CrossCore® Embedded Studio 1.0.0 Linker and Utilities Manual

(including ADSP-BFxxx and ADSP-21xxx)

Revision 1.0, March 2012

Part Number
82-100115-01

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

ANALOG DEVICES

Copyright Information

Printed in the USA.

Disclaimer

Trademark and Service Mark Notice

CONTENTS

MEMORY OVERLAYS AND ADVANCED LDF COMMANDS

ARCHIVER

MEMORY INITIALIZER

UTILITIES

LDF PROGRAMMING EXAMPLES FOR BLACKFIN PROCESSORS

LDF PROGRAMMING EXAMPLES FOR SHARC PROCESSORS

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for Analog Devices embedded processors.

Purpose of This Manual

The *Linker and Utilities Manual* contains information about the linker and utility programs for Blackfin® (ADSP-BFxxx) and SHARC® (ADSP-21xxx) processors. These processors 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. The manual describes the linking process in the CrossCore® Embedded Studio application environment.

This manual provides information on the linking process and describes the syntax for the linker's command language—a scripting language that the linker reads from the linker description file (`.ldf`). The manual leads you through using the linker, archiver, and utilities to produce DSP programs and provides reference information on the file utility software.

Intended Audience

The primary audience for this manual is programmers 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", provides an overview of the linker and utility programs.
- Chapter 2, "Linker", describes how to combine object files into reusable library files to link routines referenced by other object files.
- Chapter 3, "Linker Description File", describes how to write an `.ldf` file to define the target.
- Chapter 4, "Memory Overlays and Advanced LDF Commands", describes how overlays and advanced LDF commands are used for memory management and complex linking.
- Chapter 5, "Archiver", describes the `elfar` archiver utility used to combine object files into library files, which serve as reusable resources for code development.
- Chapter 6, "Memory Initializer", 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", lists and describes the file formats that the development tools use as inputs or produce as outputs.
- Appendix B, "Utilities", describes the utility programs that provide legacy and file conversion support.

- Appendix C, "LDF Programming Examples for Blackfin Processors", provides code examples of `.ldf` files used with Blackfin processors.
- Appendix D, "LDF Programming Examples for SHARC Processors", provides code examples of `.ldf` files used with SHARC processors.

What's New in This Manual

This is Revision 1.0 of the *Linker and Utilities Manual*. This manual documents linker support for SHARC and Blackfin processors. For future revisions, this section will document functionality that is new to updates of CrossCore Embedded Studio, modifications due to new processors, and fixes to reported problems.

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/technical_support`
- E-mail tools questions to `processor.tools.support@analog.com`
- E-mail processor questions to
  `processor.support@analog.com (World wide support)`
  `processor.china@analog.com (China support)`
- 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 CrossCore Embedded Studio linker supports the following processor families from Analog Devices.

- Blackfin (ADSP-BFxxx)
- SHARC (ADSP-21xxx)

Refer to the CrossCore Embedded Studio online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices Web site and the CrossCore Embedded Studio online help.

Analog Devices Web Site

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

To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, 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 about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit MyAnalog.com to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit http://ez.analog.com to sign up.

Notation Conventions

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

Example	Description
Close command (**File** menu)	Titles in bold style reference sections indicate the location of an item within the CrossCore Embedded Studio 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 ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `letter gothic` font.
filename	Non-keyword placeholders appear in text with italic style format.
	**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 CrossCore Embedded Studio development tools and their use in the DSP project development process.

The code examples in this manual have been compiled using release 1.0.0.

This chapter includes:

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`) consumes object and library files to produce executable files, which can be loaded onto a simulator or target processor. The linker also combines the debug information from the input files that is embedded into an executable file and is used by the debugger. The linker also can optionally produce map files and other reports to help the user understand the result of the linker process. Debug information is embedded in the executable file.

After running the linker, you test the output with a simulator or emulator. Refer to 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 *Loader and Utilities 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 the CrossCore Embedded Studio Integrated Development Environment (IDE), the linker utility consumes object files (`.doj`) and library files (`.dlb`) 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 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 toward producing an executable file is to compile or assemble C, C++, or assembly source files into *object files*. The CrossCore Embedded Studio development software assigns a `.doj` extension to object files (Figure 1-1).

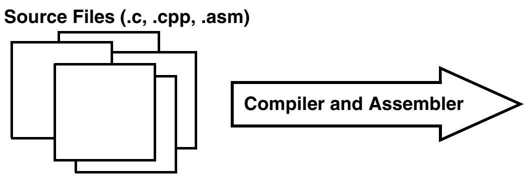

Figure 1-1. Compiling and Assembling

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 CrossCore Embedded Studio 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 "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 the above 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;
_abs.end;
```

In the above 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.

In the example, the linker knows what code is associated with the label `_abs` because it is delimited with the label `_abs.end`. For some linker features, especially unused section elimination (see "ELIMINATE_SECTIONS()" on page 3-35), the linker must be able to determine the end of code or data associated with a label. In assembly code, the end of a function data block can be marked with a label with the same name as the label at the start of the name with .end appended to it. It is also possible to prepend a "." in which case the label will not appear in the symbol table which can make debugging easier.

Listing 1-1 shows uses of .end labels in assembly code.

Listing 1-1. Using Labels in Assembly Code

```
start_label:
                       // code
start_label.end        // marks end of code section

new_label:
                       // code
new_label.END:         // end label can be in upper case

one_entry:             // function one_entry includes the code
                       // in second_entry
   second_entry:       // more code
.one_entry.end:
   .second_entry.end:  // prepended "." omits end label
                       // from the symbol table
```

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 are program (for code) and data1 (for data). Additional input section names are defined in .ldf files. (For more information on memory mapping, see "Specifying the Memory Map" on page 2-13.)

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 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:

The `section ("name")` extension is optional and applies only to the declaration to which it is applied. Note that the new function (`func2`) does not have `section ("extern")` and will be placed in the default input section `program`. For more information on LDF sections, refer to "Specifying the Memory Map" on page 2-13.

```
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 *C/C++ Compiler and Library Manual* for the appropriate target processor and "Placing Code on the Target" on page 2-34.

ⓘ 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).

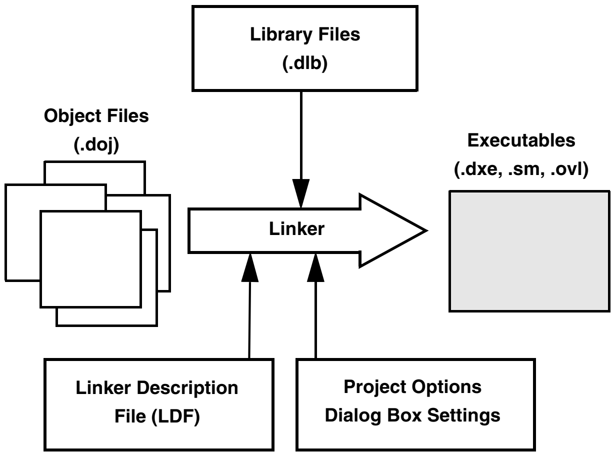

Figure 1-2. Linking Diagram

Linking enables your code to run efficiently in the target environment. Linking is described in detail in Chapter 3, "Linker".

When developing a new project, use the **New Project** wizard to generate the project's `.ldf` file. For more information, search online help for "Project Wizard".

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 from within the IDE. 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.

"." Character Identifier

The 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 valid character in labels. This behavior creates 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 with the preprocessor because this syntax does not yield any replacement, and the preprocessor does not parse the `reg` as a separate identifier. The macro must 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 information on the `pp` preprocessor, see the *Assembler and Preprocessor Manual*.

The compiler has it own preprocessor that permits the use of preprocessor commands within C/C++ source. The compiler preprocessor automatically runs before the compiler. For more information, see the *C/C++ Compiler and Library Manual* for the appropriate target architecture.

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 *Loader and Utilities Manual* which provides detailed descriptions of the processes and options used to generate boot-loadable loader (`.ldr`) files for the appropriate target processor. This manual also describes the splitting utility, which creates the non-boot loadable files that execute from the processor's external memory.

In general:

- SHARC 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`) to generate non-bootable PROM image files, which execute from the processor's external memory.

- SHARC ADSP-2116x/2126x/2136x/2137x/2147x/2148x 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`).
- SHARC processors use the splitter utility (`elfSpl21k.exe`) to generate non-bootable PROM image files, which execute from the processor's external memory.
- 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`).

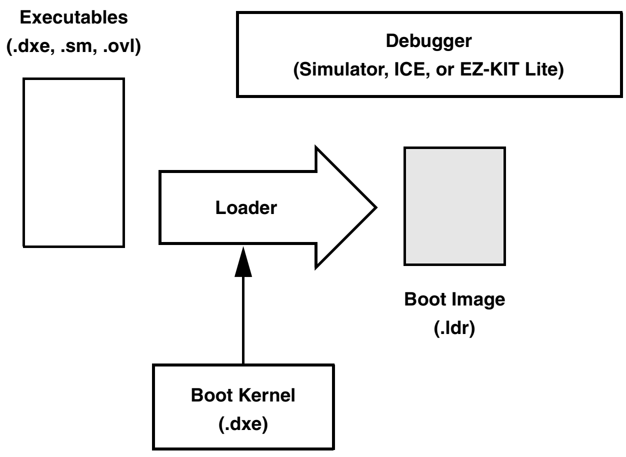

Figure 1-3. Using the Loader to Create an Output File

CrossCore Embedded Studio includes boot kernel files (`.dxe`), which are used automatically when you run the loader. You can also customize the provided boot kernel source files by modifying and rebuilding them.

Figure 1-4 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 illustrates the generation of a loader file for a multiprocessor architecture.

The `.sm` and `.ovl` files should reside in the same directory that contains the input `.dxe` file(s) or in the current working directory. If your system does not use shared memory or overlays, `.sm` and `.ovl` files are not required.

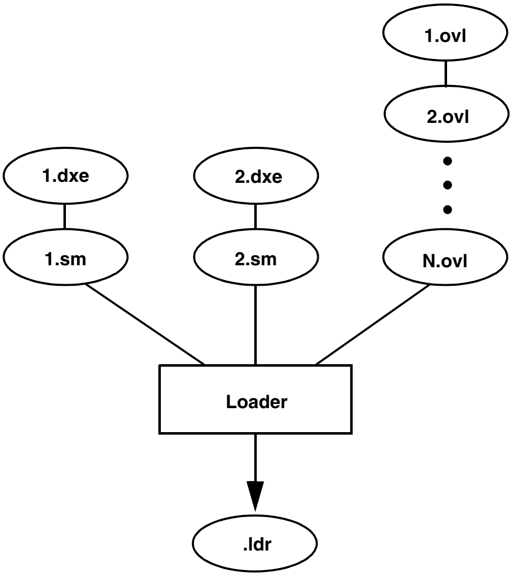

Figure 1-4. Input Files for a Multiprocessor System

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

2 LINKER

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

You can run the CrossCore Embedded Studio linker from a command line or from the IDE.

You can load linker output into the 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.

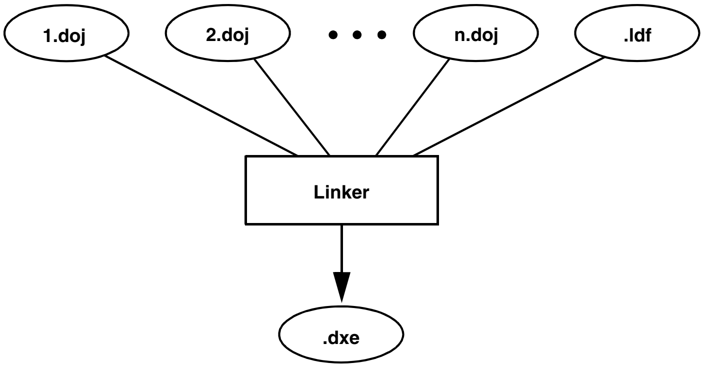

Figure 2-1. Linking Object Files to Produce an Executable File

When developing a new project, use the **New Project** wizard to generate the project's .ldf file. For more information, search online help for "Project Wizard".

In a multiprocessor system, a .dxe file for each processor is generated. For example, for a dual-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 4-3 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 CrossCore Embedded Studio 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 application.

