Figure 4-51).

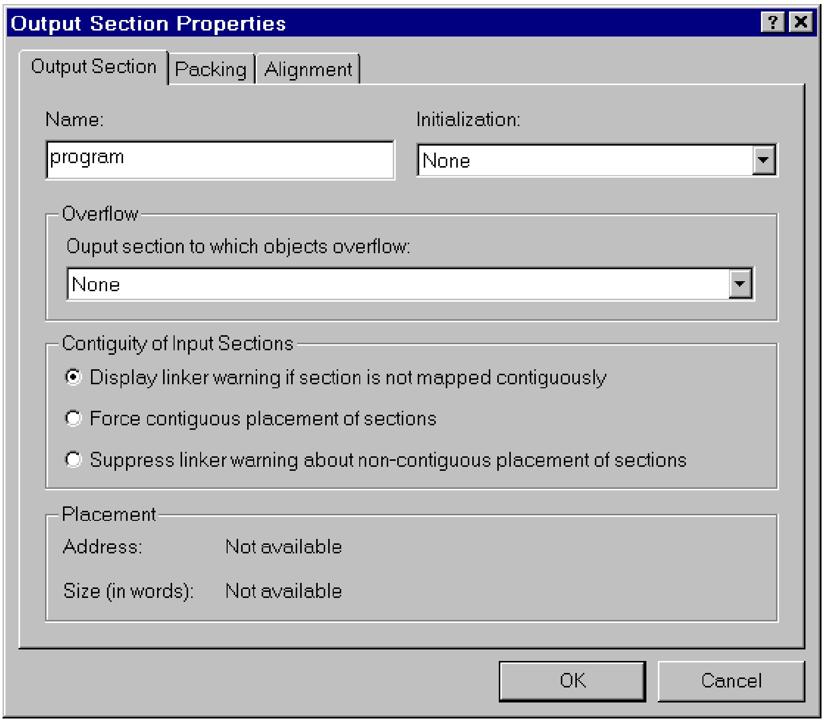


Figure 4-51. Output Section Properties Dialog Box – Output Section Tab

The selections in the output section/segment list include "None" (for no overflow) and "All" output sections. Pin objects to an output section by right-clicking the object and choosing **Pin to output section**.

You can:

- Type a name for the output section in **Name**.
- In **Overflow**, select an output section into which the selected output section will overflow; 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 is done, the box displays the output section's actual address and size.
- **Initialization** allows you to choose the initialization qualifier for an output section. The choices are:
  - **None**: No initialization qualifier is specified for this output section.
  - **No initialization**: No data initialization will happen for this output section.
  - **Initialize to zero**: The memory space for this section will be initialized to zero at either "load" or "runtime", if invoked with the linker's `-meminit` switch. If the `-meminit` switch is not used, the memory is initialized at "load" time when the `.DXE` file is loaded via VisualDSP++ IDDE, or boot-loaded by the boot kernel. If the memory initializer is invoked, the C/C++run-time library (CRTL) will process embedded information to initialize the memory space during the CRTL initialization process.

  - **Initialize at runtime**: If the linker is invoked with the `-meminit` switch, this section will be filled at runtime. If the `-meminit` switch is not specified, the section is filled at "load" time.

- Contiguity of Input Sections allows you to choose whether or not code or data in an output section should be mapped contiguously. The choices are:

  - **Display linker warning if section is not mapped contiguously**

  - **Force contiguous placement of sections**

  - **Suppress linker warning about non-contiguous placement of sections in the operating system**

- Specify the **Packing** (on page 4-65) and **Alignment** (with **Fill value**) properties (on page 4-67) as needed.

Managing Packing Properties

Use the **Packing** tab to specify the packing format that the linker employs 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, and you can change this order. It can be viewed via the **Packing order** box.

To specifypacking properties:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties** and click the **Packing** tab (Figure 4-52).

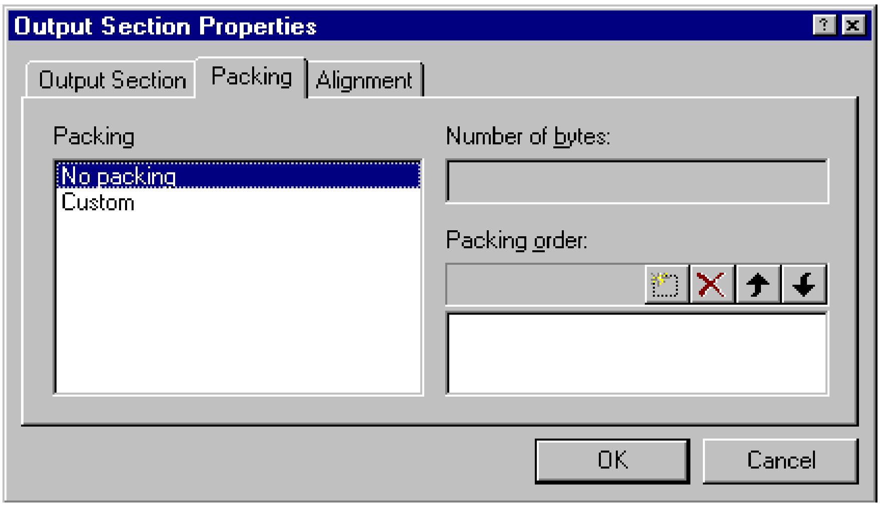


Figure 4-52. Memory Segment Properties Dialog Box – Packing Tab

3. In **Packing** method, select a method.

Method	Description
`No packing`	Specifies no packing. **Number of bytes** and **Packing order** are grayed out.
`Custom`	Permits the selection of number of bytes and packing order.
other choices-	Specifies the number of bytes and packing order of the selected method. The list of packing methods is derived from the included `packing.h` file. Packing method information (number of bytes and packing order) appears, but you cannot change it.

4. In **Number of bytes** (if **Custom** is selected), specify the number of bytes to be reordered at one time. This value does not include the number of null bytes inserted into memory.

5. In **Packing** order, specify byte packing. To do that, select a byte and perform one of these actions:

   - Click the keyboard's **Up** arrow or **Down** arrow key.
   - Drag and drop it to a new location.
   - Insert a null byte by clicking on **Insert**.
   - Delete a null byte by selecting the null byte and clicking **Delete**.

6. Click **OK**.

Managing Alignment and Fill Properties

Use the **Alignment** tab to set the alignment and fill values for the output section. When the output section is aligned on an address, the linker fills the gap with zeros (0), `NOP` instructions, or a specified value.

To specify alignment properties:

1. Right-click a memory segment in the **Memory Map** pane.
2. Choose **Properties**.
3. Click the **Alignment** tab (Figure 4-53).

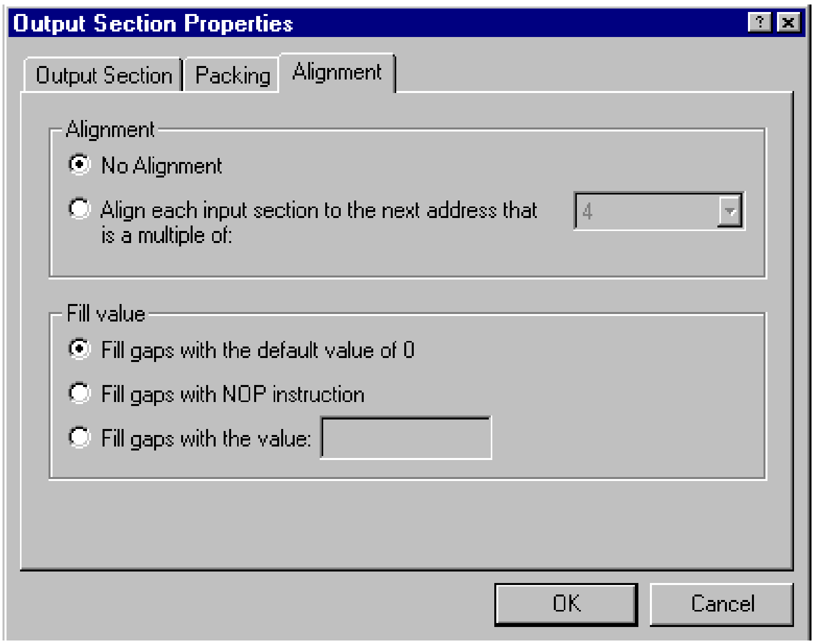


Figure 4-53. Output Section Properties – Alignment Tab

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

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

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

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

Managing Overlay Properties

Use the **Overlay** tab to add/choose the output file for the overlay, its "live" memory, and its linking algorithm.

To specify overlay properties:

1. Right-click an overlay object in the **Memory Map** pane.
2. Choose **Properties** and click the **Overlay** tab (Figure 4-54)

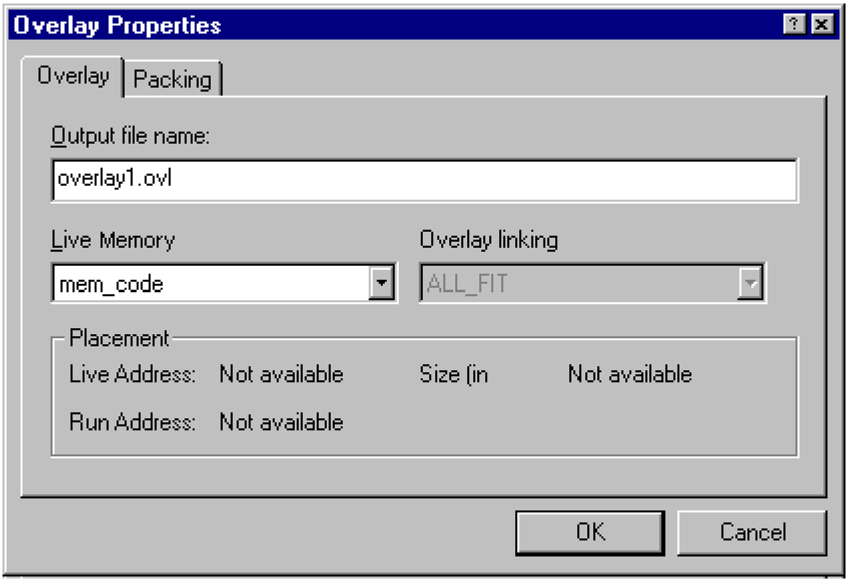


Figure 4-54. Overlay Properties Dialog Box – Overlay Tab

Use the **Output file name** box to specify the name of the overlay file (`.OVL`).

The **Live Memory** drop-down list contains all output sections or memory segments within one output section. The “live” memory is where the overlay is stored before it is swapped into memory.

The **Overlay linking algorithm** box permits one overlay algorithm—`ALL_FIT`. Expert Linker does not currently allow changes to this setting. When `ALL_FIT` is used, the linker tries to fit all of the mapped objects into one overlay.

The **Placement** box provides the following information:

- **Live Address**—The starting address of the overlay
- **Run Address**—The starting address where the overlay is swapped into memory at runtime
- **Size**—The overlay’s size

Click the **Packing** tab to specify byte packing order.

The **Browse** button is only available after the overlay build and when the symbols are available. Clicking **Browse** opens the **Browse Symbols** dialog box.

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

Managing Stack and Heap in Processor Memory

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

Figure 4-55 shows stack and heap output sections in the **Memory Map** pane. Right-click on either of them to display its properties.

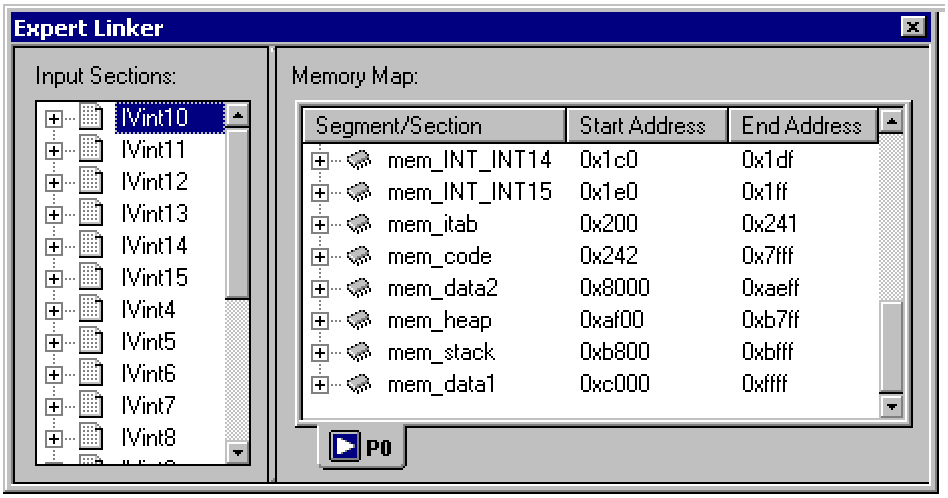


Figure 4-55. Memory Map Window With Stack and Heap Sections

Use the **Global Properties** dialog box to select **Show stack/heap usage** (Figure 4-56). This option graphically displays the stack/heap usage in memory (Figure 4-57).

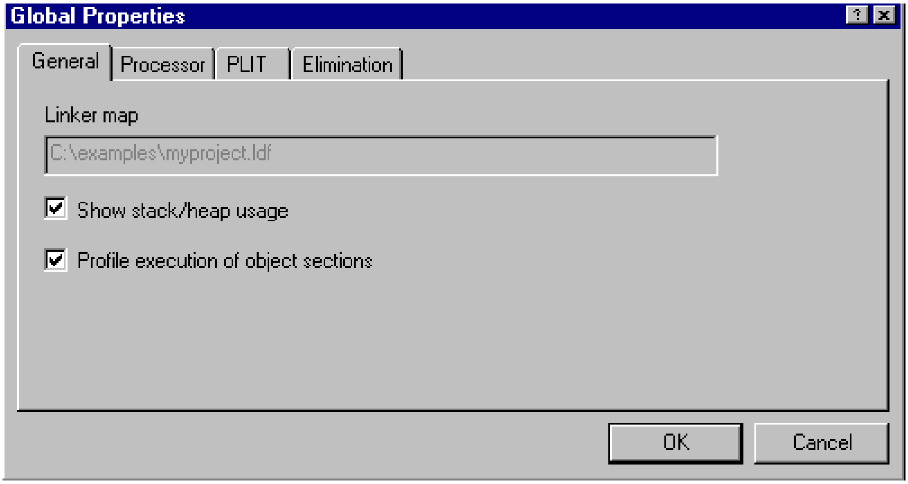


Figure 4-56. Global Properties – Selecting Stack and Heap Usage

The Expert Linker can:

- Locate stacks and heaps and fill them with a marker value.

  This occurs after loading the program into a processor target. The stacks and heaps are located by their output section names, which may vary across processor families.

- Search the heap and stack for the highest memory locations written to by the DSP program.

  This action occurs when the target halts after running the program. (assume the unused portion of the stack or heap starts here). The Expert Linker updates the memory map to show how much of the stack and heap are unused.

Use this information to adjust the size of your stack and heap. This information helps make better use of the processor memory, so the stack and heap segments do not use too much memory.

Use the graphical view (**View Mode -> Graphical Memory Map**) to display stack and heap memory map blocks. Figure 4-57 shows a possible memory map after running a project program.

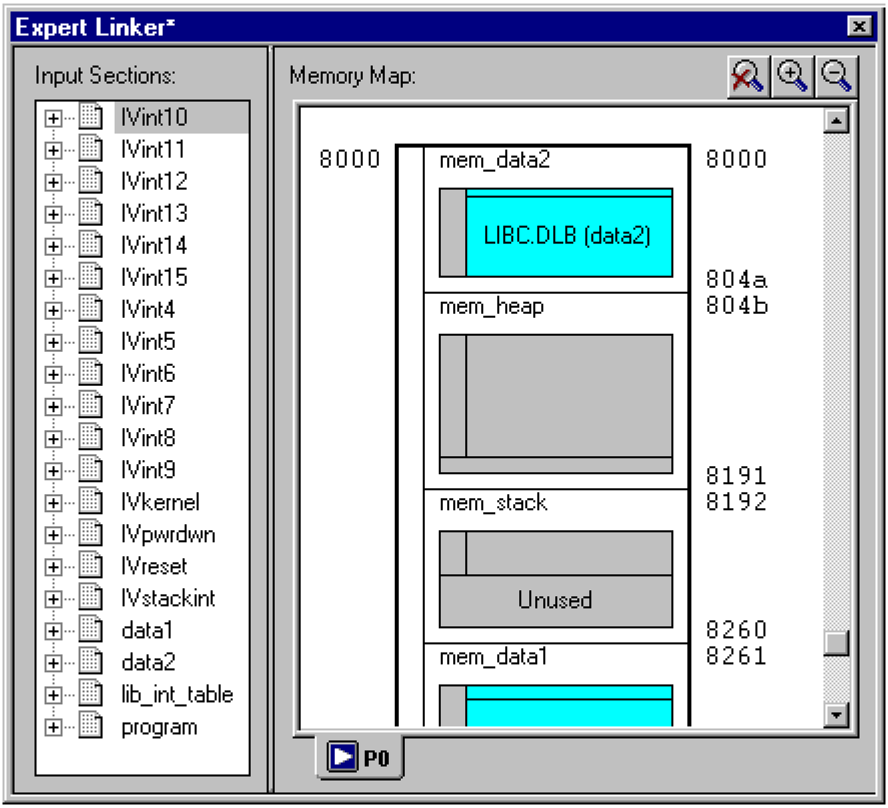


Figure 4-57. Graphical Memory Map Showing Stack and Heap Usage

Managing Shared Memory Properties

Specify the path and name of the file used by shared memory. This procedure assumes the Expert Linker window is open.

To specify shared memory properties:

1. In the **Memory Map** pane, click the **Shared Memory** tab (located at the bottom of dialog box).

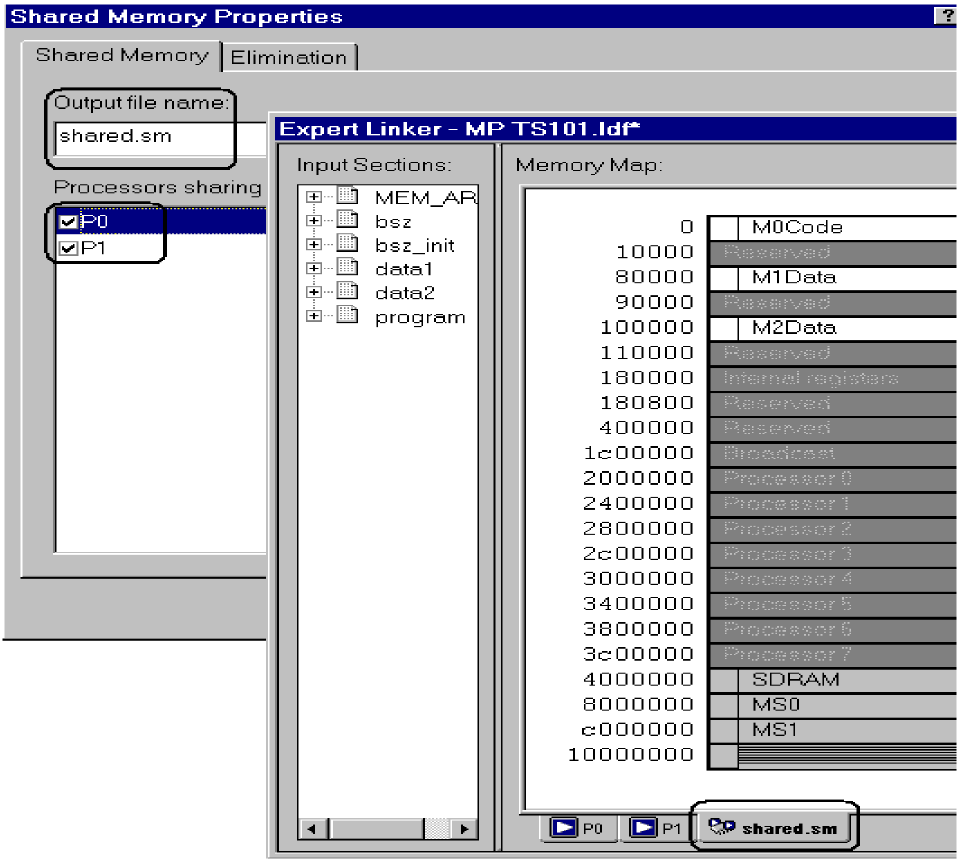


Figure 4-58. Shared Memory Tab

2. Right-click anywhere on the **Memory Map** pane.
   Note: Do not right-click on a memory segment, output section, input section, or overlay.

3. Choose **Properties**.
   The **Shared Memory** page of the **Shared Memory Properties** dialog box appears.

4. In **Output file name**, specify the name of the output file for the shared memory.

5. In **Processors sharing this memory**, select the processors that share the file whose name appears in Output file name. Selecting a processor links its executable file against this shared memory file.

6. Optionally, click the **Elimination** tab (see "Managing Elimination Properties" on page 4-56) and specify options.

7. Click **OK**.

5 MEMORY OVERLAYS AND ADVANCED LDF COMMANDS

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

This chapter includes:

This chapter generally uses code examples for TigerSHARC processors. If used, other processor's code examples are marked accordingly.

Overview

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

Since internal memory is much faster than external memory, it may be desirable to keep large amounts of program code in external memory and to swap parts of it to internal memory for speed in execution. Such a program is said to run in "*overlays*."

In case of Blackfin architecture, to use the full memory bandwidth, Blackfin processors possess an instruction cache. Instructions which come from cache memory free up the program memory bus for data access fetch from program memory. This feature permits multiply accumulate instructions to fetch a multiplier and a multiplicand, compute a product, and add the new product to the accumulator, all in a single instruction. In this case, it is important to locate the multiplier data in program memory and the multiplicand data in data memory.

Blackfin processors have an integrated instruction and data cache, which can be used to reduce the manual process of moving instructions and data in and out of core. Memory overlays can also be used to run an application efficiently, but they require a user-managed set of DMAs. Since Blackfin processors do not have a separate built-in overlay manager, the DMA controller must be used to move code in and out of internal L1 memory. For this reason, it is almost always better to use the instruction cache.

It is the programmer's responsibility to assign data buffers to memory. This is done with instructions to the linker. In addition, each DSP project must fit into a different memory arrangement. Different DSP boards have different amounts of memory, located at different addresses.

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

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

The overlay manager is a user-defined function responsible for insuring that a required symbol (function or data) within an overlay is in the run-time memory when it is needed. The transfer occurs using the direct memory access (DMA) capability of the processor. The overlay manager may also handle other advanced functionality described in "Introduction to Memory Overlays" on page 5-5 and "Overlay Managers" on page 5-7.

Memory Management Using Overlays

To reduce DSP system costs, many applications employ processors with small amounts of on-chip memory and place much of the program code and data off-chip. The linker supports the linking of executable files for systems with overlay memory. Applications notes on the Analog Devices Web site provide detailed descriptions of this technique; for example,

- EE-66 "Using Memory Overlays " for SHARC processors
- EE-180 "Using Code Overlays from ROM on the ADSP-21161N EZ-KIT Lite"
- EE-230 "Code Overlays on the ADSP-2126x SHARC Family of DSPs"

This section describes the use of memory overlays. The topics are:

The following LDF commands facilitate overlay features.

- "OVERLAY_GROUP{}" on page 5-32
- "PLIT{}" on page 5-36

Introduction to Memory Overlays

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

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

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

Figure 5-1 demonstrates the concept of memory overlays. The two memory spaces are: *internal* and *external*. The *external* memory is partitioned into four overlays. The *internal* memory contains the main program, an overlay manager function, and two memory segments reserved for execution of overlay program instructions.

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

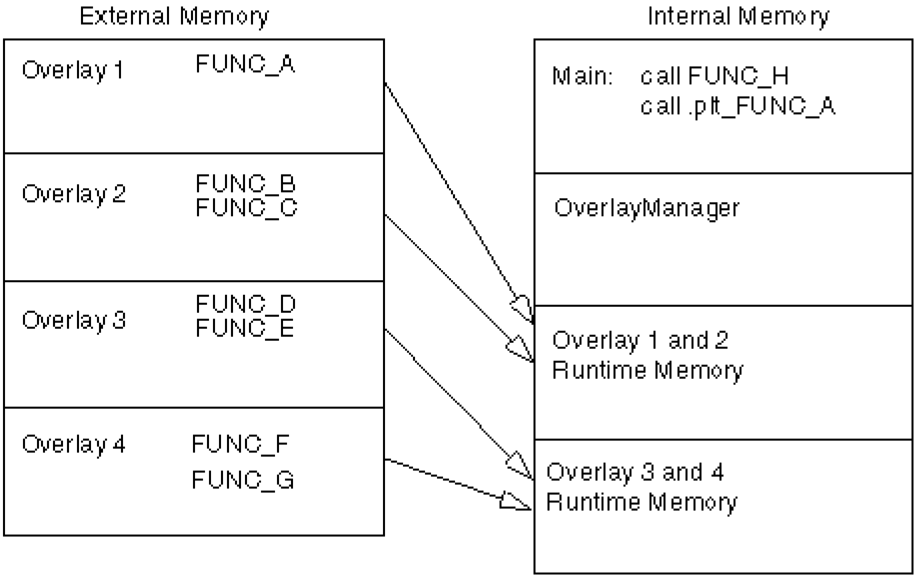


Figure 5-1. Memory Overlays

the location in internal memory where overlay 2 is designated to run. When `FUNC_D` is required, the overlay manager loads overlay 3 into its designated run-time memory.

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

Overlay Managers

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

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

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

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

  If the overlay is already in internal memory, the overlay transfer is bypassed and execution of the overlay routine begins immediately
- Load an overlay while executing a function from a second overlay (or a non-overlay function)

You may require an overlay manager to perform other specialized tasks to satisfy the special needs of a given application. Overlay managers for Blackfin processors must be developed by the user. Refer to EE-66, EE-180, and EE-230 (for SHARC processors) for example overlay managers.

Breakpoints on Overlays

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

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

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

Memory Overlay Support

The overlay support provided by the DSP tools includes:

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

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

Listing 5-1 shows the portion of an `.LDF` file that defines two overlays. This overlay declaration configures the two overlays to share a common run-time memory space. The syntax for the `OVERLAY_INPUT{}` command is described in "OVERLAY_GROUP{}" on page 3-39.

In this TigerSHARC code example, OVLY_one contains FUNC_A and lives in memory segment ovl_code; OVLY_two contains functions FUNC_B and FUNC_C and also lives in memory segment ovl_code.

Listing 5-1. Overlay Declaration in an LDF File

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

    OVERLAY_INPUT {
      OVERLAY_OUTPUT (OVLY_two.ovl)
      INPUT_SECTIONS (FUNC_B.doj(program) FUNC_C.doj(sec_code))
      } >ovl_code
  } >M0Code
```

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

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

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

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

Overlay "live" and "run" word sizes differ when internal memory and external memory widths differ. A system containing either 16-bit-wide or 32-bit-wide external memory requires data packing to store an overlay containing instructions.

> The Blackfin processor architecture supports byte addressing that uses 16-bit, 32-bit, or 64-bit opcodes. Thus, no data packing is required.
> The SHARC processor architecture supports 48-bit instruction words. The overlay "live" word size (number of words in the overlay) is based on the number of 32-bit words required to pack all of the 48-bit instructions.

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

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

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

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

```
j4 = [j0 += j3];;
j5 = j4 * j6;;
CALL FUNC_A;;          /* Call to function in overlay */
[j3 += J3] = j5;;
```

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

```
j4 = [j0 += j3];;
j5 = j4 * j6;;
CALL .plt_FUNC_A;      / * Call to PLIT entry */
[j3 += j3] = j5;;
```

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

Listing 5-2 is an example PLIT definition from an .LDF file, where register j4 is set to the value of the overlay ID that contains the referenced symbol and register j5 is set to the run-time address of the referenced symbol. The last instruction branches to the overlay manager that uses the initialized registers to determine which overlay to load (and where to jump to execute the called overlay function).

Listing 5-2. PLIT Definitions in LDF

```
PLIT    //   TigerSHARC PLIT
{
   j4 = PLIT_SYMBOL_OVERLAYID;;
   j5 = PLIT_SYMBOL_ADDRESS;;
   JUMP OverlayManager;;
}

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

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

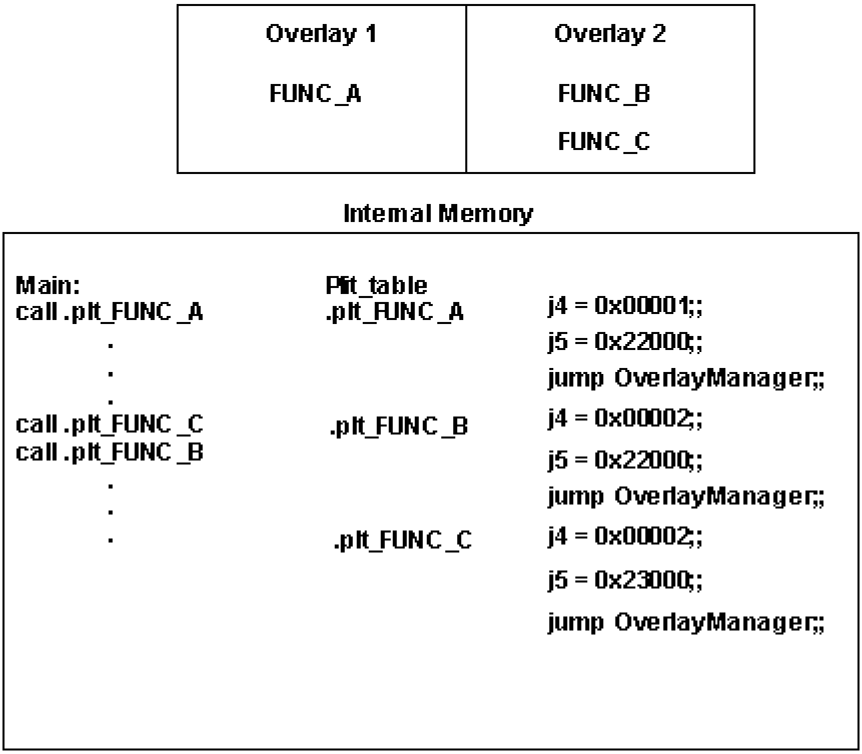


Figure 5-2. Expanded PLIT Table (for TigerSHARC Processors)

For example, the first PLIT entry in Listing 5-2 is for the overlay symbol `FUNC_A`. The linker replaces the constant name `PLIT_SYMBOL_OVERLAYID` with the ID of the overlay containing `FUNC_A`. The linker also replaces the constant name `PLIT_SYMBOL_ADDRESS` with the run-time address of `FUNC_A`.

When the overlay manager is called via the jump instruction of the PLIT table, `j4` contains the referenced function's overlay ID and `j5` contains the referenced function's run-time address. The overlay manager uses the overlay ID and run-time address to load and execute the referenced function.

Example – Managing Two Overlays

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

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

The overlay manager:

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

Several code segments for the LDF and the overlay manager follow with appropriate explanations.

Listing 5-3. FFT Overlay Example 1

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

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

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

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

To verify whether an overlay is in internal memory, place the overlay ID of this overlay into a register (for example, `j4`) and compare this value to the overlay ID of each overlay already loaded by loading these overlay values into a register (for example, `j5`).

```
              /* Is overlay already in internal memory? */
   j4 = j4 - j5;;
              /* If so, do not transfer it in. */
   if jeq, jump skipped_DMA_setup;;
```

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

The following code segment calls the four FFT functions.

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

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

```
PLIT
{
    j4 = PLIT_SYMBOL_OVERLAYID;;
    j5 = PLIT_SYMBOL_ADDRESS;;
    JUMP OverlayManager(db);;
}
```

Register `j4` contains the overlay ID with the referenced symbol, and register `j5` contains the run-time address of the referenced symbol. The final instruction moves the program counter (PC) to the starting address of the overlay manager. The overlay manager uses the overlay ID in conjunction with the overlay constants generated by the linker to transfer the proper overlay into internal memory. Once the transfer is complete, the overlay manager sends the PC to the address of the referenced symbol stored in `j5`.

Linker-Generated Constants

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

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

The constants provide the following information to the overlay manager.

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

Overlay Word Sizes

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

Table 5-1 shows the linker-generated constants and examples of processor-specific addresses.

The overlay manager places the constants in arrays as shown in Figure 5-3 and Figure 5-4. The arrays are referenced by using the overlay ID as the index to the array. The index or ID is stored in a Modify register (`Jn/Kn`

Table 5-1. Linker-Generated Constants and Processor-Specific Addresses

Constant	TigerSHARC	SHARC	Blackfin
_ov_startaddress_1	0x04000000	0x20000	0x00000000
_ov_startaddress_2	0x04000080	0x20000	0x00000010
_ov_endaddress_1	0x0400007F	0x20017	0x0000000F
_ov_endaddress_2	0x04000100	0x00018	0x00000010
_ov_word_size_run_1	0x04000080	0x00018	0x00000010
_ov_word_size_run_2	0x04000080	0x00018	0x00000010
_ov_word_size_live_1	0x04000080	0x00010	0x00000010
_ov_word_size_live_2	0x04000080	0x00010	0x00000010
_ov_runtimestartaddress_1	0x04001000	0x08800	0xF0001000
_ov_runtimestartaddress_2	0x04001080	0x08800	0xF0001000

for TigerSHARC processors and `M#` for SHARC and Blackfin processors), and the beginning address of the array is stored in the Index register (`Jm/Km` for TigerSHARC processors and `I#` for SHARC and Blackfin processors).

```
.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;
```

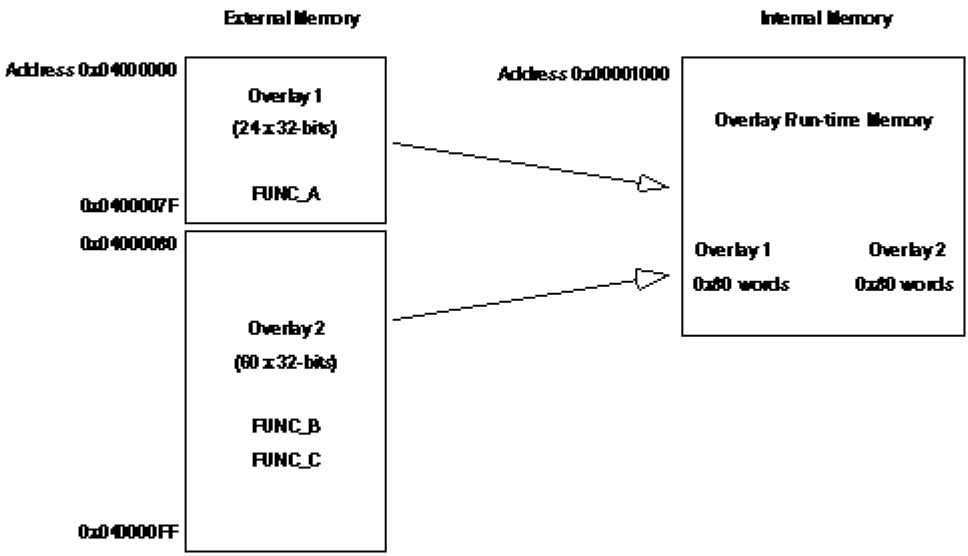


Figure 5-3. TigerSHARC Processor Overlay Live and Run Memory Sizes

Figure 5-4 shows the difference between overlay “live” and “run” size in SHARC processor memory:

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

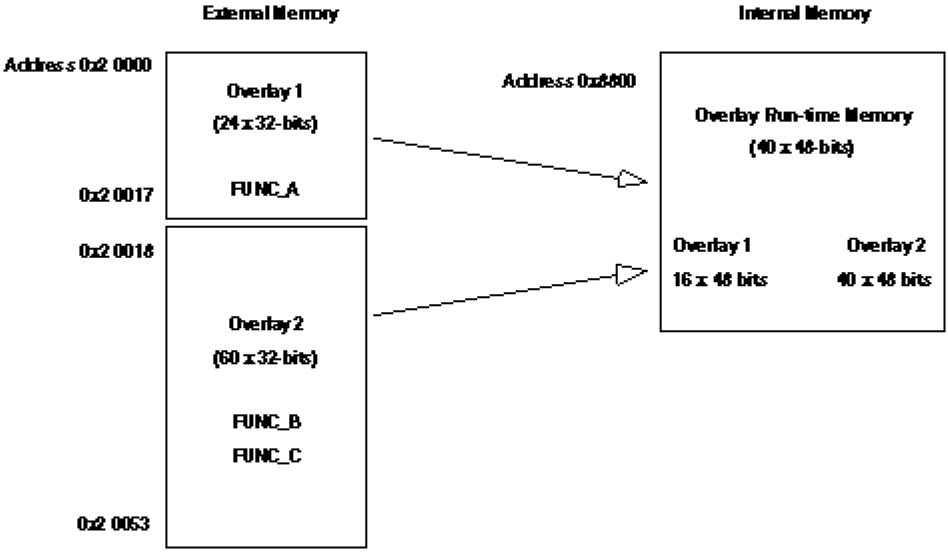


Figure 5-4. SHARC Processor Overlay Live and Run Memory Sizes

Storing Overlay ID

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

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

Overlay Manager Function Summary

In summary, the overlay manager routine does the following.