Directing Linker Operation

Linker operations are directed by these options and commands:

- Linker command-line switches (options). Refer to "Linker Command-Line Reference" on page 2-39.
- In an IDE environment: Options on the linker pages of the **Tool Settings** dialog box. See "Project Builds" on page 2-9.
- LDF commands. Refer to "LDF Commands" on page 3-31 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.

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 **Tool Settings** tab for the linker page displays the name of the `.ldf` file, which provides the 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 (input section names) that can be used to direct one or more portions of object code to particular input sections.
- As directed by the `.ldf` file, the linker maps each input section in the object code to an output section.
- 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. CrossCore Embedded Studio provides an `.ldf` file that supports the default mapping of each processor type.

When developing a new project, use the **New Project** wizard to generate the project's `.ldf` file. For more information, search online help for "Project Wizard".

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.

Linker Symbol Resolution

In addition to placing input sections from object files into output sections in the executable file, the linker also resolves all references to symbols. When an object file refers to a symbol that appears in another object file, or even in the same object file, the linker needs to replace the reference to the symbol with the address of where the symbol was mapped.

If a symbol is not defined in any of the object files that are being linked, either passed to the linker on the command line or named in the LDF, the linker will attempt to resolve the symbol by looking to see if it is defined in any of the libraries that are part of the link. Library files are also passed to the linker either by the command line or by explicitly being named in the LDF file.

Library files (or archives) are special collections of object files that are useful for sharing functions across many programs. For information on how to create a library file, refer to Chapter 5, “Archiver”.

When an object file has a symbol reference to a global, the linker needs to resolve that symbol—that is determine which global in another file that the symbol references. The first place searched is the list of all global symbols in the object files specified for the link. When considering what symbols to search, the linker considers only symbols that are in sections mapped by the LDF into the final executable. So if the symbol `my_symbol` was defined in a section that was not mapped by any commands in the LDF, a reference to the symbol would generate the linker error:

```
[Error li1060]   The following symbols are referenced, but not
        mapped:
        '_my_symbol' referenced from need_my_symbol.doj(program)
```

The linker helps the user by reporting as warnings those symbols that were defined by an object file but did not get mapped by the LDF. For example:

```
[Warning li2060]   The following input section(s) that contain
         program code and/or data have not been placed into the
         executable for processor 'p0'as there are no relevant
         commands specified in the LDF:

         my_symbol.doj(unmapped_section_name)
```

The warning `li2060` is not reported if the link was successful. The warning is reported when `Error li1060` is reported as an aid to the programmer who may have mistakenly forgotten to map the section `"unmapped_section_name"` in the LDF.

There is no ordering for object files; all object files are given equal consideration when resolving symbols. It is an error for multiple object files to define the same symbol name, but only if two or more of these symbols occur in sections that are mapped by the LDF. It is only an error if the symbols in both files are in sections that are mapped by the LDF. The same symbol can be defined as a global in two object files without error, if one of the symbols is defined in a section that is not mapped by the LDF.

If after considering all object files there are still symbol references that are not resolved, the linker will search for the symbol in the library files that were specified for the link. When considering libraries, the order of the libraries is important. The linker will only search libraries for a symbol until it finds the symbol. If a symbol is defined in more than one library it would not be an error. The linker simply resolves the reference using the first occurrence it finds. In all cases, only symbols that are in sections that are mapped in the LDF are considered. A library that contains an object that defines `my_symbol`, but where `my_symbol` is in a section that is not mapped, does not resolve the symbol, and the linker continues to look in other libraries.

Libraries are searched in the order that they appear in commands in the LDF. There is a distinction between macro definitions and commands. The appearance of the name of a library in a definition of a macro does not count as an appearance in a command. If the LDF contains the following statements, the definition of the macro `$libs` is not considered a LDF command:

```
        $LIBS = liba.dlb, libb.dlb
...
        INPUT_SECTIONS( libc.dlb(sectionfoo) )
        INPUT_SECTIONS( $LIBS(sectionfoo) )
```

The first LDF command with a library is the `INPUT_SECTIONS()` command with `libc.dlb`. The next `INPUT_SECTIONS()` command marks the appearance of `liba.dlb` and `libb.dlb` (in that order). The order that libraries are searched by the linker is `libc`, `liba`, then `libb`. If all libraries have an object that defines `my_symbol`, only the object from `libc.dlb` is added to the link; once the linker can satisfy the reference, there is no further searching for the symbol.

The ordering of libraries is fixed for the entire linking process. It does not matter what sections an `INPUT_SECTIONS` command is mapping for determining the library order. It is strictly dependent on the fact that the library name appears in any LDF command.

When it is necessary to override an object defined in a library, link the program with both the library and also a replacement `.doj` file. Usually, the replacement `.doj` file defines all the same symbols as the original version (though symbols that are not referenced by the program can be omitted). The linker then uses the `.doj` file and ignores the object from the library. Note that if the program references any symbols that are (mapped) in the library object but not in the replacement, then the link may fail.

When a symbol that needs to be resolved is found in the library, the linker extracts the object file that defines that symbol and adds it to the other objects used in the link. The extracted object is then subject to the same processing as any other object file specified for the link. If the extracted object has other global symbols that conflict with global symbols in other object files, then the linker reports an error for multiple definitions of a symbol. For example, if the object in the library defines `symbol_a` and `symbol_b` and the object is needed to resolve a reference to `symbol_a`, it is possible that the definition of `symbol_b` can conflict with another definition in the other object files.

Linking Environment for Windows

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

CrossCore Embedded Studio provides an environment that simplifies the processor program build process. From CrossCore Embedded Studio, you specify build options from the **Tool Settings** dialog box and modify files, including the linker description file (`.ldf`). Error and warning messages appear in the **Console** view.

Project Builds

The linker runs from an operating system command line, issued from the IDE or a command prompt window. The IDE provides an intuitive interface for processor programming. When you open CrossCore Embedded Studio, 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.

Within CrossCore Embedded Studio, specify tool settings for project builds. Use the **Settings** dialog box pages to select the target processor, type, and name of the executable file, as well as the CrossCore Embedded Studio toolchain available for the selected processor.

When using the IDE, use the linker pages from the **Tool Settings** dialog box to select and/or set linker functional options.

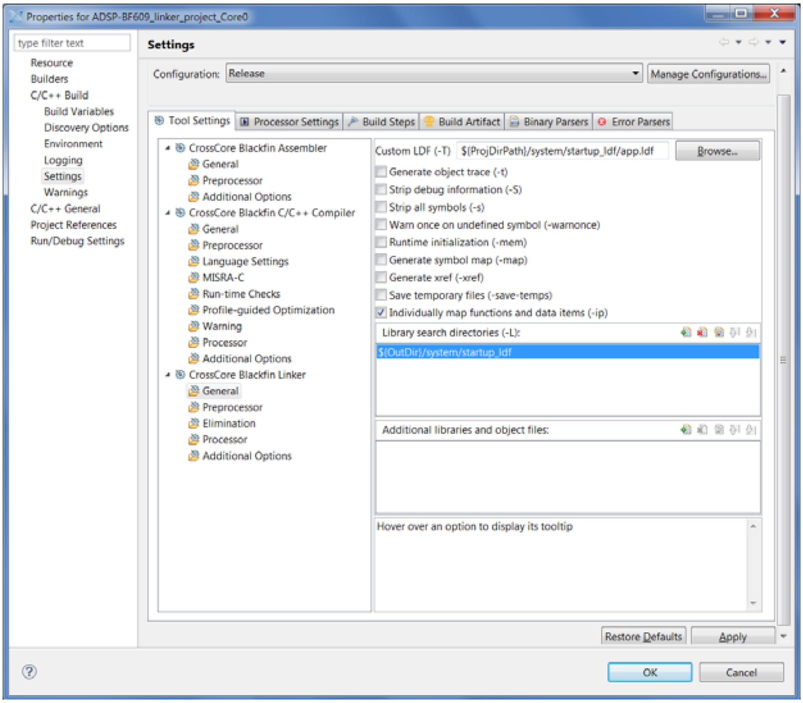

Figure 2-2. Tool Settings Dialog Box: Blackfin Linker: General Page

There are five sub-pages you can access—**General**, **Preprocessor**, **Elimination**, **Processor**, and **Additional Options**. Figure 2-2 shows a sample linker project. Most dialog box options have a corresponding compiler command-line switch as described in "Linker Command-Line Switches" on page 2-44.

Use the **Additional Options** page to enter appropriate file names, switches, and parameters that do not have corresponding controls on the dialog box but are available as compiler switches.

Due to different processor architectures, different linker page options are available. Use context-sensitive online help in CrossCore Embedded Studio to obtain information on dialog box controls (linker options). To do so, click on the "?" button and then click on the field, box, or button for which you need information.

Linker Warning and Error Messages

Linker messages are written to standard output. 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. For example,

```
linker -proc ADSP-unknown a.doj
[Error li1010]    The processor 'ADSP-unknown' is
                  unknown or unsupported.
```

Interpreting Linker Messages

You can find descriptions of linker messages in the online help.

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 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 **All options** box on the linker page of the **Tool Settings** dialog box in the IDE).

Be sure to understand the processor's memory architecture, which is described in the appropriate processor's *Hardware Reference* and in its data sheet.

This section contains:

- "Representing Memory Architecture" on page 2-12
- "Specifying the Memory Map" on page 2-13
- "Placing Code on the Target" on page 2-34
- "Passing Arguments for Simulation or Emulation" on page 2-39

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 2-13.
- **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 3-39.

Specifying the Memory Map

An embedded 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 3-56.

The following topics are important when allocating memory:

- "Memory Usage and Default Memory Segments" on page 2-13
- "Memory Characteristics Overview" on page 2-25
- "Linker MEMORY{} Command in an LDF" on page 2-28

Memory Usage and Default Memory Segments

Input section names are generated automatically by the compiler or are specified in the assembly source code. The `.ldf` file defines memory segment names and output section names. The default .ldf file handles all compiler-generated input sections (refer to the "Input Section" column in Table 2-1 and Table 2-2). 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) to dump contents of an output section (for example, data1) of an object 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 the appropriate target processor.

Refer to your processor's default .ldf file and to the processor's *Hardware Reference* for details. Also see "Wildcard Characters" on page 2-34.

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

- "Default Memory Segments for SHARC Processors"
- "Default Memory Segments for Blackfin Processors"
- "Blackfin Special "Table" Input Sections"

Default Memory Segments for SHARC Processors

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

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

Input Section	Output Section	Memory Segment
seg_pmco	seg_pmco	seg_pmco
seg_dmda	seg_dmda	seg_dmda
seg_pmda	seg_pmda	seg_pmda
seg_rth	seg_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_ctdml	dxe_ctdm	mem_ctdm
seg_vtbl	seg_vtbl	seg_dmda
seg_sram	seg_sram	seg_sram
.bss	.bss	seg_dmda
.gdt .gdtl	seg_dmda	seg_dmda
.frt	seg_dmda	seg_dmda
.cht	seg_dmda	seg_dmda
.edt	seg_dmda	seg_dmda
.rtti	seg_dmda	seg_dmda
For ADSP-213xx/ADSP-214xx Processors Only:		
seg_stak	stackseg	seg_stak
seg_ext_code	seg_ext_code	seg_ext_code
seg_heap	heap	seg_heap
seg_ext_data	seg_ext_data	seg_ext_dmda

Table 2-1. Section Mapping in the Default SHARC LDF File (Cont'd)

Input Section	Output Section	Memory Segment
seg_sdram	seg_sdram_data	seg_ext_dmda
seg_flash	seg_flash	seg_flash
For ADSP-214xx Processors Only:		
seg_ext_code	seg_ext_code	seg_ext_swco
seg_swco	seg_swco	seg_int_code

For more information on stack and heap allocation, see "Memory Usage" in the *C/C++ Compiler Manual for SHARC Processors*. Several input sections and memory segments are used in the default `.ldf` files for ADSP-211xx/212xx/213xx/214xx processors, which must be present in the user's `.ldf` file. These sections are described in detail below.

.bss
This section contains global zero-initialized data. The linker places the contents of this data section in `seg_dmda`.