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

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

Reducing Overlay Manager Overhead

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

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

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

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

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

The overlay manager first determines whether the referenced symbol is in internal memory. If not, it sets up the DMA transfer. If the symbol is not in internal memory and the flag is set for execution, the core waits 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 `j4` is set to the value placed in the flag buffer, and the value in `j4` 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 whether the function call is a load or an execute.

```
call fft_first_2_stages;;
j4 = 0;;
[j31 + prefetch] = j4;;
call fft_middle_stages;;
j4 = 0;;
[j31 + prefetch] = j4;;
call fft_next_to_last;;
j4 = 0;;
[j31 + prefetch] = j4;;
call fft_last_stage;;
j4 = 0;;
[j31 + prefetch] = j4;;
```

The next set of instructions represents a load function call.

```
call fft_middle_stages;;
     /* This function call pre-loads the function */
     /* into the overlay run memory. */
j4 = 1;;
```

```
[j31 + prefetch] = j4;;
     /* Set pre-fetch flag to 1 to indicate a load. */
```

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

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

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

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

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

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

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

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

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

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

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

Using PLIT{} and Overlay Manager

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

Refer to "PLIT{}" on page 5-36 for basic syntax information. Refer to "Introduction to Memory Overlays" on page 5-5 for detailed information on overlays.

Figure 5-5 shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker generates special symbols for overlays. These overlay symbols are:

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

The # indicates the overlay number.

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

The two functions in Figure 5-5 describe different overlays. By default, the linker generates PLIT code only when an unresolved function reference is resolved to a function definition in overlay memory.

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

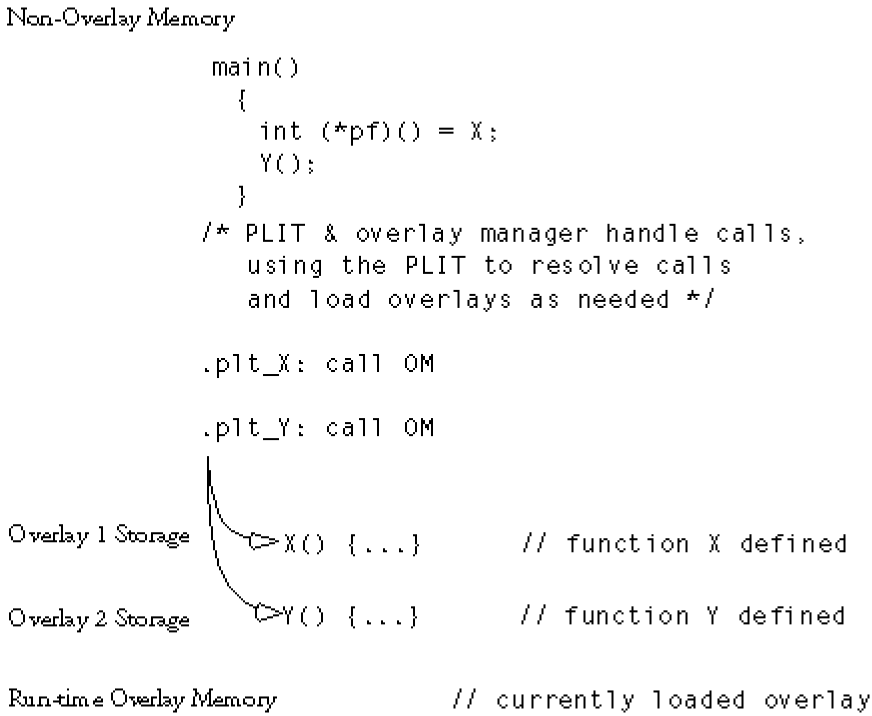


Figure 5-5. PLITs and Overlay Memory; main() Calls to Overlays

When the reference and the definition reside in the same executable file, the linker does not generate PLIT code. However, you can force the linker to output a PLIT, even when all references can be resolved locally. The `.plit` command sets up data for the overlay manager, which first loads the overlay that defines the desired symbol, and then branches to that symbol.

Inter-Overlay Calls

PLITs can be used to resolve inter-overlay calls, as shown in Figure 5-6. Structure the `.LDF` file in a way that ensures the PLIT code generated for inter-overlay function references is part of the `.plit` section for `main()`, which is stored in non-overlay memory.

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

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

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

Inter-Processor Calls

PLITs resolve inter-processor overlay calls, as shown in Figure 5-7, for systems that permit one processor to access the memory of another processor.

When one processor calls into another processor's overlay, the call increases the size of the `.plit` section in the executable file that manages the overlay.

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

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

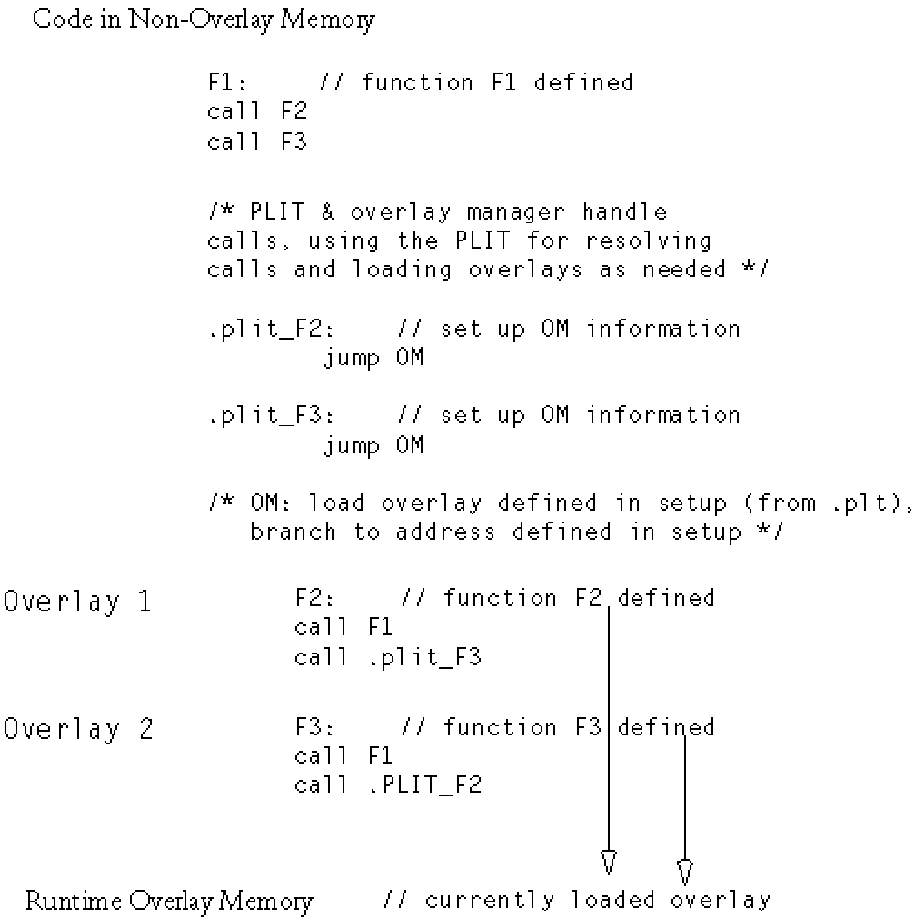


Figure 5-6. PLITs and Overlay Memory – Inter-Overlay Calls

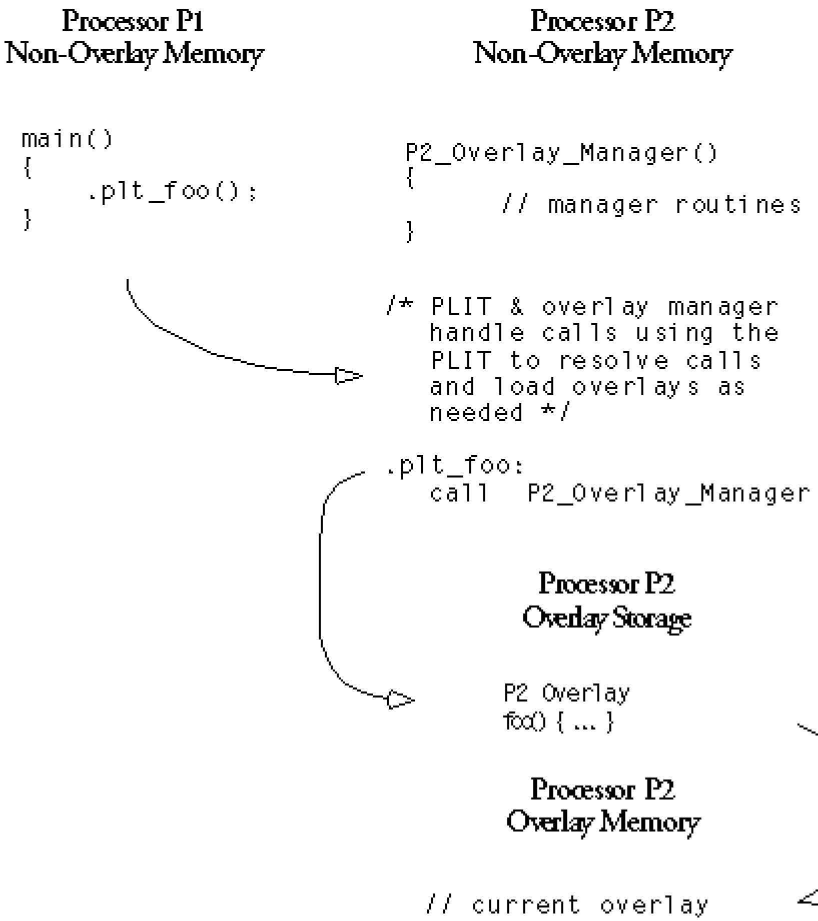


Figure 5-7. PLITs and Overlay Memory – Inter-Processor Calls

Advanced LDF Commands

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

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

- "MPMEMORY{}"
- "OVERLAY_GROUP{}" on page 5-32
- "PLIT{}" on page 5-36
- "SHARED_MEMORY{}" on page 5-40

For detailed information on other LDF commands, refer to "LDF Commands" on page 3-29.

MPMEMORY{}

The MPMEMORY{} command specifies the offset of each processor's physical memory in a multiprocessor target system. After you declare the processor names and memory segment offsets with the MPMEMORY{} command, the linker uses the offsets during multiprocessor linking. Refer to "Memory Overlay Support" on page 5-8 for a detailed description of overlay functionality.

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

Figure 5-8 shows MPMEMORY{} command syntax.

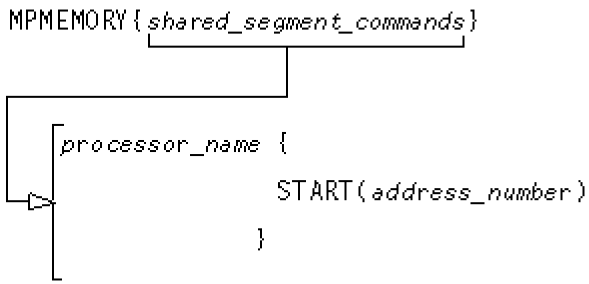


Figure 5-8. MPMEMORY{} Command Syntax Tree

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

- *`shared_segment_commands`* – Contains *`processor_name`* declarations with a `START{}` address for each processor's offset in multiprocessor memory. Processor names and linker labels follow the same rules. For more information, refer to "LDF Expressions" on page 3-18.

- *`processor_name{placement_commands}`* – Applies the *`processor_name`* offset for multiprocessor linking. Refer to "PROCESSOR{}" on page 3-45 for more information.

ⓘ The `MEMORY{}` command specifies the memory map for the target system. The LDF must contain a `MEMORY{}` command for global memory on the target system and may contain a `MEMORY{}` command that applies to each processor's scope. An unlimited number of memory segments can be declared within each `MEMORY{}` command. For more information, see "MEMORY{}" on page 3-35. See "Memory Characteristics Overview" on page 2-31 for memory map descriptions.

OVERLAY_GROUP{}

The OVERLAY_GROUP{} command provides legacy support. This command is deprecated and is not recommended for use. When running the linker(with a SHARC processor), the following warning may occur.

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

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

Overlays may be *grouped* or *ungrouped*. Use the OVERLAY_INPUT{} command to support ungrouped overlays. Refer to "Memory Overlay Support" on page 5-8 for a detailed description of overlay functionality.

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

Overlay declarations syntactically resemble the SECTIONS{} commands. They are portions of SECTIONS{} commands.

The OVERLAY_GROUP{} command syntax is:

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

Figure 5-9 demonstrates grouped overlays.

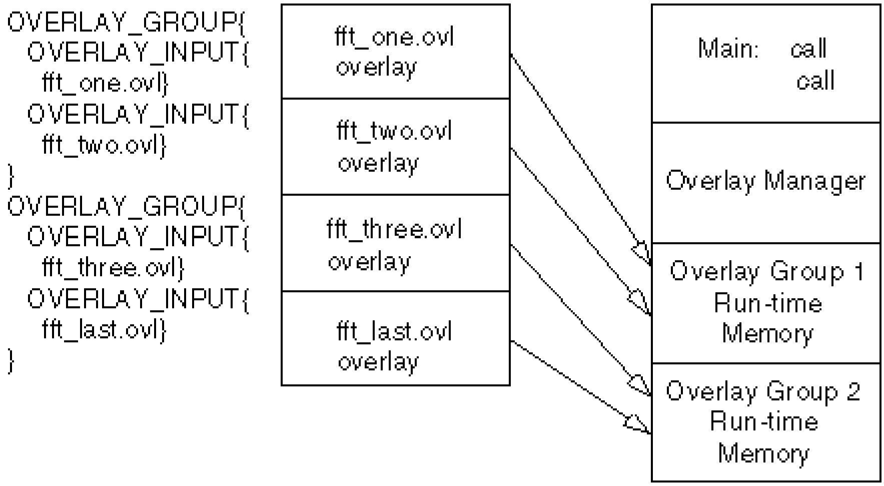


Figure 5-9. Example of Overlays – Grouped

In the simplified examples in Listing 5-4 and Listing 5-5, the functions are written to overlay (`.OVL`) files. Whether functions are disk files or memory segments does not matter (except to the DMA transfer that brings them in). Overlays are active only while being executed in run-time memory, which is located in the `program` memory segment.

Ungrouped Overlay Execution

In Listing 5-4, as the FFT progresses and overlay functions are called in turn, they are brought into run-time memory in sequence as four function transfers. Figure 5-10 shows the ungrouped overlays.

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

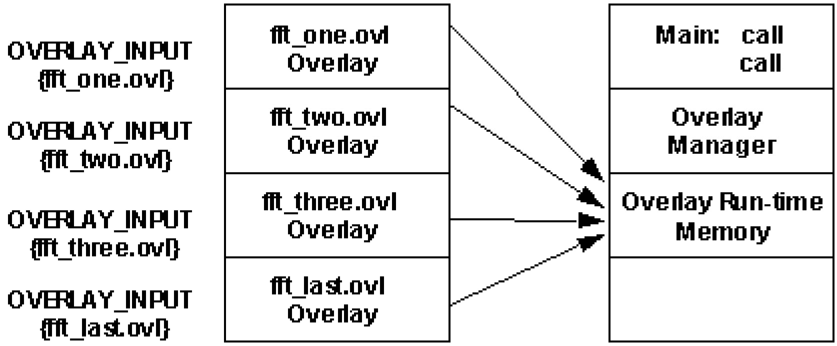


Figure 5-10. Example of Overlays – Not Grouped

Listing 5-4. LDF Overlays – Not Grouped

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

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

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

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

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

Grouped Overlay Execution

Listing 5-5 shows a different implementation of the same algorithm. The overlay functions are grouped in pairs. Since all four pairs of routines reside simultaneously, the processor executes both routines before paging.

Listing 5-5. LDF Overlays – Grouped

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

PLIT{}

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

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

The `PLIT{}` command provides a template from which the linker generates assembly code when a symbol resolves to a function in overlay memory. The code typically handles a call to a function in overlay memory by calling an overlay memory manager. Refer to "Memory Overlay Support" on page 5-8 for a detailed description of overlay and PLIT functionality.

A `PLIT{}` command may appear in the global LDF scope, within a `PROCESSOR{}` command or within a `SECTIONS{}` command. For an example of using a `PLIT{}` command, see "Using PLIT{} and Overlay Manager" on page 5-25.

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

PLIT Syntax

Figure 5-11 shows the general syntax of the `PLIT{}` command and indicates how the linker handles a symbol (`symbol`) local to an overlay function.

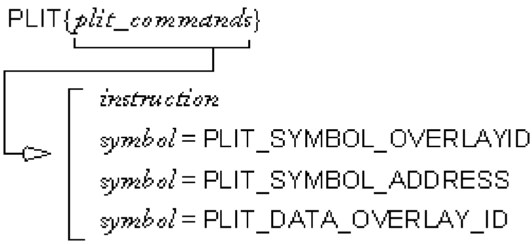


Figure 5-11. PLIT{} Command Syntax Tree

Parts of the `PLIT{}` command are:

- *`instruction`* – None, one, or multiple assembly instructions. The instructions may occur in any reasonable order in the command structure and may precede or follow symbols. The following two constants contain information about *symbol* and the overlay in which it occurs. You must supply instructions to handle that information.

- `PLIT_SYMBOL_OVERLAYID` – Returns the overlay ID

- `PLIT_SYMBOL_ADDRESS` – Returns the absolute address of the resolved symbol in run-time memory

Command Evaluation and Setup

The linker first evaluates the sequence of assembly code in each *`plit_command`*. Each line is passed to a processor-specific assembler, which supplies values for the symbols and expressions. After evaluation, the linker places the returned bytes into the `.PLIT` output section and manages the addressing in that output section.