.rtti
This section is used by the C++ run-time type identification support, when enabled.

seg_rth
This section contains the interrupt vector table. By default, this is located in the start-up file (for example, `160_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 2-55 for more information.)

seg_init_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 contains the addresses of constructors 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 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
```

In the default LDFs for the ADSP-2116x and ADSP-2126x processors, the run-time stack and heap are also allocated from this section.

seg_stak
In the LDFs for ADSP-2136x/2137x and ADSP-2147x/2148x processors, this section is the area where the run-time stack is located. Local variables, function parameters, and so on are stored here.

In the LDFs for ADSP-2116x and ADSP-2126x processors, the run-time stack is located in `seg_dmda`.

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 with the `-section` compiler switch.

seg_sram
This section is SDRAM memory.

seg_heap
In the default LDFs for ADSP-2136x/2137x/2147x/2148x processors, this section is the area from which memory allocation functions and operators (`new`, `malloc()`, and so on) allocate memory.

In the default LDFs for ADSP-2116x and ADSP-2126x processors, the memory allocation functions and operators allocate memory from `seg_dmda`.

In the VDK LDFs, this section is the area from which memory allocation functions and operators allocate memory.

seg_flash
In the LDFs for ADSP-213xx/ADSP-214xx processors, this section is flash memory.

seg_ext_swco
In the LDFs for ADSP-214xx processors, this section is the external memory for sections that contain short-word instructions (using the variable instruction set).

seg_ext_nwco
In the LDFs for ADSP-214xx processors, this section is the external memory for sections that contain normal-word instructions (using the legacy instruction set).

seg_ext_dmda
In the LDFs for ADSP-214xx processors, this section is external memory used for global data qualified with the "`dm`" keyword.

seg_ext_pmda
In the LDFs for ADSP-214xx processors, this section is the external memory for global program data that is qualified with the "`pm`" keyword.

Other Memory Segments

The compiler and libraries also use other data sections that are linked into one of the above memory segments. 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_ctdml`.

.gdt, .gdtl, .frt, .cht, and .edt
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 "Blackfin Special "Table" Input Sections" on page 2-22.

Default Memory Segments for Blackfin Processors

The default `.ldf` files in `Blackfin\ldf` show the mapping of input sections to output sections and memory segments. There are several input sections present in the default `.ldf` file and their uses are detailed below.

See "Linker Description File" in Chapter 3, Linker Description File for more information on `.ldf` files and help on customization. Generation and configuration of a custom .ldf file is available when creating a new project (or via the **Tool Settings** tab in the IDE).

program
This section is the default location for program code.

data1
This section is the default location for global program data.

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

constdata
This section is used for global data that is declared as constant and for literal constants such as strings and array initializers.

cplb_data
This section stores CPLB configuration tables. In particular, the cplbtabx.doj files (where x indicates the target) mapped by .ldf files are placed into this section.

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.

ctor
This section contains addresses of C++ constructor functions which are to be called before `main()` to construct static objects. The mapping of `ctor` must be followed directly by the mapping of `ctor`.

bsz
This section is used to map global zero-initialized data. This section does not actually contain data; it is zero-filled upon loading via the IDE, via a command line, or when processed by the loader.

bsz_init
This section contains run-time initialization data. (See "-meminit" on page 2-55.) 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 is used when 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
In most `.ldf` files and LDF configurations, 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.

sdram0_bank{1|2|3}
This section is used to map code and data into separate SDRAM banks which are defined when SDRAM is partitioned in the default `.ldf` files.

sdram_bsz
This section is the same as section `bsz`, except it is placed in SDRAM when SDRAM is enabled.

sdram_shared
This section is used to map code and data into the part of memory shared between core A and core B on multicore systems.

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 Segments

The compiler and libraries also use other data sections that are linked into one of the above memory segments. 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, .cht, .edt, and .rtti
These data sections are used to hold data used during the handling of exceptions. See "Blackfin Special "Table" Input Sections".

Blackfin Special "Table" Input Sections

The following "table" data sections are used to hold data used during the handling of exceptions. Generally, the linker maps these sections into read-only memory.

.gdt

The `.gdt` 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

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

.edt

The `.edt` section (exception dispatch table) is used by the C++ exception library to map from try blocks to catch blocks.

.cht

The `.cht` 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.

.frt

The `.frt` section (function range table) is used by the C++ exception library during exception processing to unwind the stack of active functions.

primio_atomic_lock

The `primio_atomic_lock` section is used by the control variable that is used to ensure atomic file I/O. It must be in shared memory and not cached.

mc_data

The `mc_data` section is used to hold the core-specific storage on multi-core systems.

.rtti

The `.rtti` section is used by the C++ run-time type identification support, when enabled.

cplb
The `cplb` section is in `.ldf` files for legacy reasons.

Input Sections in Blackfin Default LDFs for User Code/Data

ⓘ These sections are not normally used by the Blackfin compiler and libraries.

L1_data
This section is used to allow global data to be mapped explicitly into L1 data SRAM using the `section` pragma or directive. This input section maps data to both banks A and B where present on the target.

L1_data_a
This section is not normally used by the compiler and libraries.

L1_data_b
This section is similar to `L1_data_a`, except that it is used to map data into L1 data B SRAM where it is present on the target chip.

L1_code
This section is used to allow code to be mapped explicitly into L1 code SRAM using the section pragma or directive.

L1_bcz
This section is used to map global zero initialized data into L1 data SRAM using the section pragma or directive.

L2_bcz
This section is used to map global zero-initialized data to L2 for parts which have L2 memory using the section pragma or directive.

L2_sram
This section can be used to map code and data into L2 for non-multicore parts that have L2 SRAM.

l2_sram
This section can be used on a multicore system to map code and data into L2 for parts which have L2 memory.

L2_sram_a
This section is used to map code and data into the part of L2 memory reserved for core A on a multicore system.

L2_sram_b
This section is used to map code and data into the part of L2 memory reserved for code B on a multicore system.

l2_shared
This section is used to map code and data into the part of L2 memory shared between core A and core B.

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 processor's *Hardware Reference* 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 I/O 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 0.5M bits. Each block can be configured for different combinations of code and data storage. All of the memory can be accessed as 16-, 32-, 48-, or 64-bit words. The memory can be configured in each block as a maximum of 16K words of 32-bit data, 8K words of 64-bit data, 32K words of 16-bit data, 10.67K words of 48-bit instructions (or 40-bit data), or combinations of different word sizes up to 0.5M bits. This gives a total for the complete internal memory: a maximum of 32K words of 32-bit data, 16K words of 64-bit data, 64K words of 16-bit data, and 21K words 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

ADSP-21161 processors have 2M bits of internal memory space; 1M bits are addressable. The 1M bits of memory is divided into two 0.5-M bit blocks: Block 0 and Block 1. The additional 1M bits of the memory space is reserved on the ADSP-21161 processor. Table 2-2 shows the maximum number of data or instruction words that can fit in each 0.5-M bit internal memory block.

Table 2-2. Words Per 0.5-M Bit Internal Memory Block

Word Type	Bits Per Word	Maximum Number of Words Per 0.5-M bit Block
Instruction	48-bits	10.67K words
Long word data	64-bits	8K words
Extended-precision normal word data	40-bits	10.67K words
Normal word data	32-bits	16K words
Short word data	16-bits	32K words

External Memory

Although 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 (DAG). External memory banks are fixed sizes that can be configured for various wait state and access configurations.

The processor can address 254M words of external memory space. 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 2-3 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.

Table 2-3. 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 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.

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 64M words. For non-SDRAM memory, each bank can address up to 16M words. The remaining 48M words are reserved. These reserved addresses for non-SDRAM accesses are aliased to the first 16M spaces within the bank.

Blackfin Memory Characteristics

Details of the Blackfin processor memory characteristics can be found in the data sheets for individual processors, available in the appropriate *Hardware Reference*.

Linker MEMORY{} Command in an LDF

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

Listing 2-1. ADSP-21469 MEMORY{} Command Code

```
MEMORY {

  // The ADSP-21469 SHARC has 5 Mbit RAM and 4 Mbit ROM in total.

  // ------------------- BLOCK 0 ----------------------------
  // 0x0008 C000 to 0x0009 3FFF  Normal word (48) Space
  //                                              (1.5 Mbit RAM)
  // 0x0009 2000 to 0x0009 DFFF  Normal word (32) Space
  //                                              (1.5 Mbit RAM)
  // 0x0012 4000 to 0x0013 BFFF  Short word  (16) Space
  //                                              (1.5 Mbit RAM)
  iv_code
    { TYPE(PM RAM) START(0x0008C000) END(0x0008C0a7) WIDTH(48) }
  seg_int_code
    { TYPE(SW RAM) START(0x001241f8) END(0x0013BFFF) WIDTH(16) }

  // ------------------ BLOCK 1 -----------------------------
  // 0x000A C000 to 0x000B 3FFF  Normal word (48) Space
  //                                              (1.5 Mbit RAM)
  // 0x000B 2000 to 0x000B DFFF  Normal word (32) Space
  //                                              (1.5 Mbit RAM)
  // 0x0016 4000 to 0x0017 BFFF  Short word  (16)  Space
  //                                              (1.5 Mbit RAM)
  seg_dmda
    { TYPE(DM RAM) START(0x000b2000) END(0x000bbfff) WIDTH(32) }
  seg_stak
    { TYPE(DM RAM) START(0x000bc000) END(0x000bdfff) WIDTH(32) }

  // -------------------- BLOCK 2 ---------------------------
```

```
// 0x000C 0000 to 0x000C 5554  Normal word (48) Space
//                                              (1 Mbit RAM)
// 0x000C 0000 to 0x000C 7FFF  Normal word (32) Space
//                                              (1 Mbit RAM)
// 0x0018 0000 to 0x0018 FFFF  Short word (16) Space
//                                             (1 Mbit RAM)
seg_pmda
  { TYPE(PM RAM) START(0x000C0000) END(0x000C7FFF) WIDTH(32) }

// ---------------- BLOCK 3 ------------------------------
// 0x000E 0000 to 0x000E 5554  Normal word (48) Space
//                                              (1 Mbit RAM)
// 0x000E 0000 to 0x000E 7FFF  Normal word (32) Space
//                                              (1 Mbit RAM)
// 0x001C 0000 to 0x001C FFFF  Short word (16)  Space
//                                              (1 Mbit RAM)
seg_heap
  { TYPE(DM RAM) START(0x000e0000) END(0x000e7fff) WIDTH(32) }

// ---------------- SDRAM (DDR2) -------------------------
// 0x0020 0000 to 0x021F FFFF  Short word (16) Space
//                                             (124 MB RAM)
seg_ext_swco
  { TYPE(SW RAM) START(0x00600000) END(0x0065FFFD) WIDTH(16) }
seg_ext_nwco
  { TYPE(PM RAM) START(0x00220000) END(0x005FFFFF) WIDTH(16) }
seg_ext_dmda
  { TYPE(DM RAM) START(0x00900000) END(0x01DFFFFD) WIDTH(16) }
seg_ext_pmda
  { TYPE(DM RAM) START(0x01E00000) END(0x022FFFFF) WIDTH(16) }
seg_flash
  { TYPE(DM RAM) START(0x04000000) END(0x043FFFFF) WIDTH(8) }
} /* MEMORY */
```

Listing 2-2. ADSP-BF609 MEMORY{} Command Code

```
MEMORY {
#if defined(CORE0)
  // Core 0 Memory
  MEM_L1_SCRATCH
    { START(0xFFB00000) END(0xFFB00FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_CODE_CACHE
    { START(0xFFA10000) END(0xFFA13FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_CODE
    { START(0xFFA00000) END(0xFFA0FFFF) TYPE(RAM) WIDTH(8) }

  MEM_L1_DATA_B
    { START(0xFF900000) END(0xFF907FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_DATA_A
    { START(0xFF800000) END(0xFF807FFF) TYPE(RAM) WIDTH(8) }

  MEM_L2_SRAM_ICC
    { START(0xC8080000) END(0xC808041F) TYPE(RAM) WIDTH(8) }
  MEM_L2_SRAM_UNCACHED
    { START(0xC8080420) END(0xC808820F) TYPE(RAM) WIDTH(8) }
  MEM_L2_SRAM
    { START(0xC8090000) END(0xC80A7FFF) TYPE(RAM) WIDTH(8) }

  MEM_SDRAM_BANK0
    { START(0x00000004) END(0x00FFFFFF) TYPE(RAM) WIDTH(8) }
  MEM_SDRAM_BANK1
    { START(0x01000000) END(0x01FFFFFF) TYPE(RAM) WIDTH(8) }
  MEM_SDRAM_BANK2
    { START(0x02000000) END(0x02FFFFFF) TYPE(RAM) WIDTH(8) }
  MEM_SDRAM_BANK3
    { START(0x03000000) END(0x03FFFFFF) TYPE(RAM) WIDTH(8) }
#endif
```

```
#if defined(CORE1)
  // Core 1 Memory
  MEM_L1_SCRATCH
    { START(0xFF700000) END(0xFF700FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_CODE_CACHE
    { START(0xFF610000) END(0xFF613FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_CODE
    { START(0xFF600000) END(0xFF60FFFF) TYPE(RAM) WIDTH(8) }

  MEM_L1_DATA_B
    { START(0xFF500000) END(0xFF507FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_DATA_A
    { START(0xFF400000) END(0xFF407FFF) TYPE(RAM) WIDTH(8) }

  MEM_L2_SRAM_ICC
    { START(0xC8080000) END(0xC808041F) TYPE(RAM) WIDTH(8) }
  MEM_L2_SRAM_UNCACHED
    { START(0xC8088210) END(0xC808FFFF) TYPE(RAM) WIDTH(8) }
  MEM_L2_SRAM
    { START(0xC80A8000) END(0xC80BFFFF) TYPE(RAM) WIDTH(8) }

  MEM_SDRAM_BANK0
    { START(0x04000000) END(0x04FFFFFF) TYPE(RAM) WIDTH(8) }
  MEM_SDRAM_BANK1
    { START(0x05000000) END(0x05FFFFFF) TYPE(RAM) WIDTH(8) }
  MEM_SDRAM_BANK2
    { START(0x06000000) END(0x06FFFFFF) TYPE(RAM) WIDTH(8) }
  MEM_SDRAM_BANK3
    { START(0x07000000) END(0x07FFFFFF) TYPE(RAM) WIDTH(8) }
#endif
} /* MEMORY */
```

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.

Entry Address

The entry address field can be set using:

- The `-entry` command-line switch (on page 2-53), where option's argument is a symbol.
- The `ENTRY(`*`symbol`*`)` command (on page 3-35) in the `.ldf` file. If `-entry` and `ENTRY()` are both present, they must be the same. Neither overrides the other. If there is a mismatch, the linker detects an error.
- In the absence of the `-entry` switch or the `ENTRY()` command, the value of the global file symbol `start`, or LDF symbol `start`, is used, if present.
- If none of the above is used, the address is 0.

Multiprocessor/Multicore Applications

The `-entry` switch for a multiprocessor/multicore `.ldf` file applies the same entry address to all processors. If the entry addresses differ (multiprocessor systems), use `ENTRY()` commands in the `.ldf` file – do not use the `-entry` switch.

If the `-entry` switch is specified, it is an error if any of the processors utilize an `ENTRY()` command with a different specification.

Wildcard Characters

The linker supports the use of wildcards in input section name specifications in the `.ldf` file. The `*` and `?` wildcard characters are provided on input section names so that you can specify multiple input sections.

`*` – Matches any number of characters

`?` – Matches any one character

For information about wildcard characters used (and an example) with the `INPUT_SECTIONS` command, see "INPUT_SECTIONS()" on page 3-59.

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.

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

2. Compare the input sections list to the memory segments specified in the `MEMORY{}` command. Identify the memory segment into which each `.SECTION` must be placed.

3. Combine the information from these two lists to write one or more `SECTIONS{}` commands in the `.ldf` file.

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

Listing 2-3 presents a `SECTIONS{}` command that would work with the `MEMORY{}` command in Listing 2-1.

Listing 2-3. ADSP-21161 SECTIONS{} Command in the LDF

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

Listing 2-4. ADSP-BF533 SECTIONS{} Command in the LDF

```
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 */
```

Linking With Attributes – Overview

Attributes are used within the .ldf file to create virtual subsets from the usual input sources. Attributes are associated with .doj files, including those within the library. Once created, these subsets exist for the duration of the link and can be used anywhere a library or object list normally appears within an .ldf file.

Attributes are used within the .ldf file to reduce the usual set of input files into more manageable subsets. Inputs are in two forms (objects and libraries) both of which appear in lists within the .ldf file. Filters can be applied to these lists to winnow out momentarily-undesirable objects.

An *attribute* is a name/value pair of strings. A valid attribute name is a valid C identifier.

Attribute names and attribute values are case-sensitive. Windows *filenames* can be used as values, with care and consistency.

An attribute is associated with an object (.doj), but not with a library (.dlb), not with a symbol name, and not with an ELF section. An object has zero or more attributes associated with it. A given object may have more than one attribute with the same name associated with it.

Using attributes, the filtering process can be used to remove some objects from consideration, providing that the same objects are not included elsewhere via other filters (or through unfiltered mappings). A filter operation is done with curly braces, and can be used to define sub-lists and sub-libraries. It may also be used in INPUT_SECTIONS commands (refer to "INPUT_SECTIONS()" on page 3-59).

The linker reads the `.ldf` file and uses the `{...}` filter commands (for example, `INPUT_SECTIONS` commands) to eliminate some input objects from consideration before resolving symbols. The linker does not change its behavior if no filter commands are present in the `.ldf` file.

Passing Arguments for Simulation or Emulation

The symbol `_argv_string` is a null-terminated string that, if it contains anything other than null, will be split at each space character and placed in the `argv[]` array that gets passed to the `main` function on system startup.

Linker Command-Line Reference

This section provides reference information, including:

- "Linker Command-Line Syntax" on page 2-39
- "Linker Command-Line Switches" on page 2-44

When you use the linker via the IDE, the settings on the linker pages of the **Tool Settings** tab correspond to linker command-line switches. Provided here is the detailed descriptions of the linker's command-line switches and their syntax. For more information, refer to 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>` to proceed. If the command line does not include `-proc` *`processor`*, the `.ldf` file following the `-T` switch must contain an `ARCHITECTURE()` command. The linker command may contain both, but then the `ARCHITECTURE()` command in the `.ldf` file must match the `-proc` *`processor`*.
>
> Use `-proc` *`processor`* instead of the deprecated `-D`*`architecture`* switch on the command line to select the target processor. See Table 2-5 for more information.

All other switches are optional, and some commands are mutually exclusive.

The following are example linker commands.

```
linker -proc ADSP-21161 p0.doj -T target.ldf -t -o program.dxe
linker -proc ADSP-BF533 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`.

The linker can be controlled by the compiler driver via the `-flags-link` command-line switch, which passes explicit options to the linker. For more information, refer to Chapter 1 of the *C/C++ Compiler and Library Manual* for the appropriate processor.

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

- "Command-Line Object Files" on page 2-41
- "Command-Line File Names" on page 2-42
- "Object File Types" on page 2-43

Command-Line Object Files

The command line must identify 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 CrossCore Embedded Studio. 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

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 from which 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 2-4 lists valid file extensions used by the linker.

Command-Line File Names

Some linker switches take a file name as a parameter. Table 2-4 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 2-4.

Table 2-4. 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\Blackfin\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`) files and library (`.dlb`) files that appear on the command line as object files to be linked. The linker treats executable (`.dxe`) files 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 2-5 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 can 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-BF533 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-BF533`
  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 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 the 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 2-5 briefly describes each linker switch. Each individual switch is described in detail following this table. See "Project Builds" on page 2-9 for information on the CrossCore Embedded Studio **Tool Settings** dialog box.

Table 2-5. Linker Command-Line Switch Summary

Switch	Description	More Info
`@`*`file`*	Uses the specified file as input on the command line	on page 2-48
`-D`*`processorID`*	Specifies the target processor ID. The use of `-proc` *`processorID`* is recommended.	on page 2-48
`-L` *`path`*	Adds the path name to search libraries for objects	on page 2-48

Table 2-5. Linker Command-Line Switch Summary (Cont'd)

Switch	Description	More Info
`-M`	Produces dependencies	on page 2-49
`-MM`	Builds and produces dependencies	on page 2-49
`-Map` *`file`*	Outputs a map of link symbol information to a file	on page 2-49
`-MD`*`macro`*`[=`*`def`*`]`	Defines and assigns value `def` to a preprocessor macro	on page 2-50
`-MUD`*`macro`*	Undefines the preprocessor macro	on page 2-50
`-S`	Omits debugging symbols from the output file	on page 2-50
`-T` *`filename`*	Identifies the LDF to be used	on page 2-51
`-Werror` *`number`*	Promotes the specified warning message to an error	on page 2-51
`-Wwarn` *`number`*	Demotes the specified error message to a warning	on page 2-51
`-W`*`number`*	Selectively disables warnings by one or more message numbers. For example, `-W1010` disables warning message `li1010`.	on page 2-51
`-e`	Eliminates unused symbols from the executable	on page 2-52
`-ek` *`secName`*	Specifies a section name in which elimination should not take place	on page 2-52
`-es` *`secName`*	Names input sections (*`secName`* list) to which the elimination algorithm is applied	on page 2-52
`-ev`	Eliminates unused symbols verbosely	on page 2-53
`-entry`	Specifies entry address where an argument can be either a symbol or an address	on page 2-53
`-flag-meminit`	Passes each comma-separated option to the memory initializer utility	on page 2-53
`-flag-pp`	Passes each comma-separated option to the preprocessor	on page 2-53
`-h` `-help`	Outputs the list of command-line switches and exits	on page 2-53
`-i` *`path`*	Includes search directory for preprocessor `include` files	on page 2-53

Table 2-5. Linker Command-Line Switch Summary (Cont'd)

Switch	Description	More Info
`-ip`	Fills fragmented memory with individual data objects that fit	on page 2-54
`-jcs2l`	Converts out-of-range short calls and jumps to the longer form. It also allows the linker to convert out-of-range branches to indirect calls and jump sequences.	on page 2-54
`-keep` *symName*	Keeps the specified symbol from being eliminated	on page 2-55
`-meminit`	Causes post-processing of the executable file	on page 2-55
`-nomemcheck`	Turns off LDF memory checking	on page 2-56
`-o` *filename*	Outputs the named executable file	on page 2-56
`-od` *filename*	Specifies the output directory	on page 2-56
`-pp`	Stops after preprocessing	on page 2-56
`-proc` *processor*	Selects a target processor	on page 2-56
`-reserve-null`	Directs the linker to reserve 4 addressable units (words) in memory at address 0x0	on page 2-57
`-s`	Strips symbol information from the output file	on page 2-57
`-save-temps`	Saves temporary output files	on page 2-57
`-si-revision` *version*	Specifies silicon revision of the specified processor	on page 2-57
`-sp`	Skips preprocessing	on page 2-58
`-t`	Outputs the names of link objects	on page 2-58
`-tx`	Outputs full names of link objects	on page 2-58
`-v` `-verbose`	Verbose: Outputs status information	on page 2-59
`-version`	Outputs version information and exits	on page 2-59
`-warnonce`	Warns only once for each undefined symbol	on page 2-59
`-xref`	Produces a cross-reference file	on page 2-59

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

@*filename*

The `@` switch causes the linker to treat the contents of *`filename`* 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 the contents of the input file serves to separate tokens.

-D*processor*

The `-D`*`processor`* (define processor) switch specifies the target processor (architecture); for example, `-DADSP-BF533`.

> The `-proc` *`processor`* switch (on page 2-56) 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 installation. This switch (or `-proc` *`processor`* switch) must be used if no `.ldf` file is specified on the command line. (See `-T` on page 2-51.) This switch (or `-proc` *`processor`* 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 a 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.

To add multiple search paths, repeat the switch or specify a list of paths terminated by semicolons (;) with the final semicolon being optional.

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 generate make dependencies 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 checks 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.map.xml`.

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 tool releases, the map file is dependant on particular installations of CrossCore Embedded Studio. Thus, the `.xml` map file can be used only on the machine on which 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.

XSLT is a language for transforming XML documents. CrossCore Embedded Studio includes the following XSLT files for transforming and displaying the XML map files produced by the linker in a browser.

- `System\linker_map_ss1.xsl`
  Does not display symbols that start with a dot. This file is the default.
- `System\linker_map_ss2.xsl`
  Cause all symbols to be displayed.

Note that the compiler and libraries may use symbols that start with a dot for local data and code.

-MD*macro*[=*def*]

The `-MD`*`macro[=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 (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 `-MD`*`macro`* 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 source debugging information from the output file. Compare this switch to the `-s` switch on page 2-57.