To help write an overlay manager, the linker generates PLIT constants for each symbol in an overlay. Data can be overlaid, just like code. If an overlay-resident function calls for additional data overlays, include an instruction for finding them.

After the setup and variable identification are completed, the overlay itself is brought (via DMA transfer) into run-time memory. This process is controlled by assembly code called an overlay manager.

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

Overlay PLIT Requirements and PLIT Examples

Both the `.plit` output section (allocating space for PLIT) and the `PLIT{}` command are necessary when specifying PLIT for overlays. The `.LDF` file must allocate space in memory to hold PLITs built by the linker. Typically, that memory resides in the program code memory segment.

No input section is associated with the `.plit` output section. The LDF allocates space for linker-generated routines, which do not contain (input) data objects.

A typical LDF declaration for that purpose is:

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

This segment allocation does not take any parameters. You write the structure of this command according to the PLIT syntax. The linker creates an instance of the command for each symbol that resolves to an overlay. The linker stores each instance in the `.plit` output section, which becomes part of the program code's memory segment.

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

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

Simple PLIT – States are not Saved

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

The following is an example of `PLIT{}` command implementation for TigerSHARC processors.

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

PLIT
{
    j4 = PLIT_SYMBOL_OVERLAYID;;
    j5 = PLIT_SYMBOL_ADDRESS;;
    JUMP OverlayManager(db);;
}
```

The following is an example of `PLIT{}` command implementation for Blackfin processors.

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

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

PLIT – Summary

A PLIT is a template of instructions for loading an overlay. For each overlay routine in the program, the linker builds and stores a list of PLIT instances according to that template, as it builds its executable file. The linker may also save registers or stack context information. The linker does not accept a PLIT without arguments.

If you do not want the linker to redirect function calls in overlays, omit the `PLIT{}` commands entirely.

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

The overlay manager can also:

- Be helped by manual intervention. Save the target's state on the stack or in memory before loading and executing an overlay function, to ensure it continues correctly on return. However, you can implement this feature within the PLIT section of your LDF. **Note:** Your program may not need to save this information.
- Initiate (jump to) the routine that transfers the overlay code to internal memory, after given the previous information about its identity, size and location: `_OverlayManager`. "Smart" overlay managers first check whether an overlay function is already in internal memory to avoid reloading the function.

SHARED_MEMORY{}

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

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

Your LDF may contain multiple `SHARED_MEMORY{}` commands, but the maximum number of processors that can access a shared memory is processor-specific. The `SHARED_MEMORY{}` command must appear within the scope of a `MEMORY{}` command. The `PROCESSOR{}` commands declaring the processors that share this memory must also appear within this scope.

Figure 5-12 shows the syntax for the `SHARED_MEMORY{}` command, followed by definitions of its components.

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

Figure 5-12. SHARED_MEMORY{} Command Syntax

The command components are:

- `OUTPUT()` – Specifies the output file name (*`file_name`*`.SM`) of the shared memory executable (`.SM`) file. An `OUTPUT()` command in a `SHARED_MEMORY{}` command must appear before the `SECTIONS{}` command in that scope.

- `SECTIONS()` – Defines sections for placement within the shared memory executable (`.SM`) file.

Figure 5-13 shows the scope of `SHARED_MEMORY{}` commands in the LDF.

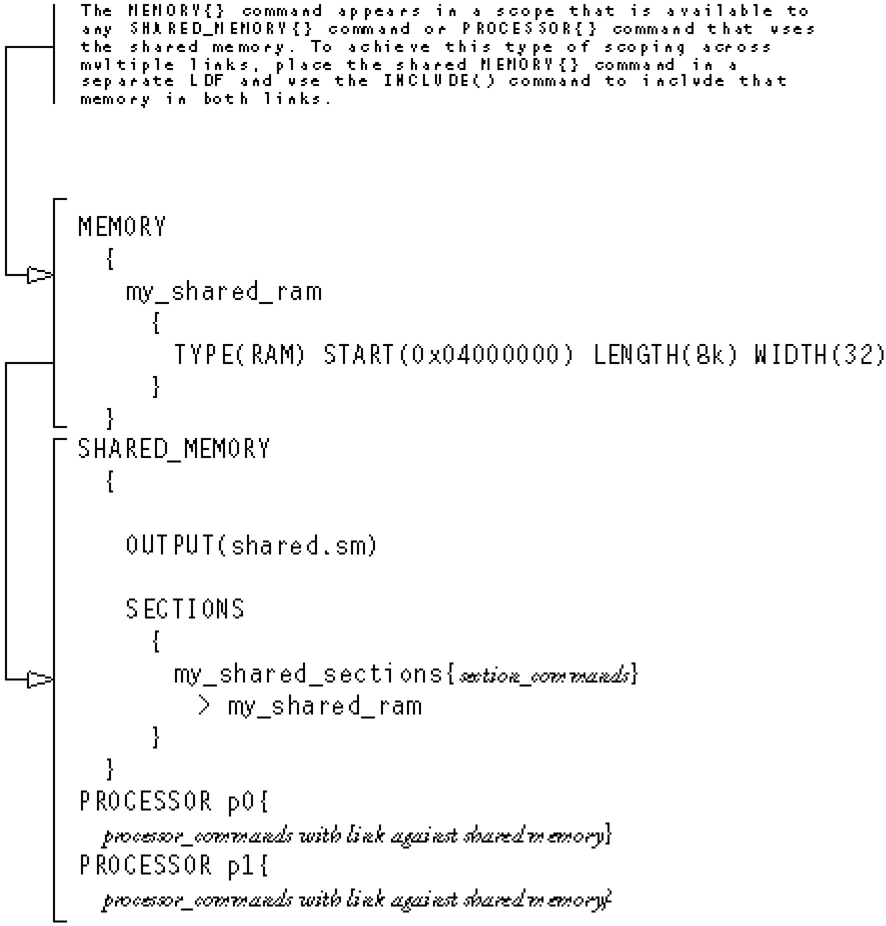


Figure 5-13. LDF Scopes for SHARED_MEMORY{} Command

6 ARCHIVER

The VisualDSP++ archiver, `elfar.exe`, combines object (`.DOJ`) files into library files, which serve as reusable resources for code development. The VisualDSP++ linker rapidly searches library files for routines (library members) referred to by other object files and links these routines into the executable program.

This chapter provides:

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

Archiver Guide

The elfar.exe utility combines and indexes object files (or any other files) to produce a searchable library file. It performs the following operations, as directed by options on the elfar command line:

- Creates a library file from a list of object files
- Appends one or more object files to an existing library file
- Deletes file(s) from a library file
- Extracts file(s) from a library file
- Prints the contents of a specified object file of an existing library file to stdout
- Replaces file(s) in an existing library file
- Encrypts symbol(s) in an existing library file
- Allows embedded version information into a library built with elfar

The archiver can run only one of these operations at a time. However, for commands that take a list of file names as arguments, the archiver can input a text file that contains the names of object files (separated by white space). The operation makes long lists easily manageable.

The archiver, which is sometimes called a librarian, is a general-purpose utility. It combines and extracts arbitrary files. This manual refers to DSP object (.DOJ) files because they are relevant to DSP code development.

Creating a Library From VisualDSP++

Within the VisualDSP++ development environment, you can choose to create a library file as your project's output. To do so, specify **DSP library file** as the target type on the **Project** page of the **Project Options** dialog box.

VisualDSP++ writes its output to `<projectname>.DLB`. To modify or list the contents of a library file or perform any other operations on it, run the archiver from the `elfar` command line (as shown in "Archiver Command-Line Reference" on page 6-11).

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

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

Table 6-1. File Name Extensions used with Archiver

Extension	File Description
`.DLB`	Library file
`.DOJ`	Object file. Input to archiver.
`.TXT`	Text file used as input with the `-i` switch

Making Archived Functions Usable

In order to use the archiver effectively, you must know how to write archive files, which make your DSP functions available to your code (via the linker), and how to write code that accesses these archives.

Archive usage consists of two tasks:

- Writing *archive routines*, functions that can be called from other programs
- Accessing archive routines from your code

Writing Archive Routines: Creating Entry Points

An archive routine (or function) in code can be accessed by other programs. Each routine must have a globally visible start label (*entry point*). Code that accesses that routine must declare the entry point's name as an external variable in the calling code.

To create entry points:

1. Declare the start label of each routine as a global symbol with the assembler's `.GLOBAL` directive. This defines the entry point.

   The following code fragment has two entry points, `dIriir` and `FAE`.

   ```
    ...
    .global dIriir;
    .section data1;
    .byte2 FAE = 0x1234,0x4321;

   .section program;
   .global FAE;
   dIriir: R0=N-2;

   P2 = FAE;
   ```

2. Assemble and archive the code containing the routines. Use either of the following methods.

   - Direct VisualDSP++ to produce a library (see "Creating a Library From VisualDSP++" on page 6-3). When building the project, the object code containing the entry points is packaged in `<projectname>.DLB`. You can extract an object file, for example, to incorporate it in another project.

   - When creating executable or unlinked object files from VisualDSP++, archive them afterwards from the `elfar` command line.

Accessing Archived Functions From Your Code

Programs that call a library routine must use the assembler's `.EXTERN` directive to specify the routine's start label as an external label. When linking the program, specify one or more library (`.DLB`) files to the linker, along with the names of the object (`.DOJ`) 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. Linking libraries is faster than using individual object files, and you do not have to enter all the file names, just the library name.

In the following example, the archiver creates the `filter.dlb` library containing the object files: `taps.doj`, `coeffs.doj`, and `go_input.doj`.

```
elfar -c filter.dlb taps.doj coeffs.doj go_input.doj
```

If you then run the linker with the following command line, the linker links the object files `main.doj` and `sum.doj`, uses the default `.LDF` file (for example, `ADSP-TS201.ldf`), and creates the executable file (`main.dxe`).

```
linker -DADSP-TS201 main.doj sum.doj filter.dlb -o main.dxe
```

Assuming that one or more library routines from `filter.dlb` are called from one or more of the object files, the linker searches the library, extracts the required routines, and links the routines into the executable file.

Archiver File Searches

File searches are important in the archiver's process. The archiver supports relative and absolute directory names, default directories, and user-specified directories for file search paths. File searches include:

- *Specified path* – If relative or absolute path information is included in a file name, the archiver searches only in that location for the file.
- *Default directory* – If the path information is not included in the file name, the archiver searches for the file in the current working directory.

Tagging an Archive With Version Information

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

Basic Version Information

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

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

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

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

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

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

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

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

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

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

User-Defined Version Information

You can provide any number of user-defined version values by supplying a text with those values. The text file can have any number of entries. Each line in the file begins with a name (a single token, for example, non-embedded white space), followed by a space and then the value associated with that name. As an example, consider the file foo.txt:

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

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

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

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

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

Printing Version Information

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

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

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

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

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

Removing Version Information From an Archive

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

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

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

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

Checking Version Number

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

Adding Text to Version Information

Additional text can be added to the end of the version information.

For example,

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

Archiver Command-Line Reference

The archiver processes object files into a library file with a .DLB extension, which is the default extension for library files. The archiver can also append, delete, extract, or replace member files in a library, as well as list them to stdout. This section provides the following reference information on the archiver command line and linking.

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

elfar Command Syntax

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

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

Table 6-2 describes each switch.

Example

```
elfar -v -c my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

This command line runs the archiver as follows:

-v – Outputs status information

-c my_lib.dlb – Creates a library file named my_lib.dlb

fft.doj sin.doj cos.doj tan.doj – Places these object files in the library file

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

Symbol Encryption

When employing symbol encryption, use the following syntax.

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

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

Archiver Parameters and Switches

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

Table 6-2. Command-Line Options and Entries

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

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

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

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

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

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

Command-Line Constraints

The elfar command is subject to the following constraints.

- Select one action switch (a, c, d, e, p, r, s, M, or MM) only in a single command.
- Do not place the verbose operation switch, -v, in a position where it can be mistaken for an object file. It may not follow the *lib_file* during an append or create operation.
- The file include switch, -i, must immediately precede the name of the file to be included. The archiver's -i switch enters a list of members from a text file instead of listing each member on the command line.
- Use the library file name first, following the switches. The -i and -v switches are not operational switches, and can appear later.
- When using the archiver's -p switch, it is not necessary to identify members on the command line.
- Enclose file names containing white space or colons within straight quotes.
- Append the appropriate file extension to each file. The archiver assumes nothing, and does not do it for you.
- Wildcard options are supported with the use of the wildcard character "*").

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

Archiver Symbol Name Encryption

Symbol name encryption protects intellectual property contained in an archive (.DLB) library that might be revealed when using meaningful symbol names. Code and test a library with meaningful symbol names, and then use archive library encryption on the fully tested library to disguise the names.

Source file names in the symbol tables of object files in the archive are not encrypted. The encryption algorithm is not reversible. Also, encryption does not guarantee a given symbol is encrypted the same way when different libraries, or different builds of the same library, are encrypted.

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

Command Syntax

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

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

where:

-s – Selects the encryption operation.

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

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

in_library_file – Specifies the name of the archive (.DLB) file to be encrypted. This file is not altered by the encryption process, unless in-archive is the same as out-archive.

exclude-file – Specifies the name of a text file containing a list of symbols not to be encrypted. The symbols are listed one or more to a line, separated by white space.

type-letter – The initial letter of type-letter provides the initial letter of all encrypted symbols.

Encryption Constraints

All local symbols can be encrypted, unless they are correlated with a symbol having external binding that should not be encrypted. Symbols with external binding can be encrypted when they are used only within the library in which they are defined. Symbols with external binding that are not defined in the library (or are defined in the library and referred to outside of the library) should not be encrypted. Symbols that should not be encrypted must be placed in a text file, and the name of that file given as the exclude-file command-line argument.

Some symbol names have a prefix or suffix that has special meaning. The debugger does not show a symbol starting with "." (period), and a symbol starting with "." and ending with ".end" is correlated with another symbol. For example, ".bar" would not be shown by the debugger, and "._foo.end" would correlated with the symbol "_foo" appearing in the same object file. The encryption process encrypts only the part of the symbol after any initial "." and before any final ".end". This part is called the root of the symbol name. Since only the root is encrypted, a name with a prefix or suffix having special meaning retains that special meaning after encryption.

The encryption process ensures that a symbol with external binding is encrypted the same way in all object files contained in the library. This process also ensures that correlated symbols within an object file are encrypted the same way, so they remain correlated.

The names listed in the `exclude-file` are interpreted as root names. Thus, "`_foo`" in the `exclude-file` prevents the encryption of the symbol names "`_foo`", "`._foo`", "`_foo.end`", and "`._foo.end`".

The `type-letter` argument, which provides the first letter of the encrypted part of a symbol name, ensures that the encrypted names in different archive libraries can be made distinct. If different libraries are encrypted with the same `type-letter` argument, unrelated external symbols of the same length may be encrypted identically.

7 MEMORY INITIALIZER

VisualDSP++ 4.0 supports the Memory Initializer (`MemInit.exe`) tool. The Memory Initializer's main function is to modify executable files (`.DXE` files) so that the programs are self-initializing. It does this by converting the program's RAM-based contents into an initialization stream which it embeds into the executable file.

This chapter provides:

Memory Initializer Overview

The Memory Initializer may be used with processor systems where the RAM memory needs to be initialized with the code and data stored in the ROM memory before the execution of the application code begins. This is generally true for a processor system running in NO-BOOT mode.

The initialization stream generated by the Memory Initializer is consumed by a dedicated Run-Time Library (RTL) routine. Following a system reset, the RTL routine searches the initialization stream and initializes the processor's RAM memory with the data in the initialization stream before the call to `main()`, the starting point of the application code.

In creating the initialization stream, the Memory Initializer can, in most cases, effectively reduce the overall size of an executable file by combining contiguous, identical initialization into a single block. For example, a large zero-initialized array in an executable file can be compressed to a single small data block by the Memory Initializer.

In addition to a primary executable file (`.dxe`), the Memory Initializer accepts one or more additional executable files called "callback" executable files, and includes their data and instructions in the initialization stream. The RTL routine is able to call and execute them before conducting the process of the memory initialization for the primary application. This allows you to perform memory configuration and any other set-up functions that must occur before the code and data are extracted from ROM memory.

Basic Operation of Memory Initializer

This section describes the basic operations of the Memory Initializer, its input and output files, as well as basic initialization stream generated by the Memory Initializer.

Input and Output Files

The Memory Initializer takes an executable file (.DXE) as a primary input file and augments it by adding an initialization stream. The enhanced executable file is written as the output file.

Processing the Primary Input Executable File
After opening an input primary executable file, the Memory Initializer looks for sections, marked with the Initialization Flag in their section headers or specified from the command line, and extracts the data and instructions from them to make the primary initialization stream.

By default, the stream is saved in the dedicated memory section called ".meminit" in the output file. For the sections from which the Memory Initializer extracts no data, the Memory Initializer simply copies them from the input file to the output file. Sections that are processed by the Memory Initializer to form the initialization stream are not needed in the output executable file, as their contents will be regenerated at run time when the initialization stream is processed. Therefore, by default such sections are not copied to the output file in order to reduce the size of the executable file.

Processing Callback Input Executable Files

In addition to a primary input executable file, the Memory Initializer optionally accepts a number of individually-built "callback" executable files specified with the -init switch (on page 7-15).

The Memory Initializer sequentially processes the callback executable files, one at a time. After opening an input callback executable file, the Memory Initializer looks for all of the sections marked with the Initialization Flag and PROGBITS qualifier (it indicates the section contains either the instructions or data or both), and extracts the data and instructions from them make a callback initialization stream. When this stream is built up, all the callback .DXE files are processed in the order they are specified on the command line.

The Memory Initializer continues making a callback initialization stream from each of the callback executable files and pre-pending it to the primary initialization stream in the same sequence the callback executable files appear in the command line until the last callback executable file is processed.

When processing a callback executable file, the Memory Initializer extracts all the code and data from it to make up the callback initialization stream regardless of the Memory Initializer command-line switches used only for the primary input file. Those switches are:

- "-BeginInit Initsymbol" on page 7-14
- "-Init Initcode.dxe" on page 7-15
- "-NoAuto" on page 7-16
- "-NoErase" on page 7-16
- "-Section Sectionname" on page 7-17
- "-SectDestination Sectionname"on page 7-17

This ensures the integrity of the code and data from each callback executable file in the callback initialization stream – the code can be executed independently and successfully regardless of the Memory Initializer command-line switches.