-T *filename*

The `-T` *`filename`* (linker description file) switch directs the linker to use *`filename`* as the name of the `.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 CrossCore Embedded Studio support has been installed. In such cases, the processor ID does not appear in the **All options** box of the **Tool Settings** tab on the linker page.

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.

-Werror [*number*]

The `-Werror` switch directs the linker to promote the specified warning message to an error. The *`number`* argument specifies the message to promote.

-Wwarn [*number*]

The `-Wwarn` 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 `-W`*`number`* or `-w`*`number`* (warning suppression) switches selectively disables warnings specified by one or more message numbers.
For example, `-W1010` disables warning message `li1010`. Optionally, this switch 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 3-37): `___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 3-37) 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. In the absence of the `-es` switch or the `ELIMINATE_SECTIONS()` LDF command (on page 3-35), the linker applies elimination to all sections. Both this switch and the `ELIMINATE_SECTIONS()` LDF command 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 3-37): `___ctor_NULL_marker` and `___lib_end_of_heap_descriptions`

-entry

The `-entry` switch indicates the entry address where an argument can be either a symbol or an address.

-ev

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

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

The `-flags-meminit` switch passes each comma-separated option to the memory initializer utility. (For more information, see “Memory Initializer” in Chapter 6, Memory Initializer.)

-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 to the search path for included files.

To add multiple directories, repeat the switch or specify a list of directories terminated by semicolons (;) with the final semicolon being optional.

-ip

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 IDE), 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.

-jcs2l

Used with Blackfin processors only.

The `-jcs2l` (jump/call short to long) switch directs the linker to convert out-of-range calls and jump instructions to a code sequence that will use an indirect jump or call. Because the indirect sequence uses a register `P1`, the expansion will only be applied to instructions that use the `CALL.X` or `JUMP.X` opcodes.

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.

-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-process executable file) switch directs the linker to post-process the `.dxe` file through the memory initializer utility. (For more information, see "Memory Initializer" in Chapter 6, Memory Initializer.) 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 IDE or the boot loader). Refer to "SECTIONS{}" on page 3-56 for more information on section initialization. For information about the __MEMINIT predefined macro, see "__MEMINIT__" on page 3-31.

-nomemcheck

The `-nomemcheck` (memory checking off) switch allows you to turn off memory checking.

-o *filename*

The `-o filename` (output file) switch sets the value of the `$COMMAND_LINE_OUTPUT_FILE` macro which is normally used as a parameter to the LDF `OUTPUT()` command, which specifies the output file name. If no `-o` is present on command line, the `$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. 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) is printed to a file with the same name as the `.ldf` file with an `.is` extension. This file is in the same directory as the `.ldf` file.

-proc *processor*

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

```
linker -proc ADSP-BF533 p0.doj p1.doj p2.doj -o program.dxe
```

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

-reserve-null

The `-reserve-null` switch directs the linker to reserve four addressable units (words) in memory at address 0x0. The switch is useful for C/C++ programs, to avoid allocation of code or data at the 0x0 (NULL pointer) address.

-s

The `-s` (strip 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 linker to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The *`version`* parameter represents a silicon revision of the processor specified by the `-proc` switch (on page 2-56). For example,

```
linker -proc ADSP-BF533 -si-revision 0.1
```

If silicon version “`none`” is used, no errata workarounds are enabled. Specifying silicon version “`any`” enables 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 appropriate for the latest silicon revision are enabled.

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

The linker passes the `-si-revision <silicon version>` switch when invoking another CrossCore Embedded Studio tool, for example when the linker invokes the assembler.

Example:

The Blackfin linker invoked as

```
linker -proc ADSP-BF533 -si-revision 0.1 …
```

invokes the assembler with

```
easmblkfn -proc ADSP-BF533 -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.

-tx

The `-tx` (full trace) switch directs the linker to output the full names of link objects (full directory path) 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

The `-xref` switch directs the linker to produce an XML cross-reference file `xref.xml` in the linker output directory. The XML file can be opened in a web-browser for viewing.

This linker switch is distinct from the `-xref` compiler driver switch.

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.

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 Blackfin Processors" and Appendix D, "LDF Programming Examples for SHARC Processors" for example `.ldf` files for typical systems.

This chapter contains:

The CrossCore Embedded Studio 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 the *Assembler and Preprocessor Manual*.

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

Refer to example DSP programs shipped with CrossCore Embedded Studio 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 *do not* write an `.ldf` file, *do not* import an `.ldf` file into your project, or *do not* have CrossCore Embedded Studio generate an `.ldf` file, the linker links the code using a default `.ldf` file. The default `.ldf` file name appears in the linker's **All options** box in the IDE **Tool Settings** tab of the linker page. 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 CrossCore Embedded Studio 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 2-11 for more information.

Generated LDFs

On the Blackfin and SHARC platforms, the IDE allows you to generate and configure a custom Linker Description File (`.ldf`). This is the quickest and easiest way to customize your `.ldf` files. See online help for more information.

Default LDFs

The name of each `.ldf` file indicates the intended processor (for example, `ADSP-BF531.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
```

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 configurations are selected by defining preprocessor macros on the linker command line. This can be specified from **Tool Settings > Linker > Preprocessor** or directly from the command line.

At the top of the default Blackfin `.ldf` files, you will find documentation on the macros you can use to configure the default `.ldf` files.

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

See "Common Notes on Basic LDF Examples" on page 3-9 for basic information on LDF structure.

See "LDF Programming Examples for Blackfin Processors" on page C-1 and "LDF Programming Examples for SHARC Processors" on page D-1 for code examples for Blackfin and SHARC processors, respectively.

Example 1 – Basic LDF for Blackfin Processors

Listing 3-1 is an example of a basic `.ldf` file for ADSP-BF533 processors (formatted for readability). Note the `MEMORY{}` and `SECTIONS{}` commands and refer to "Common Notes on Basic LDF Examples" on page 3-9. Other LDF examples are provided in "LDF Programming Examples for Blackfin Processors".

Listing 3-1. Example LDF for ADSP-BF533 Processor

```
ARCHITECTURE(ADSP-BF533)
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 default `.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 three 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.
- Defining the `USE_CACHE` macro 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 SDRAM.
- Defining the `USE_SDRAM` macro has the same effect as defining the `USE_CACHE` macro, except that code and data are mapped to the L1 Cache/SRAM areas.

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.

A common user error occurs when cache is enabled despite not having specified `USE_CACHE`. This leads to code or data corruption as cache activity overwrites the contents of SRAM. Therefore, the LDFs use the following "guard symbols":

```
___l1_code_cache
```

```
___l1_data_cache_a
```

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

```
___l1_data_cache_b
```

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 *C/C++ Compiler and Library Manual*, section "*Caching and Memory Protection*".

Example 2 – Basic LDF for SHARC Processors

Listing 3-2 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 "Common Notes on Basic LDF Examples" on page 3-9. Other examples for assembly and C source files are in "LDF Programming Examples for SHARC Processors".

Listing 3-2. 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 CrossCore Embedded Studio 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
```

Common Notes on Basic LDF Examples

In the following description, the `MEMORY{}` and `SECTIONS{}` commands connect the program to the target processor. For syntax information on LDF commands, see "LDF Commands" on page 3-31.

- ARCHITECTURE(ADSP-xxxxx) specifies the target architecture (processor). For example, ARCHITECTURE(ADSP-BF533). The architecture dictates possible memory widths and address ranges, the register set, and other structural information used by the debugger, linker, and loader. The target architecture must be one of those installed with CrossCore Embedded Studio.

- SEARCH_DIR() specifies directory paths searched for libraries and object files (on page 3-55). For example, the argument $ADI_DSP\Blackfin\lib specifies one search directory for Blackfin 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 installed with the tools, $OBJECTS in the SECTIONS() command specifies the object files to be searched for specific input sections.

  As another example, $ADI_DSP expands to the CrossCore Embedded Studio home directory.

- $COMMAND_LINE_OBJECTS (on page 3-27) is an LDF *command-line macro*, which expands in the .ldf file into the list of input files that appears on the command line.

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 order the files are listed in `INPUT_SECTIONS()` commands. As noted above, this order is typically the order listed in `$OBJECTS` (`$COMMAND_LINE_OBJECTS`).

You may customize the `.ldf` file to link objects in any desired order. Instead of using default macros such as `$OBJECTS`, each `INPUT_SECTION` command can have one or more explicit object names.

The following examples are functionally identical:

Example 1:

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

Example 2:

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

- The `MEMORY{}` command (on page 3-39) 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 3-49) generates a single executable file.
- The `OUTPUT()` command (on page 3-49) produces an executable (`.dxe`) file and specifies its file name.

  In the 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 **Tool Settings** tab when the linker is invoked using the IDE).

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

- `SECTIONS{}` (on page 3-56) 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 that the linker uses as an input to resolve the mapping to the appropriate memory segment declared in the `.ldf` file.
  - For example, in 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 File 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

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

A C++ style comment begins with `//` and ends at the end of the line.

For more information on `.ldf` file structure, see:

- "Link Target Description" on page 2-12
- "Placing Code on the Target" on page 2-34

Also see "LDF Programming Examples for Blackfin Processors" on page C-1 and "LDF Programming Examples for SHARC Processors" on page D-1 for code and `.ldf` file structure examples for Blackfin and SHARC processors, respectively.

Command Scoping

The two LDF scopes are *global* and *command* (see Figure 3-1).

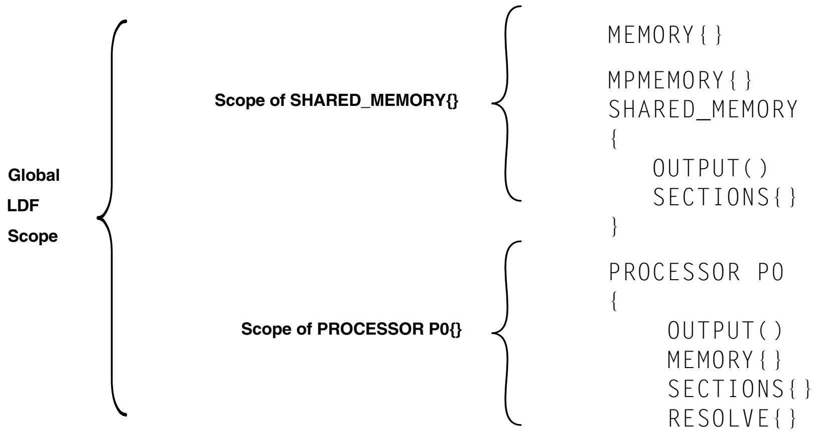

Figure 3-1. LDF Command Scoping Example

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 appears in command scopes is restricted to those scopes.

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, an underscore, or point
    - May contain any letters, underscores, digits, or points
    - Are delimited by white space
    - Do not conflict with any keywords
    - Are unique

Table 3-1 lists valid items used in expressions.

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)

Table 3-1. Valid Items in Expressions (Cont'd)

Convention	Description
*number*k or *number*K	A decimal number multiplied by 1024
B#number or b#number	A binary number

LDF Keywords, Commands, and Operators

Descriptions of LDF keywords, operators, macros, and commands are provided in the following sections.

- "LDF Keywords" on page 3-18
- "Miscellaneous LDF Keywords" on page 3-19
- "LDF Operators" on page 3-19
- "LDF Macros" on page 3-26
- "Built-in Preprocessor Macros" on page 3-29
- "LDF Commands" on page 3-31

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

LDF Keywords

Table 3-2 lists all general LDF keywords (used in Blackfin and SHARC processor families).

Table 3-2. LDF Keywords Summary

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BOOT	COMMON_MEMORY
DEFINED	DYNAMIC	ELIMINATE
ELIMINATE_SECTIONS	ENTRY	END
FALSE	FILL	FIRST_FIT
INCLUDE	INPUT_SECTION_ALIGN	INPUT_SECTIONS
INPUT_SECTIONS_PIN	INPUT_SECTIONS_PIN_ EXCLUSIVE	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
PLIT_SYMBOL_ADDRESS	PLIT_SYMBOL_OVERLAYID	PROCESSOR
RAM	RESERVE	RESOLVE
RESERVE_EXPAND	ROM	
SEARCH_DIR	SECTIONS	SHARED_MEMORY
SHT_NOBITS	SIZE	SIZEOF
START	TYPE	DATA64
VERBOSE	WIDTH	XREF
PM	DM	SW

Miscellaneous LDF Keywords

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

Table 3-3. Miscellaneous LDF 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" on page 2-59.

For more information about other LDF keywords, see "LDF Operators" on page 3-19, "LDF Macros" on page 3-26, and "LDF Commands" on page 3-31.

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_code` 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(dxe_dmda)
```

The linker generates the constant `ldf_start_dmda` and assigns it the start address of the `dxe_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);
```

EXECUTABLE_NAME() Operator

Syntax:

```
EXECUTABLE_NAME(symbol_name)
```

The `EXECUTABLE_NAME()` command can appear in any output section that is mapped to data memory. The effect of the command is to create a local variable with the name specified in the command. The contents of the variable is a null-terminated C string that contains the name of the `.dxe` file that is produced by the linker. This feature can be useful for users, but is necessary for Profile-Guided Optimization support in the compiler. (Refer to the *C/C++ Compiler and Library Manual*, section *"Using Profile-Guided Optimization"*.)

The `EXECUTABLE_NAME()` command is case sensitive. The `symbol_name` argument provides the name of the variable where the string with the executable name is stored.

To create the data with the string, the linker creates an assembly code file, uses the appropriate assembler to generate an object file that is then added to the link. The assembly and object file are saved in the same location as the target executable. For example, building a Debug configuration using the IDE produces <target>.dxe in the Debug folder of the project. The EXECUTABLE_NAME() command leaves a <target>.dxe.asm and <target>.dxe.doj in the same folder.

MEMORY_END() Operator

Syntax:

MEMORY_END(*segment_name*)

This operator returns the end address (the address of the last word) of the named memory segment.

Example:

This example reserves six words at the end of a mem_stack memory segment using the MEMORY_END operator.

```
RESERVE(reserved_space = MEMORY_END(mem_stack) - 6 + 1,
reserved_space_length = 6)
```

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.

MEMORY_START() Operator

Syntax:

MEMORY_START(*segment_name*)

This operator returns the start address (the address of the first word) of the named memory segment.

Example:

This example reserves four words at the start of a mem_stack memory segment using the MEMORY_START operator:

```
RESERVE(reserved_space =
MEMORY_START(mem_stack), reserved_space_length = 4)
```

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 code 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
```

Blackfin Code Example:

The following code 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 previous linker command. 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.
- Do not allow the location counter to 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 3-15

and “LDF Macros and Command-Line Interaction” on page 3-29.

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 (see “Built-in Preprocessor Macros” on page 3-29) 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 *C/C++ Compiler and Library Manual* for appropriate target processor and the *Assembler and Preprocessor Manual*.

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 IDE **Tool Settings** tab. Use this macro only once in your LDF for file name substitution within an `OUTPUT()` command.

- `$COMMAND_LINE_OUTPUT_DIRECTORY`

  This macro expands into the path of the output directory, which is set with the linker's `-od` switch (or `-o` switch when `-od` is not specified).

  For example, the following statement permits a configuration change (release vs. debug) without modifying the `.ldf` file.

  ```
  OVERLAY_OUTPUT($COMMAND_LINE_OUTPUT_DIRECTORY/OVL1.ovl)
  ```

- `$ADI_DSP`

  This macro expands into the path of the 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 IDE or explicitly from a command window. Many linker operations, such as input and output, are controlled through command-line entries. Use LDF macros to apply command-line inputs within the `.ldf` file.

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` file 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` file that can control complex linker features.

Built-in Preprocessor Macros

The linker's preprocessor defines a number of macros to provide information about the linker. These macros can be tested, using the `#ifdef` and related directives, to support your program's needs.

This section provides information about the following built-in preprocessor macros.

- __CCESVERSION__
- __VERSIONNUM__
- __VERSION__
- __SILICON_REVISION__
- __MEMINIT__

__CCESVERSION__

The `__CCESVERSION__` predefined macro provides CrossCore Embedded Studio product version information. The macro allows a pre-processing check to be placed within the `.ldf` file. It can be used to differentiate between releases and updates. This macro applies to all Analog Devices processors.

Syntax:

`__CCESVERSION__=0x`*`MMmmUUxx`*

Table 3-4 explains the parameters of this macro.

Table 3-4. __CCESVERSION__ Macro Parameters

Parameter	Description
MM	VersionMajor. The major release number; for example, 4 in release 4.5.
mm	VersionMinor. The minor release number; for example, 5 in release 4.5.
UU	VersionPatch. The number of the release update, such as version 4.5, update 6.
xx	Reserved for future use (always 00 initially)

__VERSIONNUM__

The `__VERSIONNUM__` predefined macro provides linker version information in hex form. The macro allows a pre-processing check to be placed within the `.ldf` file. It can be used to differentiate between linker versions. This macro applies to all Analog Devices processors.

In other words, this macro defines `__VERSIONNUM__` as a numeric variant of `__VERSION__` constructed from the version number of the linker. Eight bits are used for each component in the version number and the most significant byte of the value represents the most significant version component.

For example, a linker with version 3.6.0.0 defines `__VERSIONNUM__` as `0x03060000` and 3.6.2.10 would define `__VERSIONNUM__` to be `0x0306020A`.

__VERSION__

The __VERSION__ predefined macro provides linker version information in string form, giving the version number of the linker. The macro allows a pre-processing check to be placed within the .ldf file. It can be used to differentiate between linker versions. This macro applies to all Analog Devices processors.

For example, for linker version 3.9.1.1, the value of the macro would be 3.9.1.1.

__SILICON_REVISION__

The __SILICON_REVISION__ predefined macro value is defined by the -si-revision version switch.

For example, if the silicon revision switch (-si-revision) is set to "any", the __SILICON_REVISION__ macro is set to 0xffff. If the -si-revision switch is set to "none", the linker does not set the __SILICON_REVISION__ macro.

__MEMINIT__

The __MEMINIT__ predefined macro is defined if the -meminit switch is used on the command line.

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 3-15.

> ⓘ The linker supports the use of wildcards in section name specifications in the `.ldf` file. The `*` and `?` are provided on input section names.

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

- "ALIGN()" on page 3-33
- "ARCHITECTURE()" on page 3-33
- "COMMON_MEMORY{}" on page 3-34
- "ELIMINATE()" on page 3-34
- "ELIMINATE_SECTIONS()" on page 3-35
- "INCLUDE()" on page 3-35
- "INPUT_SECTION_ALIGN()" on page 3-36
- "KEEP()" on page 3-37
- "KEEP_SECTIONS()" on page 3-37
- "LINK_AGAINST()" on page 3-38
- "MEMORY{}" on page 3-39
- "MPMEMORY{}" on page 3-42
- "OVERLAY_GROUP{}" on page 3-43
- "PACKING()" on page 3-43
- "PLIT{}" on page 3-49
- "PROCESSOR{}" on page 3-49
- "RESERVE()" on page 3-51

- "RESERVE_EXPAND()" on page 3-53
- "RESOLVE()" on page 3-54
- "SEARCH_DIR()" on page 3-55
- "SECTIONS{}" on page 3-56
- "SHARED_MEMORY{}" on page 3-67

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, a valid entry is `ADSP-BF533`. Thus, `ADSP-BF533` is valid, but `adsp-BF533` 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 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.

COMMON_MEMORY{}

The `COMMON_MEMORY{}` command is used to map objects into memory that is shared by more than one processor. The mapping is done in the context of the processors that will use the shared memory; these processors are identified as a "master" of the common memory.

For detailed command description, refer to "COMMON_MEMORY{}" on page 4-50.

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 2-52).

When using either the linker's data elimination feature (via the command-line switches) or the `ELIMINATE()` command in an `.ldf` file, it is essential that certain objects 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 3-37):
>
> `___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 *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 2-52) may be used to specify sections where unreferenced code and data should be eliminated.

ENTRY()

The `ENTRY(`*`symbol`*`)` command specifies the entry address. The entry address is usually filled from a global symbol "`start`" (no underscore), if present. Refer to "Entry Address" on page 2-33 for more information.

Both this LDF command and the linker's `-entry` command-line switch (on page 2-53) may be used to specify the entry address.

INCLUDE()

The `INCLUDE()` command specifies additional `.ldf` files that the linker processes before processing the remainder of the current `.ldf` file. 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` files 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 that is a multiple of *`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 3-56.

The `INPUT_SECTION_ALIGN()` command is valid only within the scope of an output section. For more information, see "Command Scoping" on page 3-15. For more information on output sections, see the syntax description for "SECTIONS{}" on page 3-56.

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* double-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` switch (on page 2-52).

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, a `LINK_AGAINST()` command must be present 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.

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 3-54), 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 the file name. Place this command anywhere in the `.ldf` file.

The `MAP(filename)` command corresponds to (and may be overridden by) the linker's `-Map <filename>` command-line switch (on page 2-49). If the project specifies the generation of a symbol map (**Generate symbol map (-map)** option on the **General** linker page of the **Tool Settings** tab), the linker runs with `-Map <projectname>.xml` asserted and the `.ldf` file'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` file 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 3-15.

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*.

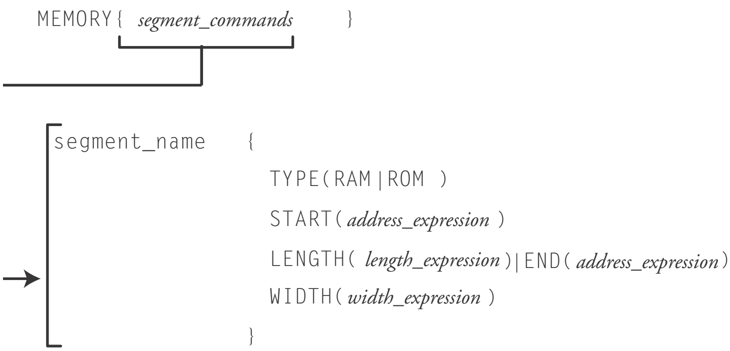

Figure 3-2. MEMORY{} Command Syntax Tree

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()`. The parts of a segment declaration are described below.

segment_name

The *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)

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

TYPE()

The TYPE() command identifies the architecture-specific type of memory within the memory segment.

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

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

For 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).

On ADSP-21261/2/6/7 and ADSP-21362/3/4/5/6 processors, it is not possible to access external memory directly, but through DMA. To validate placement of code accessible through DMA in external memory, use the DMAONLY segment qualifier to mark a memory segment in the .ldf file as external memory. For example,

```
seg_dmda {
         TYPE(DM DMAONLY)
         START(0x00200000)
         END(0x3FFFFFFF)
         WIDTH(32)
         }
<…>
seg_dmda{INPUT_SECTIONS( $OBJECTS(seg_extm) )}
> seg_dmda
```

The linker identifies the section as dmaonly. At link time, the linker verifies that the section must reside in external memory identified with the DMAONLY qualifier. More importantly, the linker checks that only sections marked `dmaonly` are placed in external memory. The linker issues an error if there is any inconsistency between memory the section is mapped to and that section's qualifier:

```
[Error el2017]  Invalid/missing memory qualifier for memory 'sec-
tion name.
```

LENGTH(length_number)/END(address_number)

The `LENGTH/END()` command 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. When you state the end address, *`address_number`* is an absolute address.

WIDTH(width_number)

The `WIDTH()` command 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. The parameters are:

- For Blackfin processors, width must be 8 (bits)
- 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 4-43 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 4-27. Refer to "Memory Management Using Overlays" on page 4-3 for a detailed description of overlay functionality.

PACKING()

The `PACKING()` command is used with ADSP-21xxx (SHARC) processors only (as described in "Packing in SHARC Processors" on page 3-45).

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 `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.

CrossCore Embedded Studio comes with the `packing.h` file in the `...\include` folder. This file provides macros that define packing commands for use in an 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 3-5) could apply to a DMA mode.

Table 3-5. DMA Packing Order

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

The order of unpacked bytes does **not** match the transfer (stored) order. Because the processor uses two bytes per short word, the above transfer translates into the format in Table 3-6.

Table 3-6. Storage Order vs. Unpacked Order

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

You specify to the linker how to accommodate 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 CrossCore Embedded Studio.

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 3-7 indicates packing format combinations for SHARC DMA overlays available under each of the two operations.

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

Table 3-7. 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 3-8. 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 processors that require packed memory for external execution are the ADSP-21161N chips. The ADSP-21161N processor supports 48-, 32-, 16-, and 8-bit-wide external memory.

In order for CrossCore Embedded Studio to execute packing directly from external memory on 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-21161N processors – 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 4-32 for a detailed description of the PLIT{} command. Refer to “Memory Management Using Overlays” on page 4-3 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 3-16.

- `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 3-15 and "MEMORY{}" on page 3-39.

- `PLIT{`*plit_commands*`}`

  Defines procedure linkage table (PLIT) commands that apply only to this specific processor. For more information, see "PLIT{}" on page 3-49.

- `SECTIONS{`*section_commands*`}`

  Defines sections for placement within the executable (`.dxe`) file. For more information, see "SECTIONS{}" on page 3-56.

- `RESOLVE{symbol, resolver}`

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

Multiprocessor/Multicore Applications

The `PROCESSOR{}` command may be used in linking projects on multiprocessor/multicore Blackfin architectures such as the ADSP-BF561 processor. For example, the command syntax for two-processor system is as follows:

```
PROCESSOR p0 {
...
}
PROCESSOR p1 {
}
```

See also "LINK_AGAINST()" on page 3-38, "MPMEMORY{}" on page 4-43, "COMMON_MEMORY{}" on page 4-50, and "SHARED_MEMORY{}" on page 4-44.

RESERVE()

The `RESERVE (start_symbol, length_symbol, min_size [,align])` command allocates address space and defines symbols *`start_symbol`* and *`length_symbol`*. The command allocates the largest free memory block available, larger than or equal to *`min_size`*. Given an optional parameter `align`, `RESERVE` allocates aligned address space.

Input:

- The *`min_size`* parameter defines a required minimum size of memory to allocate.
- The *`align`* parameter is optional and defines alignment of allocated address space.

Output:

- The *start_symbol* is assigned the starting address of the allocated address space.
- The *length_symbol* is assigned the size of the allocated address space.

A user may restrict the command by defining the `start` and `length` symbols together or individually. For example,

```
RESERVE (start_symbol = address, length_symbol, min_size)
RESERVE (start_symbol = address, length_symbol = size)
RESERVE (start_symbol, length_symbol = size [,align])
```

The `RESERVE()` command is valid only within the scope of an output section. For more information on output sections, see "Command Scoping" on page 3-15 and "SECTIONS{}" on page 3-56. Also see "RESERVE_EXPAND()" on page 3-53 for more information on how to claim any unused memory after input sections have been mapped.

Linker Error Resolutions

Linker error `li1224`:
When a user defines `length_symbol`, the `min_size` parameter is redundant and not included in the command. When a user defines `start_symbol`, the `align` parameter is redundant and not included in the command.

Linker errors `li1221`, `li1222`, and `li1223`:
When a user defines `start_symbol = address`, the `align` parameter is redundant and should not be included in the command.

When a user defines `align` parameter, the `length_symbol` or `min_size` parameter should be divisible by `align`; the `align` parameter must be a power of 2.

Given the `start_symbol` is not restricted (not defined), `RESERVE` allocates address space, starting from a segment end address.

Example

Consider an example where given memory segment [0 - 8]. Range [0 - 2] is used by an input section. To allocate address space of minimum size 4 and aligned by 2, the `RESERVE` command has minimum length requirement of 4 and alignment 2.

```
M0 {START(0), END(8), WIDTH(1)}
```

```
out{… RESERVE(start, length, 4, 2) } >M0
```

1. Allocate 4 words {5, 6, 7, 8},

   ```
   start = 5
   ```

   ```
   length = 4
   ```

2. To satisfy alignment by 2, allocate address space {4, 5, 6, 7, 8}

   ```
   start = 4
   ```

   ```
   length = 5
   ```

3. Consider length exactly 4 (not minimum 4). Allocated address space is {4, 5, 6, 7}. Address [8] is freed.

   ```
   start = 4
   ```

   ```
   length = 4
   ```

RESERVE_EXPAND()

The `RESERVE_EXPAND(`*`start_symbol, length_symbol, min_size`*`)` command may follow a `RESERVE` command and is used to define the same symbols as `RESERVE`. Ordinarily, `RESERVE_EXPAND` is specified last in an output section to claim any unused memory after input sections have been mapped. `RESERVE_EXPAND` attempts to allocate memory adjacent to the range allocated by `RESERVE`. Accordingly, *`start_symbol`* and *`length_symbol`* are redefined to include expanded address range. Refer to

"RESERVE()" on page 3-51 for more information.

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 "LINK_AGAINST()" on page 3-38 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 symbol's definition. For example,

```
RESOLVE(start, 0xFFA00000)
```

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 variable by prefixing the variable with an underscore in the `RESOLVE()` command (for example, `_symbol_name`).

Potential Problem with Symbol Definition

Assume the symbol used in the `RESOLVE()` command is defined in the link project. The linker will use that definition from the link project rather one from the *`symbol_name`*, *`resolver`*`)` (also known as "resolve-against") link project specified in the `RESOLVE()` command. For example,

```
RESOLVE(_main, p1.dxe) linker -T a.ldf -Map a.map -o
.\Debug\a.dxe
```

The linker then issues the following message:

```
[Warning li2143] "a.ldf":12 Symbol '_main' used in
resolve-against command is defined in processor 'p0'.
```

If you want to use a local definition, remove the `RESOLVE()` command. Otherwise, remove the definition of the symbol from the link project.

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 (;). On Windows, enclose long directory names with embedded spaces 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 2-48). 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-Blackfin)
MAP (SINGLE-PROCESSOR.XML)        // Generate a MAP file

SEARCH_DIR( $ADI_DSP\Blackfin\lib; ABC/XYZ )
  // $ADI_DSP is a predefined linker macro that expands
  // to the CrossCore Embedded Studio install directory. Search
for objects
  // in directory Blackfin\lib relative to the install directory
  // and to the ABC/XYZ directory.
```

SECTIONS{}

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

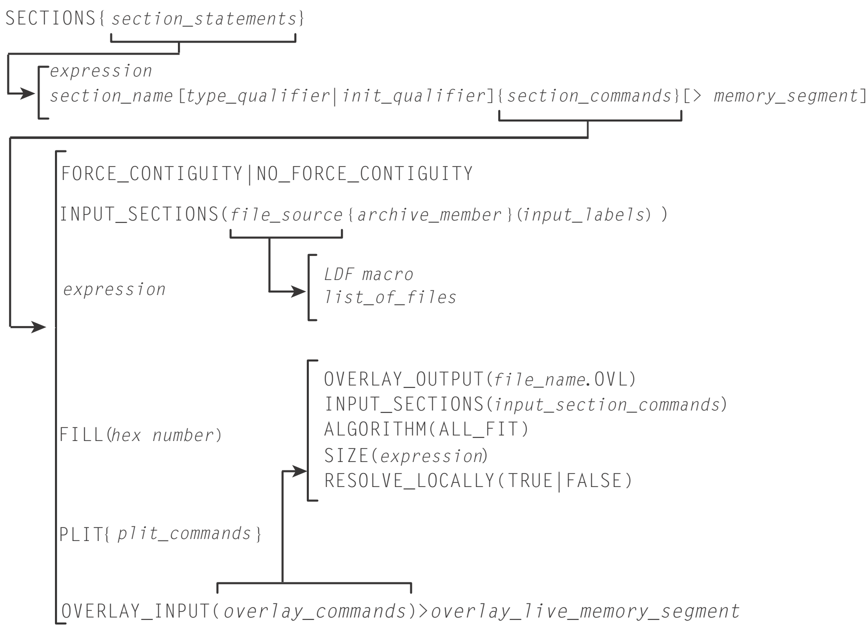

Figure 3-4. SECTIONS{} Command Syntax Tree

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 3-16.

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

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.

- *`type_qualifier`*
  Specifies the address space into which the section should be mapped and the logical organization of the data. Note that this qualifier applies only to SHARC ADSP-2146x/2147x/2148x processors.

  The qualifiers are:

  - `PM` – Program memory, contains 6 bytes per word.
  - `DM` – Data memory, contains 4 bytes per word.

- `DATA64` – Contains 8 bytes per word.
- `SW` – Contains 2 bytes per word.

The output section memory type supersedes the memory type that the section is mapped into. If the output section memory type differs from the segment type, an additional ELF section is created in the output. This ELF section contains the output section and defines its contents.

The use of an output section qualifier also instructs the linker to ignore input sections whose memory type is different than specified by the qualifier. All ignored input sections from a particular mapping command are listed in the linker log file.

- *`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 2-55), 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 commands and/or expressions, such as:

  "INPUT_SECTIONS()" on page 3-59
  "expression" on page 3-64
  "FILL(hex number)" on page 3-65
  "PLIT{plit_commands}" on page 3-65
  "OVERLAY_INPUT{overlay_commands}" on page 3-65
  "FORCE_CONTIGUITY/NOFORCE_CONTIGUITY" on page 3-67

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

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*. Optionally, you may also specify a filter *expr*. 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 spaces. The `*` and `?` wildcard characters can be used to place multiple section names from an object in a library. For more information about wildcard characters, see "Wildcard Characters" on page 2-34.

Example:

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

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

To use a wildcard character that places all sections with a prefix of "`data`" of the object "`foo.doj`" in the library "`myLib.dlb`":

```
INPUT_SECTIONS(myLib.dlb [ foo.doj (data*) ])
```

Using an Optional Filter Expression

The filter operation is done with curly braces, and can be used to define sub-lists and sub-libraries. It can be used for linking with attributes.

```
INPUT_SECTIONS( $FILES { expr } (program) )
```

The optional filter `expr` is a Boolean expression that may contain:

- Attribute operators:
  - *`name`*
    Returns *`true`* if the object has one or more attributes called *`name`*, regardless of value; otherwise, returns *`false`*.
  - *`name("string")`*
    Returns *`true`* if the attribute *`name`* has a value that matches *`string`*. The comparison is case-sensitive *`string`*. This operator may be used on multi-valued attributes. Note that *`string`* must be quoted.
  - *`name cmp-op "string"`*
    Returns *`true`* if the attribute *`name`* has a single value that matches *`string`*, according to *`cmp-op`*. Otherwise, returns *`false`*. *`Cmp-op`* can be “==” or “`!=`”, for equality and inequality, via case-sensitive string comparison. Note that *`string`* must be quoted. This operator may only be used on single-valued attributes. If the attribute does not have exactly one value, the linker generates an error.
  - *`name cmp-op number`*
    Returns *`true`* if the attribute *`name`* has a single value that numerically matches integer number (which can be negative). Otherwise, returns *`false`*. *`Cmp-op`* can be “==”, “`!=`”, “<”, “<=”, “>” or “>=”. This operator may only be used on single-valued attributes. If the attribute does not have exactly one value, the linker generates an error.
- Logical operators: “`&&`”, “`||`”, and “`!`”, having the usual C meanings and precedence.
- Parentheses, for grouping: “`(`" and “`)`”

Example:

```
$OBJS_1_and_2 = $OBJS {attr1 && attr2 };
$OBJS_3_and_2 = $OBJS { attr3("value3") && attr2 == "value2" };

   Outsec {
      INPUT_SECTIONS($OBJS_1_and_2(program))
      INPUT_SECTIONS($OBJS_3_and_2(program))
      INPUT_SECTIONS($OBJS_2 { attr2 } (program))
   } >mem
```

INPUT_SECTIONS_PIN/_PIN_EXCLUSIVE Commands

The INPUT_SECTIONS_PIN and INPUT_SECTIONS_PIN_EXCLUSIVE commands are used to allow mapping of an input section in one of several output sections, as in "*one input section to many output section*" linker feature. For example,

```
os_mem1 {
   INPUT_SECTIONS($OBJECTS(program))
} > mem1

os_mem2 {
   INPUT_SECTIONS($OBJECTS(program))
} > mem2
```

In the above example, if some of the input sections included in $OBJECTS(program) do not fit in os_mem1, the linker will try to map them into os_mem2.

An input section listed in an INPUT_SECTIONS_PIN() command will not be mapped by any INPUT_SECTIONS commands that appear later in the .ldf file, and an input section listed in INPUT_SECTIONS_PIN_EXCLUSIVE command(s) will not be mapped by any other INPUT_SECTIONS command.

Each time an input sections is mentioned in an INPUT_SECTIONS command, the linker is instructed to "give another chance" to the input section by trying to map it in different output section (given the section has not been already mapped), thus achieving the effect of "one-to-many" mapping.