By taking multiple input files, the Memory Initializer supports those systems which have to run a number of independent service applications before starting the primary application.

Initialization Stream Structure

An initialization stream made from the Memory Initializer has three major portions:

- The header of the initialization stream which holds the basic information for the Run-Time Library (RTL) routine such as the number of data blocks in the initialization stream
- The callback executable file which itself may have a number of the sub-portions with each containing a piece of the callback executable
- The initialization data and code from the primary application

Figure 7-1 shows the basic structure of the initialization stream:

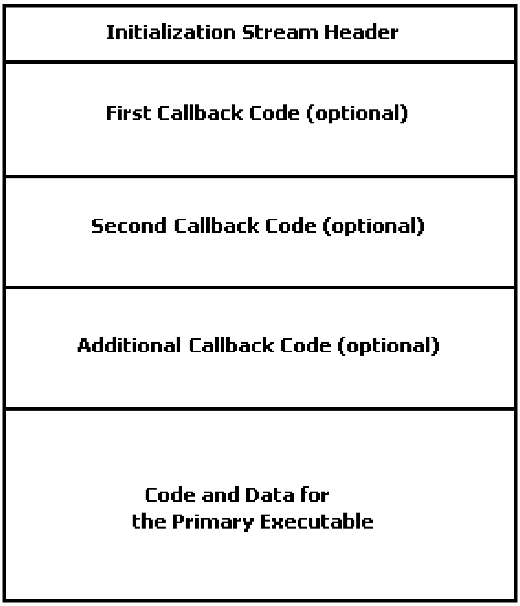


Figure 7-1. Memory Initializer Basic Initialization Stream Structure

Run-Time Library Routine Basic Operation

A Run-Time Library (RTL) routine performs the memory initialization with the initialization stream created by the Memory Initializer during runtime. It can be a dedicated RTL routine or user-provided routine called `_mi_initialize` (from the assembly code).

For more information on the definition of the initialization stream, see EE-239 for Blackfin processors

Following a system reset, the RTL routine is invoked by the application's start-up code.

The RTL routine:

1. searches for the initialization stream
2. digests the stream header
3. for each callback executable specified, the RTL routine first copies "callback" code into RAM, and then executes it. It is done piece by piece and continue until execution is complete
4. brings the code and data from the primary executable file into the processor's memory

Once each callback executable has been executed, it is no longer needed in RAM; it may be overwritten by future callback executables or by the code or data spaces of the primary executable. After all the "callback" codes are executed, the RTL routine starts to initialize the processor's memory with the initialization stream created from the primary input executable file, and overwrites the memory spaces previously initialized with the "callback" codes. After that, the RTL routine returns execution to the start-up header, and the application proceeds as normal.

If there are no callback executables to be executed, the RTL routine immediately starts the process of initializing memory for the primary application.

Using the Memory Initializer

There are several reasons why it may be beneficial to use the Memory Initializer:

- The system needs to initialize the RAM memory from the data stored in ROM.
- It is desirable to reduce the overall size of the executable.
- Initialization executable files need to run to configure the system, before the primary application starts.

If it is decided to use the Memory Initializer, the preparation starts from the linker description file (.LDF) and the source files of the project.

Preparing the Linker Description File (.LDF)

If a section is to be processed by the Memory Initializer in order to create the initialization stream, the section must be marked in the .LDF file to indicate the kind of initialization required. This is done using initialization qualifiers. The qualifiers that are used here are ZERO_INIT and RUNTIME_INIT. The sections marked with ZERO_INIT may contain only "zero-initialized" data, and the sections marked with RUNTIME_INIT may contain the data with any initialization values.

Refer to the SECTIONS descrip[tion in the *VisualDSP++ 4.0 Linker and Utilities Manual* for detailed information on these qualifiers.

The following example shows how to use the ZERO_INIT and RUNTIME_INIT qualifiers in a .LDF file to set up the section type.

```
my_zero_section ZERO_INIT
{
   INPUT_SECTION_ALIGN(4)
   INPUT_SECTIONS( $OBJECTS(my_zero_section)
                         $LIBRARIES(my_zero_section))
} >MEM_L1_DATA_A
```

```
my_data_section RUNTIME_INIT
{
   INPUT_SECTION_ALIGN(4)
   INPUT_SECTIONS( $OBJECTS(my_data_section) )
}>MEM_L1_DATA_A
```

The section `my_zero_section` is intended to hold all the zero-initialized data, and the section `my_data_section` is to hold any other initialized data. After the program is first linked, the sections in the `.DXE` file have flags set according to the qualifiers in the `.LDF` file. Then the Memory Initializer runs and processes the DXE sections according to those flags, and produces a modified output `.DXE` file.

The Memory Initializer is able to identify the DXE sections with the distinct Initialization Flag and extract the data from them to make an initialization stream. Any number of sections can be set as either `ZERO_INIT` or `RUNTIME_INIT` type in an `.LDF` file.

Please note that there are two memory sections specified in a default `.LDF` file, which also serve the Memory Initializer: `bsz_init` and `.meminit`. The `bsz_init` section is used to hold the pointer generated by the Memory Initializer, which points to the start address of the initialization stream, while the section `.meminit` is used to hold the actual initialization stream generated by the Memory Initializer. Although other sections may be selected as alternatives (using the appropriate command-line switches), it is not recommended to do so.

Preparing the Source Files

The sections marked with the `ZERO_INIT` and `RUNTIME_INIT` qualifiers need to be initialized with the proper values in the source files before being compiled. The following example shows one way to initialize a section.

```
#include <stdio.h>
#pragma section("my_data_section", RUNTIME_INIT)
unsigned int A [ 100 ] =
```

```
{ 0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,
0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd,0xaabbccdd };

#pragma section("my_zero_section", ZERO_INIT)
unsigned int B [ 128 ];

int void main()
{
   int i;
   int not_init = 0, not_zero = 0;

   for (i = 0; i < 100; i++)
      if ( A [ i ] != 0xaabbccdd )
      not_init++;

   for (i = 0; i < 128; i++)
      if ( B [ i ] != 0 )
      not_zero++;

   printf ("A[]: %d elements not initialized\n", not_init);
   printf ("B[]: %d elements not zeroed\n", not_zero);
   return 0;
}
```

Invoking the Memory Initializer

There are several ways to invoke the Memory Initializer, either from the IDDE or from a command line.

Invoking Memory Initializer from the VisualDSP++ IDDE

Use the **Project** menu in the VisualDSP++ main screen to select **Project -> Project Options -> Link** (see Figure 7-2) The **Link** option is displayed. Type `-meminit` in the **Additional options** field and then click **OK**. When the project is built, the linker calls the Memory Initializer.

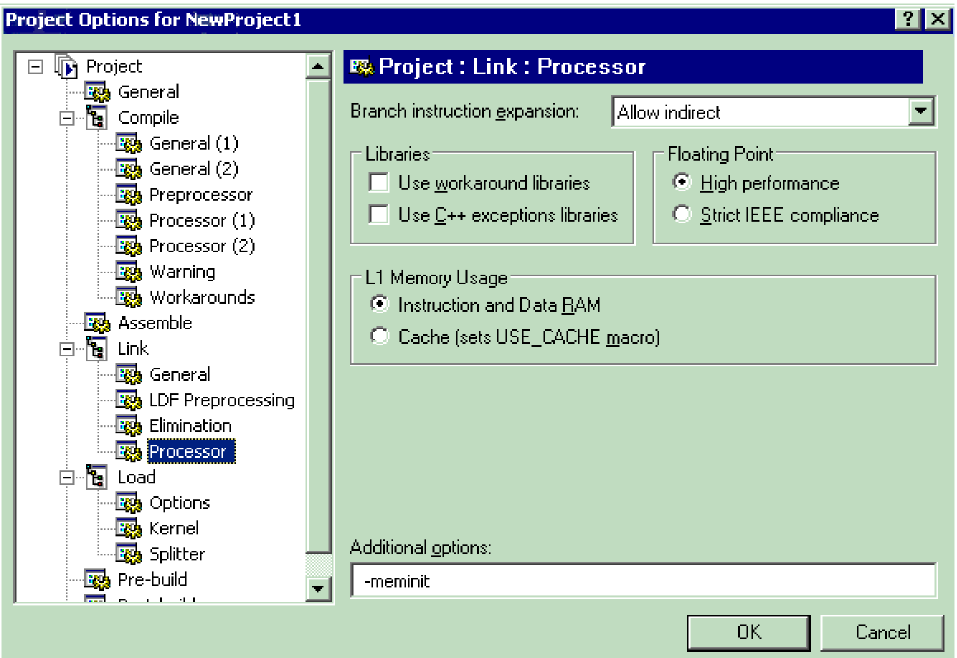


Figure 7-2. Invoking the Memory Initializer from the VisualDSP++ IDDE

Invoking Memory Initializer from Command Line

The simplest command line to invoke the Memory Initializer from the command line is:

```
meminit.exe input.dxe -o output.dxe
```

The Memory Initializer identifies all the sections with Initialization Flags in the input file, produces an initialization stream, and places it in the output file. The Memory Initializer command-line switches are listed in Table 7-1.

Invoking Memory Initializer from Linker's Command Line

The simplest way to invoke the Memory Initializer from the linker's command line is to use the linker's `-meminit` switch. The linker also provides the `-flag-meminit` switch that passes each comma-separated option to the Memory Initializer. For example,

```
linker -proc ADSP-BF535 -meminit input.dxe -o output.dxe
```

Invoking Memory Initializer from Compiler's Command Line

The simplest command line to invoke the Memory Initializer from the compiler's command line is (for example, for Blackfin processors):

```
ccblkfn -proc ADSP-BF535 -mem input.dxe -o output.dxe
```

Invoking Memory Initializer with Callback Executables

Invocation of Memory Initializer from the IDDE
Select **Project -> Project Options -> Link**. The **Link** option is displayed. Use the **Additional options** field to process callback executable files.

For example, if you have two callback executable files, `callback1.dxe` and `callback2.dxe`, and you wish to pass them to the Memory Initializer, you can enter them in the **Additional options** window as

```
-meminit -flag-meminit -Init callback1.dxe -Init callback2.dxe
```

then click **OK** (see Figure 7-3).

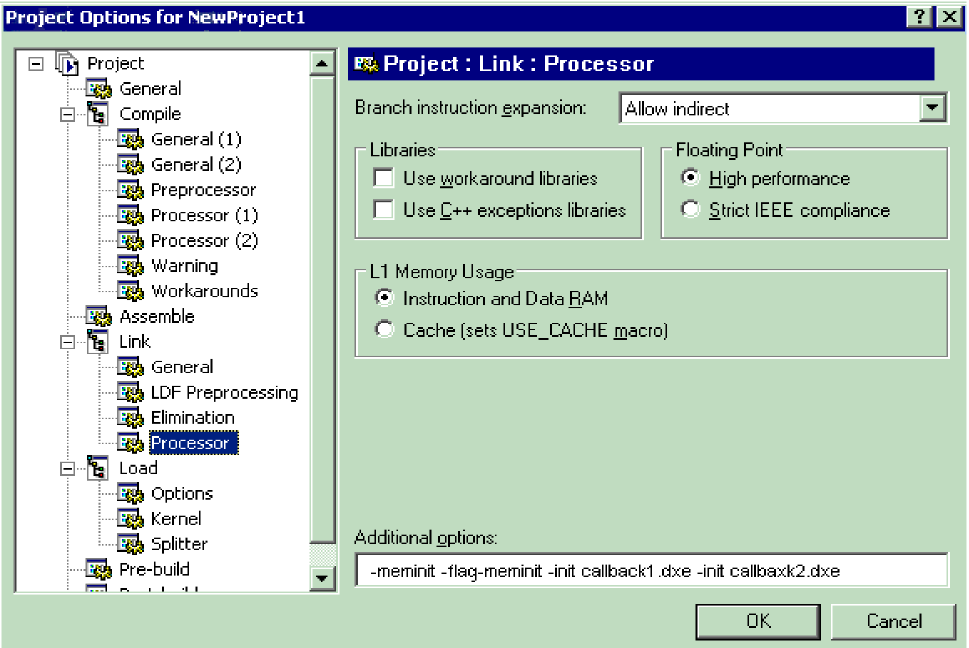


Figure 7-3. Invoking Callback Executable from the VisualDSP++ IDDE

Invocation of Memory Initializer from a Command Line

To directly invoke the Memory Initializer from a command line, use the `-Init` switch for each "callback" executable as shown below:

```
meminit.exe Input.dxe -o Output.dxe
            -Init Callback1.dxe -Init Callback2.dxe
```

Memory Initializer Command-Line Switches

The Memory Initializer provides a number of command line switches. They are not case sensitive. Table 7-1 provides a summary of the Memory Initializer switches. It is followed by a detailed description of each switch.

The listed switches are mostly optional. For a project in which the linker description file is well-defined (the `.meminit` and `bsz_init` memory sections are defined and the `ZERO_INIT` and `RUNTIME_INIT` qualifiers are set on the proper sections) and the sections are all initialized properly in the source files, most of these optional switches may not be necessarily required. By default, the Memory Initializer automatically handles everything that is needed to create an initialization stream.

Table 7-1. Command-Line Options and Entries

Item	Description
`-BeginInit` *`Initsymbol`* on page 7-14	Specifies a symbol name for a variable that holds a pointer pointing to the start address of an initialization stream.
`-h` on page 7-15	Displays the list of Memory Initializer switches.
`-IgnoreSection` *`Sectionname`* on page 7-15	Directs the Memory Initializer to NOT process a section selected in the primary input file
`-Init` *`Initcode.dxe`* on page 7-15	Specifies an executable file to be inserted into the initialization stream and executed as a callback.
`InputFile.dxe` on page 7-16	Specifies a primary input file.
`-NoAuto` on page 7-16	Directs the Memory Initializer to NOT process sections in the primary input file based on the section header flags (the section specified by a user as either `ZERO_INIT` and `RUNTIME_INIT` qualifier in the linker description file), but to only process sections specified on the command line through `-Sect SectionName`. This switch is optional.

Table 7-1. Command-Line Options and Entries (Cont'd)

Item	Description
-NoErase on page 7-16	Directs the Memory Initializer not to erase the data of the processed sections in the primary executable file.
-o *Outputfile.dxe* on page 7-17	Specifies an output file.
-Section *Sectionname* on page 7-17	Specifies a section from which the data will be extracted by the Memory Initializer. This switch can be repeated to specify a number of the sections from the specified input primary file.
-SectDestination *Sectionname* on page 7-17	Specifies a destination section (other than .meminit) to hold the initialization stream.
-v on page 7-18	(Verbose) Outputs status information as the MemInit utility processes files

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

-BeginInit Initsymbol

The -BeginInit *Initsymbol* switch is used to specify a symbol name for a variable that holds a pointer pointing to the start address of an initialization stream. The Memory Initializer updates this pointer with the start address of the initialization stream produced by the Memory Initializer.

If this switch is absent, the default symbol name "___inits" (it has three leading underscores, when called from assembly) is searched, which, by default, is in the bsz_init memory section. An error message is issued if this symbol cannot be found in the input primary file. If a symbol other than "___inits" is specified using this switch in a section other than "bsz_init", the symbol must **not** be in any of the sections specified via the -Section *Sectionname* switch (on page 7-17). It also must be able to hold a value which is no less than the maximum address value for the particular processor. The run-time library provides a default symbol of "___inits" for the Memory Initializer and, therefore, it is not necessary to use this switch in most cases. This switch has no effect on callback executable files specified using the "-Init Initcode.dxe" on page 7-15.

-h

The `-h[elp]` switch is used to display the list of Memory Initializer switches.

-IgnoreSection Sectionname

The `-IgnoreSection` *`Sectionname`* switch is used to specify a section that is **not** to be processed by the Memory Initializer. This switch can be repeated to specify a number of sections not to be processed in the primary input file. All the specified sections must exist in the primary input file.

The `-IgnoreSection` switch is optional. It is normally easier to remove a section's initialization qualifier (`ZERO_INIT` or `RUNTIME_INIT`) from the `.LDF` file than to use this switch. The switch does not affect a callback executable file specified using the `-Init` switch.

-Init Initcode.dxe

The `-Init` *`Initcode.dxe`* switch is used to specify an executable file to be inserted into the initialization stream and executed as a callback. Any number of executable files can be specified this way, and it is allowed to specify the same file name a number of times. The callback executable file must exist before the Memory Initializer is run. All the code and data from callback executable files are extracted to make up the initialization stream. This is an optional switch.

InputFile.dxe

The `InputFile.dxe` parameter is used to specify a primary input file. The Memory Initializer issues an error message if no primary input file is specified.

-NoAuto

The `-NoAuto` switch directs the Memory Initializer to **not** process sections in the primary input file based on the section header flags (the section specified as either `ZERO_INIT` and `RUNTIME_INIT` qualifier in the `.LDF` file), but to only process sections specified on the command line using the `-section` *`SectionName`* switch.

By default, the Memory Initializer automatically processes only the sections with `ZERO_INIT` and `RUNTIME_INIT` qualifiers in the `.LDF` file. This switch has no effect on the the code and data of callback executable files specified using the `-init` switch. All the code and data sections of a callback executable file are processed by the Memory Initializer regardless of this switch being used or not This switch is optional.

-NoErase

The `-NoErase` switch directs the Memory Initializer not to erase the data of the processed sections. By default, the Memory Initializer empties the sections from which the data are extracted to make up the initialization stream. This switch is valid for the primary input file only and has no effect on callback executable files. The Memory Initializer does not carry any sections of a callback executable file over to the output file, nor erase any sections, but only extracts the code and data from it to form the initialization stream.

-o Outputfile.dxe

The `-o` *`Outputfile.dxe`* switch is used to specify an output file. If this switch is absent, the Memory Initializer makes up an output file name from the root of the input file name. For example, the output file name is created as `InputFile1.dxe` if the input file name is `InputFile.dxe`. This switch is optional.

-Section Sectionname

The `-Section` *`Sectionname`* switch is used to specify a section from which the data is extracted by the Memory Initializer. This switch can be repeated to specify a number of the sections from the specified input primary file. All the section specified must exist in the specified input primary file. Note that the section name specified via the `-SectDestination` or `-IgnoreSection` switches cannot be used with the `-Section` switch.

It is not necessary to use this switch to specify any sections which already have the `ZERO_INIT` or `RUNTIME_INIT` qualifiers in the linker description file (`.LDF`), as the Memory Initializer processes such sections automatically. Using initialization qualifiers in the `.LDF` file is usually the simpler and recommended method. The `-Section` *`SectionName`* switch has no effect on callback executable files specified via the `-Init` switch. Therefore, do not use this switch to specify any sections in callback executable files.

-SectDestination Sectionname

The `-SectDestination` *`Sectionname`* switch is used to specify a destination section to hold the initialization stream. The section name must already exist in the input primary file. The Memory Initializer appends the processed initialization stream into this section. The default value for the `SectionName` (if the switch is not provided on the command line) is "`.meminit`".

The memory type (PM or DM) of the specified section (or `.meminit`) has no impact to the processed data, though generally, the memory type of `.meminit` should be PM if the processor supports the PM/DM architecture.

The Memory Initializer generates an error if the section is not found in the file. If this switch is not present, the default name is "`.meminit`". An error is also produced if the section is not large enough to hold the generated initialization stream.

> Please note that this switch does not have to be used if `.meminit` is defined in the `.LDF` file and the linker allocates the memory space for that default section to hold the initialization stream. Instead of using this switch, you could define the `.meminit` memory section in the `.LDF` file for the default initialization stream destination section. Also note that the linker expands the `.meminit` memory section to fill the remaining space.

This switch has no effect on a callback executable file. Therefore, do not use this switch to specify a section existing only in a callback executable file. This switch is optional.

-v

The `-v` or `-verbose` (verbose) switch directs the Memory Initializer to output status information as it processes files.

A FILE FORMATS

The VisualDSP++ development tools support many file formats. In some cases, several file formats for each development tool are supported. This appendix describes file formats that are prepared as input for the tools and points out the features of files produced by the tools.

This appendix discusses three types of file formats:

- “Source Files” on page A-2
- “Build Files” on page A-5
- “Debugger Files” on page A-9

Most of the development tools use industry-standard file formats. Sources that describe these formats appear in “Format References” on page A-10.

Source Files

This section describes these input file formats:

- "C/C++ Source Files" on page A-2
- "Assembly Source Files (.ASM)" on page A-3
- "Assembly Initialization Data Files (.DAT)" on page A-3
- "Header Files (.H)" on page A-4
- "Linker Description Files (.LDF)" on page A-4
- "Linker Command-Line Files (.TXT)" on page A-5

C/C++ Source Files

These are text files (with extensions such as `.C`, `.CPP`, `.CXX`, and so on) containing C/C++ code, compiler directives, possibly a mixture of assembly code and directives, and (typically) preprocessor commands.

Several "dialects" of C code are supported: pure (portable) ANSI C, and at least two subtypes[1] of ANSI C with ADI extensions. These extensions include memory type designations for certain data objects, and segment directives used by the linker to structure and place executable files.

For information on using the C/C++ compiler and associated tools, as well as a definition of ADI extensions to ANSI C, see the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual* for appropriate target architectures.

[1] With and without built-in function support; a minimal differentiator. There are others.

Assembly Source Files (.ASM)

Assembly source files are text files containing assembly instructions, assembler directives, and (optionally) preprocessor commands. For information on assembly instructions, see your processor's Programming Reference.

The 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++ 4.0 Assembler and Preprocessor Manual*.

Assembly Initialization Data Files (.DAT)

Assembly initialization data (`.DAT`) files are text files that contain fixed- or floating-point data. These files provide the initialization data for an assembler `.VAR` directive or serve in other tool operations.

When a `.VAR` directive uses a `.DAT` file for data initialization, the assembler reads the data file and initializes the buffer in the output object (`.DOJ`) file. Data files have one data value per line and may have any number of lines.

The `.DAT` extension is explanatory or mnemonic. A directive to `#include <file>` can take any file name (or extension) as an argument.

Fixed-point values (integers) in data files may be signed, and they may be decimal-, hexadecimal-, octal-, or binary-base values. The assembler uses the prefix conventions in Table A-1 to distinguish between numeric formats.

For all numeric bases, the assembler uses 16-bit words for data storage; 24-bit data is for the program code only. The largest word in the buffer determines the size for all words in the buffer. If there is some 8-bit data in a 16-bit wide buffer, the assembler loads the equivalent 8-bit value into the most significant eight bits in the 8-bit memory location and zero-fills the lower eight bits.

Table A-1. Numeric Formats

Convention	Description
`0x`*number* `H#`*number* `h#`*number*	Hexadecimal number
number `D#`*number* `d#`*number*	Decimal number
`B#`*number* `b#`*number*	Binary number.
`O#`*number* `o#`*number*	Octal number.

Header Files (.H)

Header files are ASCII text files that contain macros or other preprocessor commands that the preprocessor substitutes into source files. For information on macros or other preprocessor commands, see the *VisualDSP++ 4.0 C/C++Compiler and Library Manual* for appropriate target architectures. For information on the assembler and preprocessor, see the *VisualDSP++ 4.0 Assembler and Preprocessor Manual*.

Linker Description Files (.LDF)

Linker Description Files are ASCII text files that contain commands for the linker in the linker's scripting language. For information on this scripting language, see "LDF Commands" on page 3-29.

Linker Command-Line Files (.TXT)

Linker command-line files are ASCII text files that contain command-line input for the linker. For more information on the linker command line, see "Linker Command-Line Reference" on page 2-47.

Build Files

Build files are produced by the VisualDSP++ development tools when building a project. This section describes these build file formats:

- "Assembler Object Files (.DOJ)" on page A-5
- "Library Files (.DLB)" on page A-5
- "Linker Output Files (.DXE, .SM, and .OVL)" on page A-6
- "Memory Map Files (.XML)" on page A-6
- "Loader Output Files in Intel Hex-32 Format (.LDR)" on page A-6
- "Splitter Output Files in ASCII Format (.LDR)" on page A-8

Assembler Object Files (.DOJ)

Assembler output object (`.DOJ`) files are in binary, executable and linkable file (ELF) format. Object files contain relocatable code and debugging information for a DSP program's memory segments. The linker processes object files into an executable (`.DXE`) file. For information on the object file's ELF format, see "Format References" on page A-10.

Library Files (.DLB)

Library files, the archiver's output, are in binary, executable and linkable file (ELF) format. Library files (called archive files in previous software releases) contain one or more object files (archive elements).

The linker searches through library files for library members used by the code. For information on the ELF format used for executable files, refer to "Format References" on page A-10.

Linker Output Files (.DXE, .SM, and .OVL)

The linker's output files are in binary, executable and linkable file (ELF) format. These executable files contain program code and debugging information. The linker fully resolves addresses in executable files. For information on the ELF format used for executable files, see the TIS Committee texts cited in "Format References" on page A-10.

ⓘ The archiver automatically converts legacy input objects from COFF to ELF format.

Memory Map Files (.XML)

The linker can output memory map files that contain memory and symbol information for your executable file(s). The map file contains a summary of memory defined with `MEMORY{}` commands in the `.LDF` file, and provides a list of the absolute addresses of all symbols. Memory map files are available *only* in `.xml` format.

Loader Output Files in Intel Hex-32 Format (.LDR)

The loader can output Intel hex-32 format (`.LDR`) files. These files support 8-bit-wide PROMs. The files are used with an industry-standard PROM programmer to program memory devices for a hardware system. One file contains data for the whole series of memory chips to be programmed.

The following example shows how the Intel hex-32 format appears in the loader's output file. Each line in the Intel hex-32 file contains an extended linear address record, a data record, or an end-of-file record.

`:020000040000FA`	Extended linear address record

:0402100000FE03F0F9	Data record
:00000001FF	End-of-file record

Extended linear address records are used because data records have a 4-character (16-bit) address field, but in many cases, the required PROM size is greater than or equal to 0xFFFF bytes. Extended linear address records specify bits 16-31 for the data records that follow.

Table A-2 shows an example of an extended linear address record.

Table A-2. Example – Extended Linear Address Record

Field	Purpose
:020000040000FA	Example record
:	Start character
02	Byte count (always 02)
0000	Address (always 0000)
04	Record type
0000	Offset address
FA	Checksum

Table A-3 shows the organization of an example data record, and Table A-4 shows an end-of-file record.

Table A-3. Example – Data Record

Field	Purpose
:0402100000FE03F0F9	Example record
:	Start character
04	Byte count of this record
0210	Address

Table A-3. Example – Data Record (Cont'd)

Field	Purpose
00	Record type
00	First data byte
F0	Last data byte
F9	Checksum

Table A-4. Example – End-of-File Record

Field	Purpose
:00000001FF	End-of-file record
:	Start character
00	Byte count (zero for this record)
0000	Address of first byte
01	Record type
FF	Checksum

For more information, refer to the *VisualDSP++ 4.0 Loader Manual.*

Splitter Output Files in ASCII Format (.LDR)

When the loader is invoked as a splitter, its output can be an ASCII format file. ASCII format files are text representations of ROM memory images that you can use in post-processing. For more information, refer to no-boot mode information in the *VisualDSP++ 4.0 Loader Manual.*.

Debugger Files

Debugger files provide input to the debugger to define simulation or emulation support of your program. The debugger supports all the executable file types produced by the linker (.DXE, .SM, .OVL). To simulate I/O, the debugger also supports the assembler's data file (.DAT) format and the loader's loadable file (.LDR) formats.

The standard hexadecimal format for a SPORT data file is one integer value per line. Hexadecimal numbers do not require a 0x prefix. A value can have any number of digits, but is read into the SPORT register as:

- The hexadecimal number which is converted to binary
- The number of binary bits read which matches the word size set for the SPORT register, which starts reading from the LSB. The SPORT register then fills with zero values shorter than the word size or conversely truncates bits beyond the word size on the MSB end.

Example
In this example, a SPORT register is set for 20-bit words and the data file contains hexadecimal numbers. The simulator converts the HEX numbers to binary and then fills or truncates to match the SPORT word size. In Table A-5, the A5A5 number is filled and 123456 is truncated.

Table A-5. SPORT Data File Example

Hex Number	Binary Number	Truncated/Filled
A5A5A	1010 0101 1010 0101 1010	1010 0101 1010 0101 1010
FFFF1	1111 1111 1111 1111 0001	1111 1111 1111 1111 0001
A5A5	1010 0101 1010 0101	0000 1010 0101 1010 0101
5A5A5	0101 1010 0101 1010 0101	0101 1010 0101 1010 0101
11111	0001 0001 0001 0001 0001	0001 0001 0001 0001 0001
123456	0001 0010 0011 0100 0101 0110	0010 0011 0100 0101 0110

Format References

The following texts define industry-standard file formats supported by VisualDSP++.

- Gircys, G.R. (1988) *Understanding and Using COFF* by O'Reilly & Associates, Newton, MA
- (1993) *Executable and Linkable Format (ELF) V1.1* from the Portable Formats Specification V1.1, Tools Interface Standards (TIS) Committee

  Go to: `http://developer.intel.com/vtune/tis.htm`
- (1993) *Debugging Information Format (DWARF) V1.1* from the Portable Formats Specification V1.1, UNIX International, Inc.

  Go to: `http://developer.intel.com/vtune/tis.htm`

B UTILITIES

The VisualDSP++ development software includes several utilities, some of which run from a command line only. This appendix describes the ELF file dumper utility.

elfdump – ELF File Dumper

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

The `elfdump` utility is often used with the archiver (`elfar.exe`). Refer to "Disassembling a Library Member" on page B-3 for details.

Syntax: `elfdump [switches] [objectfile]`

Table B-1 shows switches used with the elfdump command.

Table B-1. ELF File Dumper Command-Line Switches

Switch	Description
`-c`	Stabs to mdebug conversion
`-fh`	Prints the file header
`-arsym`	Prints the library symbol table
`-arall`	Prints every library member
`-help`	Prints the list of elfdump switches to stdout
`-ph`	Prints the program header table
`-s`	Prints contents as a list of disassembled machine instructions. Also prints labels.

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

Switch	Description
-sh	Prints the section header table. This switch is the default when no options are specified.
-notes	Prints note segment(s)
-n *name*	Prints contents of the named section(s). The name may be a simple 'glob'-style pattern, using "?" and "*" as wildcard characters. Each section's name and type determines its output format, unless overridden by a modifier.
-i x0[-x1]	Prints contents of sections numbered x0 through x1, where x0 and x1 are decimal integers, and x1 defaults to x0 if omitted. Formatting rules are the same as for the -n switch.
-all	Prints everything. This is the same as -fh -ph -sh -notes -n '*'.
-ost	Omits string table sections
-v	Prints version information
objectfile	Specifies the file whose contents are to be printed. It can be a core file, executable, shared library, or relocatable object file. If the name is in the form A(B), A is assumed to be a library and B is an ELF member of the library. B can be a pattern similar to the one accepted by -n. The -n and -i options can have a modifier letter after the main option character to force section contents to be formatted as: a Dumps contents in hex and ASCII, 16 bytes per line. x Dumps contents in hex, 32 bytes per line. xN Dumps contents in hex, N bytes per group (default is N = 4). t Dumps contents in hex, N bytes per line, where N is the section's table entry size. If N is not in the range 1 to 32, 32 is used. hN Dumps contents in hex, N bytes per group. HN Dumps contents in hex, (MSB first order), N bytes per group. i Prints contents as list of disassembled machine instructions.

Disassembling a Library Member

The `elfar` and `elfdump` utilities are more effective when their capabilities are combined. One application of these utilities is for disassembling a library member and converting it to source code. Use this technique when the source of a particularly useful routine is missing and is available only as a library routine.

For information about `elfar`, refer to "Archiver" on page 6-1.

The following procedure lists the objects in a library, extracts an object, and converts the object to a listing file. The first archiver command line lists the objects in the library and writes the output to a text file.

```
elfar -p libc.dlb > libc.txt
```

> This command assumes the current directory is:
> `C:\Program Files\Analog Devices\VisualDSP\21xxx\lib`

Open the text file, scroll through it, and locate the object file you need. Then, use the following archiver command to extract the object from the library.

```
elfar -e libc.dlb fir.doj
```

To convert the object file to an assembly listing file with labels, use the following `elfdump` command line.

```
elfdump -ns * fir.doj > fir.asm
```

The output file is practically source code. Just remove the line numbers and opcodes.

Disassembly yields a listing file with symbols. Assembly source with symbols can be useful if you are familiar with the code and hopefully have some documentation on what the code does. If the symbols are stripped during linking, the dumped file contains no symbols.

Disassembling a third-party's library may violate the license for the third-party software. Ensure there are no copyright or license issues with the code's owner before using this disassembly technique.

Dumping Overlay Library Files

Use the `elfar` and `elfdump` utilities to extract and view the contents of overlay library (`.OVL`) files.

For example, the following command lists (prints) the contents (library members) of the `CLONE2.OVL` library file.

```
elfar -p CLONE2.OVL
```

The following command allows you to view one of the library members (`CLONE2.ELF`).

```
elfdump -all CLONE2.OVL(CLONE2.elf)
```

The following commands extract `CLONE2.ELF` and print its contents.

```
elfar -e CLONE2.ovl
elfdump -all CLONE2.elf
```

Switches for these commands are case sensitive.

C LDF PROGRAMMING EXAMPLES FOR TIGERSHARC PROCESSORS

This appendix provides several typical LDFs used with TigerSHARC processors. As you modify these examples, refer to the syntax descriptions in "LDF Commands" on page 3-29.

This appendix provides the following examples.

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

```
<VisualDSP++ InstallationPath>\TS\Examples
```

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

```
<VisualDSP++ InstallationPath>\TS\ldf
```

Linking a 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 C-1 shows a single-processor LDF. Note the following commands in this LDF:

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

Listing C-1. Single-Processor System LDF Example

```
ARCHITECTURE(ADSP-TS201)

SEARCH_DIR ( $ADI_DSP\TS\lib )

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

   //  $ADI_DSP is a predefined linker macro that expands to
   //  the VisualDSP++ installation directory. Search for objects
   //  in directory TS\lib relative to the installation directory

$LIBS = libc.dlb;
```

```
   //  single.doj is a user-generated file.
   //  The linker will be invoked as follows:
   //     linker -T single-processor.ldf single.doj.
   //  $COMMAND_LINE_OBJECTS is a predefined linker macro.
   //  The linker expands this macro into the name(s) of the
   //  the object(s) (.doj files) and libraries (.dlb files)
   //  $COMMAND_LINE_OBJECTS = single.doj

   //  ts_header.doj is the standard initialization file for TSxxx

$OBJS = $COMMAND_LINE_OBJECTS, ts_hdr.doj;

   //  A linker project to generate a .DXE file

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

   MEMORY                // Processor-specific memory command
   { INCLUDE("TS201_memory.ldf") }

   SECTIONS              // Specify the output sections
   {
      INCLUDE( "TS201_sections.ldf" )
   }                     // end P0 sections
}                        // end P0 processor
```

Linking Large Uninitialized or Zero-InitializedVariables

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

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers, this action can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur long download times when used with an emulator. This situation also may occur when you boot from a loader file (because of the increased file size). Listing C-2 shows an example of assembly source code. Listing C-3 shows the use of the NO_INIT and ZERO_INIT sections to avoid initialization of a segment.

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

ⓘ Refer to "SECTIONS{}" on page 3-48 for more information on the NO_INIT and ZERO_INIT section qualifiers.

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

Listing C-2. Large Uninitialized Variables: Assembly Source

```
.SECTION/NO_INIT   sdram_area;      /* 1Mx32 SDRAM */
.VAR huge_buffer[0x1000000];
```

Listing C-3. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-TS201)
$OBJECTS = $COMMAND_LINE_OBJECTS;    // Libraries & objects from
                                     // the command line
MEMORY {
   SDRAM {
      TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)
         }   // end segment
      }      // end memory

PROCESSOR P0 {
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
           // NO_INIT section isn't written to the output file
   SECTION {
      sdram_output NO_INIT {
        INPUT_SECTIONS( $OBJECTS ( sdram_area ) )}
      >mem_sdram;
   SECTION {
      zero_sdram_output ZERO_INIT {
         INPUT_SECTIONS ( $OBJECTS ( zero_sdram_area ) )}
      >mem_sdram;
      }   // end section
   }      // end processor P0
```

Linking an ADSP-TS101 MP Shared Memory System

When linking executable files for a multiprocessor system using shared memory, the .LDF file describes the multiprocessor memory offsets, shared memory, each processor's memory, and places code for each processor. Note the following in the example .LDF file in Listing C-4 on page C-6:

- The ARCHITECTURE() command defines the processor type, which can only be one type.
- The MPMEMORY{} command defines each processor's offset within multiprocessor memory.
- The SHARED_MEMORY{} command identifies the output for the shared memory items.
- The MEMORY{} command defines memory for the processors.
- The PROCESSOR{} and SECTIONS{} commands define each processor and place program sections using memory definitions for each processor's output file.
- The LINK_AGAINST() commands resolve symbols within multiprocessor memory.

Listing C-4. LDF for a Multiprocessor System With Shared Memory

```
ARCHITECTURE(ADSP-TS101)
SEARCH_DIR( $ADI_DSP\TS\lib )

// Allocate multiprocessor memory space with the MPMEMORY{}
// command. The values represent an "addend" that the linker
// uses to resolve undefined symbols in one DXE file to symbols
// defined in another DXE. The addend is added to each defined
//  symbol's value.
//  For example, PROCESSOR project PSH0 contains the undefined
```

```
//  symbol “buffer”, and PROCESSOR project PSH1 defines “buffer”
//  at address 0x22000. The linker will “fix up” the reference
//  to “buffer” in PSH0’s code to address:
//  0x22000 + MPMEMORY(PSH1) = 0x22000 + 0x2400000= 0x2422000

MPMEMORY
{
   PSH0 { START (0x2000000) }
   PSH1 { START (0x2400000) }
}
MEMORY
{
      // This is used for all processors. Alternatively,
      // a PROCESSOR can describe its own memory.

      /* Internal memory blocks are 0x10000 (64K bytes) */

M0Code  {TYPE(RAM) START(0x00000000) END(0x0000FFFF) WIDTH(32)}
M1Data  {TYPE(RAM) START(0x00080000) END(0x0008BFFF) WIDTH(32)}
M1Stack {TYPE(RAM) START(0x0008C000) END(0x0008FFFF) WIDTH(32)}
M2Data  {TYPE(RAM) START(0x00100000) END(0x0010BFFF) WIDTH(32)}
M2Heap  {TYPE(RAM) START(0x0010C000) END(0x0010C7FF) WIDTH(32)}
M2Stack {TYPE(RAM) START(0x0010C800) END(0x0010FFFF) WIDTH(32)}
SDRAM   {TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)}
MS0     {TYPE(RAM) START(0x08000000) END(0x0BFFFFFF) WIDTH(32)}
MS1     {TYPE(RAM) START(0x0C000000) END(0x0FFFFFFF) WIDTH(32)}
}

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

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

$SHARED_OBJECTS = shared.doj;

   //  The output name of this shared object is used
```

```
   //  subsequently in the LINK_AGAINST command
   //  of the PROCESSOR projects.

OUTPUT(shared.sm)

    //  shared.c has data declarations only. There is no need
    //  to specify any output other than “data2”.

SECTIONS {
   data2 {
         INPUT_SECTIONS ($SHARED_OBJECTS(data2))
         } >M1Data
   }    // end shared sections
}       // end shared memory

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

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

SECTIONS {
   // places code (instructions) in M0 (internal memory)
      code{
          FILL(0xb3c00000)
          INPUT_SECTION_ALIGN(4)
          INPUT_SECTIONS(
              $PSH0_OBJECTS(program) $LIBRARIES(program)
              )
          } >M0Code

   // place data in M2 (internal memory)
      data1{
           INPUT_SECTIONS(
               $PSH0_OBJECTS(data1) $LIBRARIES(data1)
               )
           } >M2Data
```

```
// place data in M1 (internal memory)
   data2{
        INPUT_SECTIONS(
            $PSHO_OBJECTS(data2) $LIBRARIES(data2)
            )
        } >M1Data

// place C RTL initializers in M2 (internal memory)
   ctor{
         INPUT_SECTIONS( $LIBRARIES(ctor0))
         INPUT_SECTIONS( $LIBRARIES(ctor1))
         INPUT_SECTIONS( $LIBRARIES(ctor2))
         INPUT_SECTIONS( $LIBRARIES(ctor3))
         INPUT_SECTIONS( $LIBRARIES(ctor))
         } >M2Data

// place C RTL heap table in M2 (internal memory)
   heaptab{
         INPUT_SECTIONS(
             $PSHO_OBJECTS(heaptab) $LIBRARIES(heaptab)
             )
         } >M2Data

// place C RTL JALU stack in M2 (internal memory)
   jstackseg{
         ldf_jstack_limit = .;
         ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
         } >M2Stack

// place C RTL KALU stack in M1 (internal memory)
   kstackseg{
         ldf_kstack_limit = .;
         ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
         } >M1Stack

// place C RTL heap in M2 (internal memory)
   defheapseg{
         ldf_defheap_base = .;
         ldf_defheap_size = . + MEMORY_SIZEOF(M2Heap);
```

```
            } >M2Heap
    }   //  end PSH0 sections
}       //  end PSH0 processor

    //  The third and final link project defined in this LDF file
    //  is another DXE project and will be linked against
    //  both the SHARED project and the PSH0 DXE project.

PROCESSOR_PSH1 {
    $PSH1_OBJECTS = psh1.doj, ts_hdr.doj;
    LINK_AGAINST(shared.sm, psh0.dxe)
    OUTPUT (psh1.dxe)

SECTIONS {
    // places code (instructions) in M0 (internal memory)
       code{
            FILL(0xb3c00000)
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS(
                $PSH1_OBJECTS(program) $LIBRARIES(program)
            )
            } >M0Code

    // place data in M2 (internal memory)
       data1{
            INPUT_SECTIONS(
                $PSH1_OBJECTS(data1) $LIBRARIES(data1)
            )
            } >M2Data

    // place data in M1 (internal memory)
       data2{
            INPUT_SECTIONS(
                $PSH1_OBJECTS(data2) $LIBRARIES(data2)
            )
            } >M1Data

    // place C RTL initializers in M2 (internal memory)
       ctor{
            INPUT_SECTIONS( $LIBRARIES(ctor0))
```

```
            INPUT_SECTIONS( $LIBRARIES(ctor1))
            INPUT_SECTIONS( $LIBRARIES(ctor2))
            INPUT_SECTIONS( $LIBRARIES(ctor3))
            INPUT_SECTIONS( $LIBRARIES(ctor))
            } >M2Data

    // place C RTL heap table in M2 (internal memory)
       heaptab{
            INPUT_SECTIONS(
                $PSH1_OBJECTS(heaptab) $LIBRARIES(heaptab)
            )
            } >M2Data

    // place C RTL JALU stack in M2 (internal memory)
       jstackseg{
            ldf_jstack_limit = .;
            ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
            } >M2Stack

    // place C RTL KALU stack in M1 (internal memory)
       kstackseg{
            ldf_kstack_limit = .;
            ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
            } >M1Stack

    // place C RTL heap in M2 (internal memory)
    defheapseg{
            ldf_defheap_base = .;
            ldf_defheap_size = . + MEMORY_SIZEOF(M2Heap);
            } >M2Heap
    }   //  end PSH1 sections
}       //  end PSH1 processor
```

Linking for Overlay Memory

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

```
ARCHITECTURE(ADSP-TS101)
SEARCH_DIR( $ADI_DSP\TS\lib )

MAP(overlay.map)

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

   //  Internal memory blocks are 0x10000 (64K)

MEMORY {
M0Code  {TYPE(RAM) START(0x00000000) END(0x00007FFF) WIDTH(32)}
M0_ovly {TYPE(RAM) START(0x00008000) END(0x0000FFFF) WIDTH(32)}
M1Data  {TYPE(RAM) START(0x00080000) END(0x0008BFFF) WIDTH(32)}
M1Stack {TYPE(RAM) START(0x0008C000) END(0x0008FFFF) WIDTH(32)}
M2Data  {TYPE(RAM) START(0x00100000) END(0x0010BFFF) WIDTH(32)}
M2Heap  {TYPE(RAM) START(0x0010C000) END(0x0010C7FF) WIDTH(32)}
M2Stack {TYPE(RAM) START(0x0010C800) END(0x0010FFFF) WIDTH(32)}
SDRAM   {TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)}
MS0     {TYPE(RAM) START(0x08000000) END(0x0BFFFFFF) WIDTH(32)}
MS1     {TYPE(RAM) START(0x0C000000) END(0x0FFFFFFF) WIDTH(32)}
}
   /*  end of memory  */
   /*  the M0_ovly segment is for overlay storage  */

PLIT {
         //  assign the overlay ID of resolved symbol to j4
      j4 = PLIT_SYMBOL_OVERLAYID;;
```

```
         //  assign "run" address of resolved symbol to j5
      j5 = PLIT_SYMBOL_ADDRESS;;

      JUMP _OverlayManager;;
}

$LIBRARIES = libc.dlb;
$OBJECTS = ts_hdr.doj;

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

OUTPUT(Mgrovly.dxe)

   SECTIONS {
         //  .text output section
      Code {
      INPUT_SECTIONS(
         $P0_OBJECTS(program) $LIBRARIES(program) )

   //  Specify the first overlay. This overlay is stored
   //  in the memory defined by "M0_ovly". It runs in the
   //  memory space defined by "M0Code".
   OVERLAY_INPUT {

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

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

   } > M0_ovly

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

```
OVERLAY_INPUT {
      OVERLAY_OUTPUT (overlay2.ovl )
      INPUT_SECTIONS (overlay2.doj (program) )
   } > M0_ovly
} > M0Code

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

.plit {
      // linker inserts instructions here.
      } > M0Code

data1 {
      INPUT_SECTIONS(
         $P0_OBJECTS(data1) $LIBRARIES(data1) )
      } > M2Data

data2 {
      INPUT_SECTIONS(
         $P0_OBJECTS(data2) $LIBRARIES(data2) )
      } > M1Data

//  place C RTL initializers in M0 (internal memory)
ctor{
      INPUT_SECTIONS( $LIBRARIES(ctor0))
      INPUT_SECTIONS( $LIBRARIES(ctor1))
      INPUT_SECTIONS( $LIBRARIES(ctor2))
      INPUT_SECTIONS( $LIBRARIES(ctor3))
      INPUT_SECTIONS( $LIBRARIES(ctor))
   } >M2Data

//  place C RTL heap table in M0 (internal memory)
heaptab{
      INPUT_SECTIONS(
         $PSH0_OBJECTS(heaptab) $LIBRARIES(heaptab)
      )
      } >M2Data
```

```
    // place C RTL JALU stack in M0 (internal memory)
       jstackseg{
             ldf_jstack_limit = .;
             ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
       } >M2Stack

    // place C RTL KALU stack in M1 (internal memory)
       kstackseg{
             ldf_kstack_limit = .;
             ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
       } >M1Stack

    // place C RTL heap in M0 (internal memory)
       defheapseg{
             ldf_defheap_base = .;
             ldf_defheap_size = . + MEMORY_SIZEOF(M2Heap);
       } >M2Heap
    }   //  end P0 sections
 }      //  end P0 processor
```

D LDF PROGRAMMING EXAMPLES FOR SHARC PROCESSORS

This appendix provides several typical LDFs used with SHARC processors. As you modify these examples, refer to the syntax descriptions in "LDF Commands" on page 3-29.

This appendix provides the following examples:

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

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

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

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

Linking a Single-Processor SHARC System

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

- ARCHITECTURE() defines the processor type.
- SEARCH_DIR() adds the lib and current working directory to the search path.
- $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{} defines a processor and place program sections for that processor's output file, using the memory definitions.

Listing D-1. Single-Processor System LDF Example

```
//  Link for the ADSP-21161
ARCHITECTURE(ADSP-21161)
SEARCH_DIR ( $ADI_DSP\211xx\lib )
MAP (SINGLE-PROCESSOR.XML)   // Generate a MAP file

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

   //  lib161.dlb is an ADSP-2116x-specific library
```

```
   //  and must precede libc.dlb, the C library
   //  to link the 2116x-specific routines.

$LIBS = lib161.dlb, libc.dlb;

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

   //  161_hdr.doj is the standard initialization file for 2116x
$OBJS = $COMMAND_LINE_OBJECTS, 161_hdr.doj;

   //  A linker project to generate a .DXE file
PROCESSOR P0
{
   OUTPUT ( .\SINGLE.DXE )   // The name of the output file

   MEMORY                    // Processor-specific memory command
      { INCLUDE("21161_memory.h") }

   SECTIONS                  // Specify the output sections
      {
      INCLUDE( "21161_sections.h" )
      }   // end P0 sections
}         // end P0 processor
```

Linking Large Uninitialized Variables

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

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers, this action can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur long download times when used with an emulator. This situation also may occur when you boot from a loader file (because of the increased file size). Listing D-2 shows an example of assembly source code. Listing D-3 shows the use of the NO_INIT and ZERO_INIT sections to avoid initialization of a segment.

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

ⓘ Refer to "SECTIONS{}" on page 3-48 for more information on the NO_INIT and ZERO_INIT section qualifiers.

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

Listing D-2. Large Uninitialized Variables: Assembly Source

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

Listing D-3. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-21161)
$OBJECTS = $COMMAND_LINE_OBJECTS;   // Libraries & objects from
                                    // the command line
MEMORY {
   mem_sdram {
      TYPE(DM RAM) START(0x3000000) END(0x30FFFFF) WIDTH(32)
            }  // end segment
      }  // end memory

PROCESSOR P0 {
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
           // NO_INIT section isn't written to the output file
   SECTION {
      sdram_output NO_INIT {
        INPUT_SECTIONS( $OBJECTS ( sdram_area ) )}
      >mem_sdram;
   SECTION {
      zero_sdram_output ZERO_INIT {
         INPUT_SECTIONS ( $OBJECTS ( zero_sdram_area ) )}
      >mem_sdram;
      }   // end section
   }      // end processor P0
```

Linking for MP and Shared Memory

When linking executable files for a multiprocessor system using shared memory, the .LDF file describes the multiprocessor memory offsets, shared memory, each processor's memory, and places code for each processor. Note the following in the example .LDF file in Listing D-4 on page D-6:

- The ARCHITECTURE() command defines the processor type, which can be one type only.
- The SEARCH_DIR() command adds the lib and current working directory to the search path.
- The $OBJS and $LIBS macros get object (.DOJ) and library (.DLB) file input.
- The MPMEMORY{} command defines each processor's offset within multiprocessor memory.
- The SHARED_MEMORY{} command identifies the output for the shared memory items.
- The MAP() command outputs map files.
- The MEMORY{} command defines memory for the processors.
- The PROCESSOR{} and SECTIONS{} commands define each processor and place program sections using memory definitions for each processor's output file.
- The LINK_AGAINST() commands resolve symbols within multiprocessor memory.

Listing D-4. LDF for a Multiprocessor System With Shared Memory

```
ARCHITECTURE(ADSP-21161)
SEARCH_DIR( $ADI_DSP\211xx\lib )
```

```
//  Allocate multiprocessor memory space with the MPMEMORY{}
//  command. The values represent an "offset" that the linker
//  uses to resolve undefined symbols in one DXE file to symbols
//  defined in another DXE. That is, the offset is added
//  to each defined symbol's value.
//  For example, PROCESSOR project PSH0 references the undefined
//  symbol "buffer", and PROCESSOR project PSH1 defines "buffer"
//  at address 0x22000. The linker will "fix up" the reference
//  to "buffer" in PSH0's code to address:
//  0x22000 + MPMEMORY(PSH1) = 0x22000 + 0x120000 = 0x142000

MPMEMORY
{
   PSH0 { START (0x100000) }
   PSH1 { START (0x120000) }
}
MEMORY
{
      // This is used for all processors. Alternatively,
      // a PROCESSOR can describe its own memory.

seg_rth      { TYPE(PM RAM) START(0x00040000) END(0x000400ff)
               WIDTH(48) }
seg_init     { TYPE(PM RAM) START(0x00040100) END(0x000401ff)
               WIDTH(48) }
seg_int_code { TYPE(PM RAM) START(0x00040200) END(0x00040287)
               WIDTH(48) }
seg_pmco     { TYPE(PM RAM) START(0x00040288) END(0x000419ff)
               WIDTH(48) }
seg_pmda     { TYPE(PM RAM) START(0x00042700) END(0x00043fff)
               WIDTH(32) }
seg_dmda     { TYPE(DM RAM) START(0x00050000) END(0x00051fff)
               WIDTH(32) }
seg_heap     { TYPE(DM RAM) START(0x00052000) END(0x00052fff)
               WIDTH(32) }
seg_stak     { TYPE(DM RAM) START(0x00053000) END(0x00053fff)
               WIDTH(32) }
}
```

```
$LIBRARIES = libc161.dlb;
   //  Three link projects are specified in one LDF file.
   //  The first link project is a shared memory link project
   //  against which the PROCESSOR projects are linked.

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

$SHARED_OBJECTS = shared.doj;

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

OUTPUT(shared.sm)

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

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

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

PROCESSOR_PSH0 {
   $PSH0_OBJECTS = psh0.doj, 161_hdr.doj;
   LINK_AGAINST(shared.sm)
   OUTPUT (psh0.dxe)

   SECTIONS {
      dxe_pmco{ INPUT_SECTIONS(
                 $PSH0_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco)
                } >mem_pmco
```

```
      dxe_pmda{ INPUT_SECTIONS(
                 $PSH0_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda)
                } >mem_pmda
      dxe_dmda{ INPUT_SECTIONS(
                 $PSH0_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda)
                } >mem_dmda
      dxe_init{ INPUT_SECTIONS(
                 $PSH0_OBJECTS(seg_init) $LIBRARIES(seg_init)
                } >mem_init
      dxe_rth { INPUT_SECTIONS(
                 $PSH0_OBJECTS(seg_rth)  $LIBRARIES(seg_rth)
                } >mem_rth
      stackseg {   // Allocate a stack for the application.
                ldf_stack_space = .;
                ldf_stack_length = 0x2000;
                } >mem_stak

      heap   {   // Allocate a heap for the application.
                ldf_heap_space = .;
                ldf_heap_end = ldf_heap_space + 0x2000;
                ldf_heap_length = ldf_heap_end - ldf_heap_space;
                } >mem_heap

   }   //   end PSH0 sections
}      //   end PSH0 processor

   //  The third and final link project defined in this LDF file
   //  is another DXE project and will be linked against
   //  both the SHARED project and the PSH0 DXE project.

PROCESSOR_PSH1 {
   $PSH1_OBJECTS = psh1.doj, 161_hdr.doj;
   LINK_AGAINST(shared.sm, psh0.dxe)
   OUTPUT (psh1.dxe)

   SECTIONS {
      dxe_pmco{ INPUT_SECTIONS(
                 $PSH1_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco)
                } >mem_pmco
      dxe_pmda{ INPUT_SECTIONS(
```

```
                $PSH1_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda)
               } >mem_pmda
        dxe_dmda{ INPUT_SECTIONS(
                $PSH1_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda)
               } >mem_dmda
        dxe_init{ INPUT_SECTIONS(
                $PSH1_OBJECTS(seg_init) $LIBRARIES(seg_init)
               } >mem_init
        dxe_rth { INPUT_SECTIONS(
                $PSH1_OBJECTS(seg_rth)  $LIBRARIES(seg_rth)
               } >mem_rth
        stackseg {   // Allocate a stack for the application.
               ldf_stack_space = .;
               ldf_stack_length = 0x2000;
               } >mem_stak
        heap    {    // Allocate a heap for the application.
               ldf_heap_space = .;
               ldf_heap_end = ldf_heap_space + 0x2000;
               ldf_heap_length = ldf_heap_end - ldf_heap_space;
               } >mem_heap

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

Reflective Semaphores

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

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

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

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

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

To check the semaphore, each SHARC processor reads from its own internal memory. Any object in the project can be mapped to an appropriate memory segment defined in the broadcast space for use as a reflective semaphore. If an object defining symbol `SemA` is mapped to a broadcast space, when the program writes to `SemA`, the written value appears at the aliased internal address of each processor in the cluster. Each processor may read the value using `SemA`, or read it from internal memory by selecting (`SemA-0x380000`), thus avoiding bus traffic.

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

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

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

Linking for Overlay Memory

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

```
ARCHITECTURE(ADSP-21161)
SEARCH_DIR( $ADI_DSP\211xx\lib )

MAP(overlay.map)

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

MEMORY {
  seg_rth  {TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48)}
  seg_init {TYPE(PM RAM) START(0x00040100) END(0x000401ff) WIDTH(48)}
  seg_init_code {TYPE(PM RAM) START(0x00040200) END(0x00040287) WIDTH(48)}
  seg_pmco {TYPE(PM RAM) START(0x00040288) END(0x000409ff) WIDTH(48)}
  seg_ovly {TYPE(PM RAM) START(0x00040100) END(0x000419ff) WIDTH(48)}
  seg_pmda {TYPE(PM RAM) START(0x00042700) END(0x00043fff)  WIDTH(32)}
  seg_dmda {TYPE(DM RAM) START(0x00050000) END(0x00051fff) WIDTH(32)}
  seg_heap {TYPE(DM RAM) START(0x00052000) END(0x00052fff) WIDTH(32)}
  seg_stak {TYPE(DM RAM) START(0x00053000) END(0x00053fff) WIDTH(32)}

      /*  mem_pmco is the "run" memory segment   */
      /*  mem_ovly is the "live" memory segment  */

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

PLIT {
            //  Each of five instructions are duplicated
            //  for each symbol in an overlay.
```

```
        R0 = PLIT_SYMBOL_OVERLAYID;
              //  Assigns overlay ID of the resolved symbol to R0.
        R1 = PLIT_SYMBOL_ADDRESS;
              //  Assigns “run” address of resolved symbol to R1.
        dm(_overlayID) = R0;
        dm(_pf) = R1;
        JUMP _OverlayManager;
  }

  $LIBRARIES = libc161.dlb;
  $OBJECTS = 161_hdr.doj;

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

  OUTPUT(mgrovly.dxe)

     SECTIONS {
        //  .text output section
        seg_pmco {
        INPUT_SECTIONS(
           $P0_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco) )

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

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

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

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

```
ALGORITHM( ALL_FIT)

} > mem_ovly

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

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

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

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

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

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

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

   dxe_rth  {
```

```
            INPUT_SECTIONS(
               $P0_OBJECTS(seg_rth)  $LIBRARIES(seg_rth) )
            } > mem_rth

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

         heap {    // Allocate a heap for the application.
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + 0x2000;
            ldf_heap_length = ldf_heap_end - ldf_heap_space;
            } > mem_heap

      }
   }
```

E LDF PROGRAMMING EXAMPLES FOR BLACKFIN PROCESSORS

This appendix provides several typical LDFs. used with Blackfin processors. As you modify these examples, refer to the syntax descriptions in "LDF Commands" on page 3-29.

This appendix provides the following examples.

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

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

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

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

Linking for a Single-Processor System

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

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

Listing E-1. Example .LDF File for a Single-Processor System

```
ARCHITECTURE(ADSP-BF535)

SEARCH_DIR( $ADI_DSP\Blackfin\lib )

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

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

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

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

$OBJECTS = $COMMAND_LINE_OBJECTS;

// A linker project to generate a DXE file

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

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

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

Linking Large Uninitialized or Zero-initialized Variables

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

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers, this action can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur long download times when used with an emulator. This situation also may occur when you boot from a loader file (because of the increased file size). Listing E-2 shows an example of assembly source code. Listing E-3 shows the use of the NO_INIT and ZERO_INIT sections to avoid initialization of a segment.

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

Refer to "SECTIONS{}" on page 3-48 for more information on the NO_INIT and ZERO_INIT section qualifiers.

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

Listing E-2. Large Uninitialized Variables: Assembly Source

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

Listing E-3. Large Uninitialized Variables: .LDF File Source

```
ARCHITECTURE(ADSP-BF535)
$OBJECTS = $COMMAND_LINE_OBJECTS;   // Libraries & objects from
                                    // the command line
MEMORY {
   mem_extram {
      TYPE(RAM) START(0x10000) END(0x15fff) WIDTH(8)
      }                             // end segment
   }                                // end memory

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

Linking for Assembly Source File

Listing E-5 shows an example .LDF file for an ADSP-BF535 DSP that describes a simple memory placement of an assembly source file. The file in Listing E-4 contains code and data that is to reside in, and execute from, L2 SRAM. This example assumes that the code and data declared in L2 memory is cacheable within L1 code and data memories. The LDF file includes two commands, MEMORY and SECTIONS, which are used to describe specific memory and system information. Refer to Notes for Listing 3-2 on page 3-9 for information on items in this basic example.

Listing E-4. MyFile.ASM

```
.SECTION program;
.GLOBAL main;
main:
p0.l = myArray;
p0.h = myArray;
r0 = [p0++];
…
.SECTION data1;
.GLOBAL myArray;
.VAR myArray[256] = "myArray.dat";
```

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

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

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

// Describe the physical system memory below

MEMORY{
```

```
    // 256KB L2 SRAM memory segment for user code
    // and data L2SRAM
    MEM_L2SRAM {TYPE(RAM) START(L2_START) END(L2_END) WIDTH(8)}
}

PROCESSOR p0{
    OUTPUT($COMMAND_LINE_OUTPUT_FILE)

    SECTIONS{
    L2SRAM{
        // Align L2 instruction segments on a 2-byte boundaries
        INPUT_SECTION_ALIGN(2)
        INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
        // Align L2 data segments on 1-byte boundary
        INPUT_SECTION_ALIGN(INPUT_SECTIONS
                          ($OBJECTS(data1) $LIBRARIES(data1))
        >MEM_L2SRAM
        }    // end section
    }        // end processor P0
```

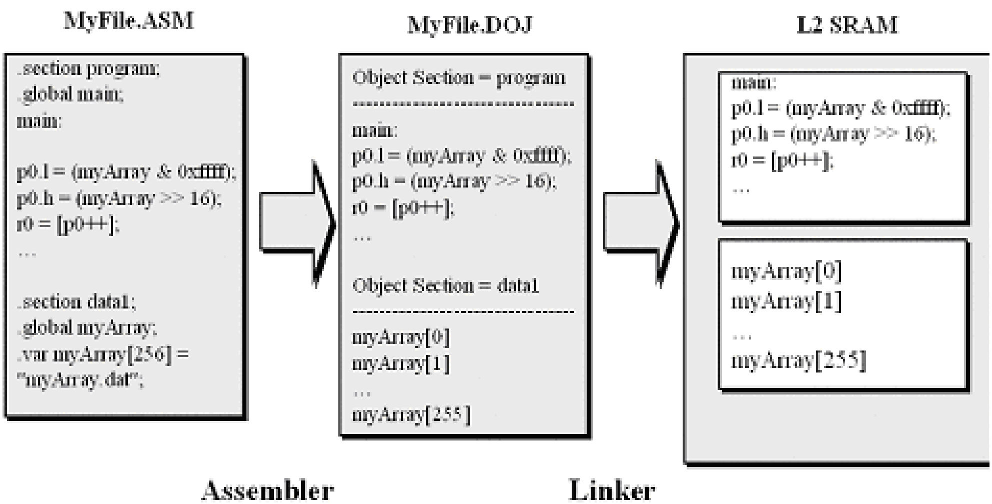


Figure E-1. Assembly-to-Memory Code Placement

Linking for C Source File – Example 1

Listing E-7 shows an example LDF that describes the memory placement of a simple C source file (Listing E-6) which contains code and data that is to reside in, and execute from, L2 SRAM. This example also assumes that the code and data declared in L2 memory is cacheable within L1 code and data memories. The LDF file includes two commands, `MEMORY` and `SECTIONS`, which are used to describe specific memory and system information. Refer to Notes for Listing 3-2 on page 3-9 for information on items in this basic example.

Listing E-6. Simple C Source File Example 1

```
int myArray[256];

void main(void){
  int i;

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

}   // end main ()
```

Listing E-7. Example: Simple C-based .LDF File for ADSP-BF535 Processor

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

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

$LIBRARIES = LIBS, librt535.dlb;

$OBJECTS = crts535.doj, $COMMAND_LINE_OBJECTS crtn535.doj;
MEMORY{
```

```
    // 248KB of L2 SRAM memory segment for user code and data
  MEM_L2SRAM {TYPE(RAM) START(0xf0000000) END(0xF003dfff) WIDTH(8)}
    // 4KB of L2 SRAM memory for C run-time stack (user mode)
  MEM_STACK {TYPE(RAM) START(0xf003e000) END(0xf003efff) WIDTH(8)}
    // 4KB of Scratch SRAM for Heap memory segment
  MEM_HEAP {TYPE(RAM) START(0xFFB00000) END(0xFFB00FFF) WIDTH(8)}
}

PROCESSOR p0{

 OUTPUT($COMMAND_LINE_OUTPUT_FILE)

  SECTIONS{}
  // Declare L2 Input objects below…
   L2_SRAM{
     // Align L2 instruction segments on a 2-byte boundaries
     INPUT_SECTION_ALIGN(2)
     INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program)
     // Align L2 data segments on a 1-byte boundary
     INPUT_SECTION_ALIGN(1)
     INPUT_SECTIONS($OBJECTS(data1) $LIBRARIES(data1)
     INPUT_SECTIONS( $OBJECTS(cplb) $LIBRARIES(cplb))
     INPUT_SECTIONS( $OBJECTS(cplb_code) $LIBRARIES(cplb_code))
     INPUT_SECTIONS( $OBJECTS(cplb_data) $LIBRARIES(cplb_data))
     // Align L2 constructor data segments on a 1-byte boundary
     // (C++ only)
     INPUT_SECTION_ALIGN(1)
     INPUT_SECTIONS($OBJECTS(constdata) $LIBRARIES(constdata))
   }>MEM_L2SRAM

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

  // 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;
}>MEM_HEAP

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

Linking for Complex C Source File – Example 2

Listing 1-3 shows an example LDF that describes the memory placement of a C source file. This file contains code and data that is to reside in, and execute from, L1, L2, Scratchpad SRAM, and external SDRAM Banks 0 through 3. The .LDF file includes two commands, MEMORY and SECTIONS, which are used to describe specific memory and system information. Refer to Notes for Listing 3-2 on page 3-9 for information on items in this complex example.

Listing E-8. Complex C Source File Example

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

...
}

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

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

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

int i;
x = 0x5555;

...
}
```

The following is an example of an LDF file (for ADSP-BF535 DSP) which is based on the complex C source from Listing 1-13. Also see Figure E-2 on page E-16.

Listing E-9. C .LDF File Example - SDRAM.LDF

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

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

$LIBRARIES = LIBS, librt535.dlb;

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

// Define physical system memory below…
MEMORY{
  // 16KB of user code in L1 SRAM segment
  MEM_L1_CODE_SRAM {TYPE(RAM) START(0xFFA00000) END(0xFFA03FFF)
                    WIDTH(8)}
  // 16KB of user data in L1 Data Bank A SRAM
  MEM_L1_DATAA_SRAM {TYPE(RAM) START(0xFF800000) END(0xFF803FFF)
                     WIDTH(8)}
  // 16KB of user data in L1 Data Bank B SRAM
  MEM_L1_DATAB_SRAM {TYPE(RAM) START(0xFF900000) END(0xFF903FFF)
                     WIDTH(8)}

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

  // 248KB of user code and data in L2 SRAM segment
  MEM_L2_SRAM {TYPE(RAM) START(0xF0000000) END(0xF003DFFF)
               WIDTH(8)}
  // 4KB for heap in L2 SRAM (for dynamic memory allocation)
  MEM_HEAP {{TYPE(RAM) START(0xF003E000) END(0xF003EFFF)
             WIDTH(8)}
```

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

  // 4 x 128MB External SDRAM memory segments
  MEM_SDRAM_BANK0 {TYPE(RAM) START(0x00000000) END(0x07FFFFFF)
                   WIDTH(8)}
  MEM_SDRAM_BANK1 {TYPE(RAM) START(0x08000000) END(0x0FFFFFFF
                   WIDTH(8)}
  MEM_SDRAM_BANK2 {TYPE(RAM) START(0x10000000) END(0x17FFFFFF)
                   WIDTH(8)}
  MEM_SDRAM_BANK3 {TYPE(RAM) START(0x18000000) END(0x1FFFFFFF)
                   WIDTH(8)}
  }   // end MEMORY{}

PROCESSOR p0{
 OUTPUT($COMMAND_LINE_OUTPUT_FILE)

 SECTIONS{
  // Input section declarations for L1 code memory
  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
  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
  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

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

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

SDRAM_0{
   // Align external SDRAM data segments on a 1-byte boundary
   INPUT_SECTION_ALIGN(1)
   INPUT_SECTIONS($OBJECTS(SDRAM_0))
}>MEM_SDRAM_BANK0

SDRAM_1{
   // Align external SDRAM data segments on a 1-byte boundary
   INPUT_SECTION_ALIGN(1)
   INPUT_SECTIONS($OBJECTS(SDRAM_1))
}>MEM_SDRAM_BANK1

SDRAM_2{
   // Align external SDRAM data segments on a 1-byte boundary
```

```
      INPUT_SECTION_ALIGN(1)
      INPUT_SECTIONS($OBJECTS(SDRAM_2))
   }>MEM_SDRAM_BANK2

   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{}
```

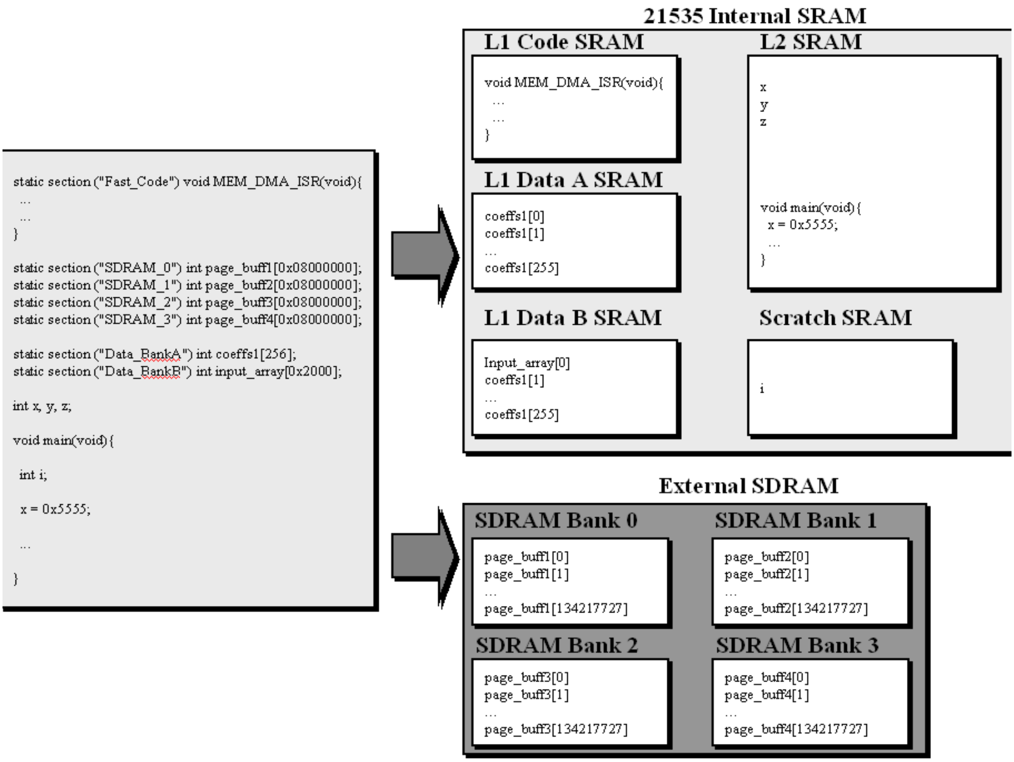


Figure E-2. C-to-Memory Code Placement

Linking for Overlay Memory

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

Listing E-10 shows an example .LDF file for an overlay-memory system. For more information on this .LDF file, see the comments in the listing.

Listing E-10. Example .LDF File for an Overlay-Memory System

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

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

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

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

```
    SECTIONS
    {
        reset { INPUT_SECTIONS($OBJECTS(IVreset))
        } >MEM_PROGRAM
        itab  { INPUT_SECTIONS($OBJECTS(IVpwrdwn))
        } >MEM_PROGRAM

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

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

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

$LIBRARIES = LIBS, librt535.dlb;

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

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

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

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

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

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

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

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

OVERLAY_OUTPUT( overlay1.ovl )

      /* Only take the code from the file overlay1.doj.
      If this code needs data, it must be either the INPUT of a
```

```
     data overlay or the INPUT to non-overlay data memory. */
   INPUT_SECTIONS(overlay1.doj( program))

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

    ALGORITHM( ALL_FIT )
    SIZE(0x100)

} > MEM_OVLY

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

} > program

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

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

   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
}
```

I INDEX

A

D

F

M

P

R

S

T

U