The INPUT_SECTIONS_PIN() and INPUT_SECTIONS_PIN_EXCLUSIVE() commands limit the effect of "one-to-many" mapping – once the input section is mentioned inside INPUT_SECTIONS_PIN(), the linker will not map it in any of the following output sections; an input section mentioned inside INPUT_SECTIONS_PIN_EXCLUSIVE() command can not be mapped in any other output section.

The commands help to avoid breaking existing LDF macros. To achieve the same affect without using INPUT_SECTIONS_PIN and INPUT_SECTIONS_PIN_EXCLUSIVE commands, the definition of the output sections would have be:

```
os_mem1 {
   INPUT_SECTIONS(b.doj(program))
   INPUT_SECTIONS(c.doj(program) d.doj(program))
} > mem1

os_mem2 {
   INPUT_SECTIONS(c.doj(program) d.doj(program))
   INPUT_SECTIONS(a.doj(program))
} > mem2
```

Without the use of general LDF macros and INPUT_SECTIONS_PIN commands, the .ldf file will have to change every time the list of objects changes.

If the same section is mentioned in more than one of INPUT_SECTIONS_PIN() commands, linker will honor the first command only.

In conjunction with attribute expressions, the commands can be used to control the order of input section placement without explicitly mentioning the object files.

```
os_internal {
   INPUT_SECTIONS_PIN($OBJECTS{high_priority}(program))
   INPUT_SECTIONS($OBJECTS(program))
} > mem_internal

os_external {
   INPUT_SECTIONS($OBJECTS(program))
   INPUT_SECTIONS_EXCLUSIVE($OBJECTS{low_priority}(program))
} > mem_external
```

In the above example,

- "`program`" input sections from input files marked with "`high_priority`" attribute can be mapped to "`mem_internal`" only
- "`program`" input sections from input files marked with "`low_priority`" attribute can be mapped to "`mem_external`" only
- All other "`program`" input section can be mapped to "`mem_internal`" or "`mem_external`"

expression

In a *`section_command`*, an *`expression`* manipulates symbols or positions the current location counter. See "LDF Expressions" on page 3-16 for details.

FILL(hex number)

In a *section_command*, the `FILL()` command specifies the hexadecimal number that the linker uses to fill gaps (created by aligning or advancing the current location counter).

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 4-32.

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 4-3. For overlay code examples, see the examples that came bundled with the development software.

The *overlay_commands* item consists of at least one of the following commands: `INPUT_SECTIONS()`, `OVERLAY_ID()`, `NUMBER_OF_OVERLAYS()`, `OVERLAY_OUTPUT()`, `ALGORITHM()`, 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 3-59.

- `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/NOFORCE_CONTIGUITY

In a *`section_command`*, the `FORCE_CONTIGUITY` command forces contiguous placement of the output section. The `NOFORCE_CONTIGUITY` command suppresses a linker warning about non-contiguous placement in the output section.

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/multi-core system. The `SHARED_MEMORY{}` command is used to produce `.sm` files.

For more information, see "SHARED_MEMORY{}" on page 4-44.

4 MEMORY OVERLAYS AND ADVANCED LDF COMMANDS

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

This chapter includes:

This chapter generally uses code examples for Blackfin processors. If used, other processor's code examples are marked accordingly.

Overview

Analog Devices processors generally have a hierarchy of memory. The fastest memory is the "internal" memory that is integrated with the processor on the same chip. For some processors, like Blackfin processors, there are two levels of internal memory (L1 and L2), with L1 memory being faster than L2 memory. Users can configure their system to include "external" memory, usually SDRAM or ROM that is connected to the part.

Ideally, a program can fit in internal memory for optimal performance. Large programs need to be expanded to use external memory. When that happens, accessing code and data in slower memory can affect program performance.

One way to address performance issues is to partition the program so that time-critical memory accesses are done using internal memory while parts of the program that are not time-critical can be placed in external memory. The placement of [program] sections into specific memory sections can be done using `MEMORY{}` and `SECTION{}` commands in the `.ldf` file.

Another way to address performance issues is via memory architecture. Some memory architectures, for example, Blackfin architecture, have instruction and data cache. The processor can be configured to bring instructions and data into faster memory for fast processing.

The third way to optimize performance is to use overlays. In an overlay system, code and data in slower memory is moved into faster memory when it is to be used. For architectures without cache, this method is the only way to run large parts of the program from fast internal memory. Even on processors with cache support, you may want to use overlays to have direct control of what is placed in internal memory for more deterministic behavior.

The overlay manager is a user-defined function responsible for ensuring that a required symbol (function or data) within an overlay is in run-time memory when it is needed. The transfer usually 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 4-4 and "Overlay Managers" on page 4-6.

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. Several applications notes (EE-Notes) on the Analog Devices Web site describe this technique in detail.

This section describes the use of memory overlays. The topics are:

The following LDF commands facilitate overlay features.

- "OVERLAY_INPUT{overlay_commands}" on page 3-65
- "PLIT{}" on page 4-32

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" (be stored) in unique locations in external memory, but "run" (execute) in a common location in internal memory. Throughout the following description, the overlay storage location is referred to as the "live" location, and the internal location where instructions are executed is referred to as the "run" (run-time) space.

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

Figure 4-1 demonstrates the concept of memory overlays. The two memory spaces are: *internal* and *external*. The *external* memory is partitioned into the live space for four overlays. The *internal* memory contains the main program, an overlay manager function, and two memory segments reserved for execution of overlay program instructions (run space).

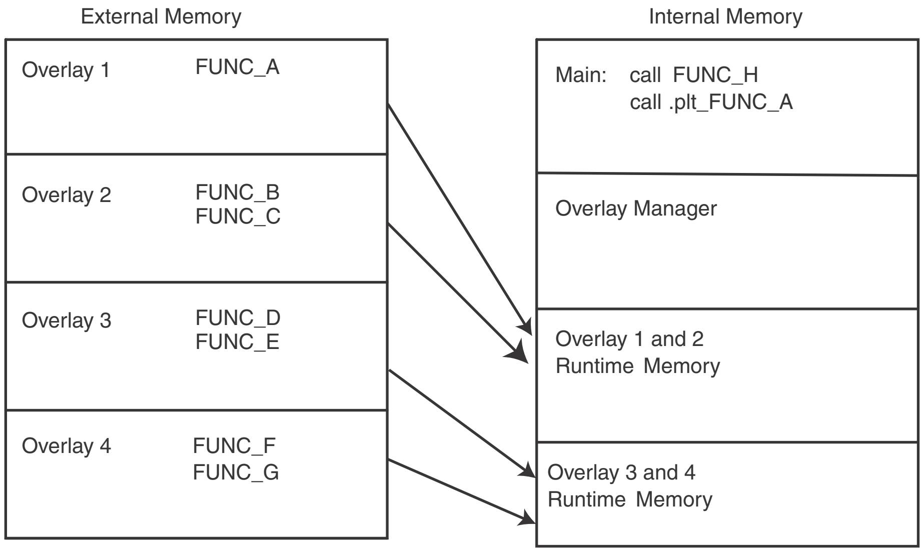

Figure 4-1. Memory Overlays

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

The transfer is typically implemented 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 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. The overlay `PLIT{}` command is typically written to 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 are application-specific and must be developed by the user.

Breakpoints on Overlays

The debugger relies on the presence of the `_ov_start` and `_ov_end` symbols to support breakpoints on overlays. These symbols should appear in the user's overlay manager for debugger support of 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 passes through this location once an overlay is 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 resumes execution.

The second breakpoint is at symbol `_ov_start`. The label `_ov_start` is 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 overlay "live" memory is "in flux". The debugger still functions without this breakpoint, but there may be 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 4-1 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_INPUT{overlay_commands}" on page 3-65.

In this code example, OVLY_one contains FUNC_A and lives in memory segment ovl_live; OVLY_two contains functions FUNC_B and FUNC_C and also lives in memory segment ovl_live.

Listing 4-1. Overlay Declaration in an LDF

```
.dxe_code
  { OVERLAY_INPUT {
      OVERLAY_OUTPUT (OVLY_one.ovl)
      INPUT_SECTIONS (FUNC_A.doj(program))
      } >ovl_live

    OVERLAY_INPUT {
      OVERLAY_OUTPUT (OVLY_two.ovl)
      INPUT_SECTIONS (FUNC_B.doj(program) FUNC_C.doj(sec_code))
      } >ovl_live
  } >ovl_run
```

The common run-time location shared by overlays OVLY_one and OVLY_two is within the ovl_run 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.

The `_ov_word_size_run_#` and `_ov_word_size_live_#` constants are both in terms of words, `_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.

Data packing only applies to SHARC processors.

The Blackfin processor architecture supports byte addressing that uses 16-, 32-, or 64-bit opcodes. Thus, no data packing is required.

Redirection
In addition to providing constants, the linker replaces overlay symbol references to the overlay manager within your code. Redirection is accomplished by means of a *procedure linkage table* (PLIT), which 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:

```
CALL FUNC_A;;           /* Call to function in overlay */
```

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

```
CALL .plt_FUNC_A;       / * Call to PLIT entry */
```

`.plt_FUNC_A` is the entry in the PLIT that contains defined instructions. These instructions prepare the overlay manager to load the overlay containing `FUNC_A`. The instructions executed in the PLIT are specified within the `.ldf` file. The user must supply the PLIT code to match the overlay manager.

Listing 4-2 is an example PLIT definition from an `.ldf` file, where register `R0` is set to the value of the overlay ID that contains the referenced symbol and register `R1` is set to the run-time address of the referenced symbol. The last instruction branches to the overlay manager that uses the initialized registers to determine which overlay to load (and where to jump to execute the called overlay function).

Listing 4-2. PLIT Definitions in LDF

```
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 4-2. The redirection function calls the PLIT table for overlays 1 and 2 (Figure 4-2). For each entry, the linker replaces the generic assembly instructions with specific instructions (where applicable).

For example, the first PLIT entry in Figure 4-2 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, `R0` contains the referenced function's overlay ID and `R1` contains the referenced function's run-time address. The overlay manager uses the overlay ID and run-time address to load and execute the referenced function.

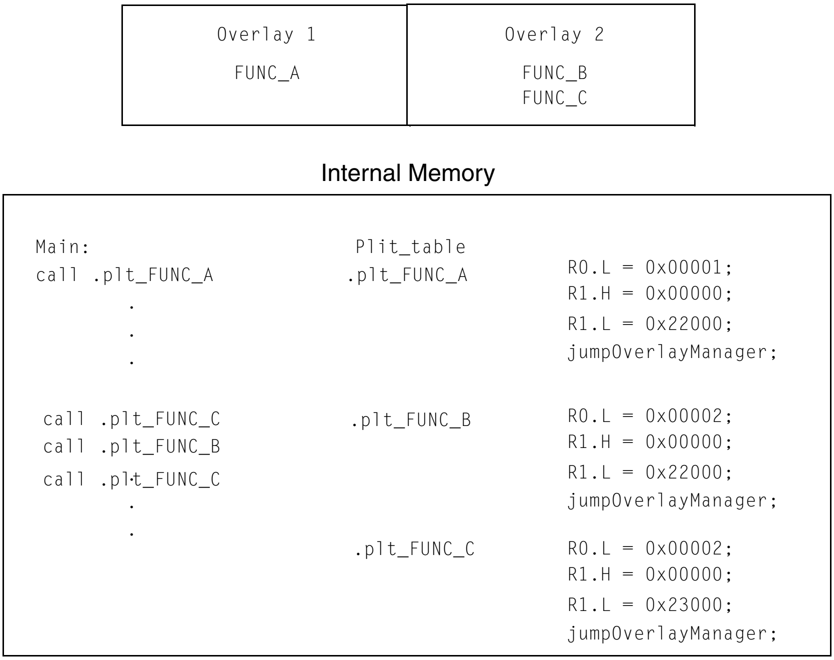

Figure 4-2. Expanded PLIT Table

Example – Managing Two Overlays

Overlay manager are user-written, and the following is an example of what an overlay manager can do. This 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 file and the overlay manager follow with appropriate explanations.

Listing 4-3. FFT Overlay Example 1

```
{ OVERLAY_INPUT
   {
      OVERLAY_OUTPUT (fft_one.ovl)
      INPUT_SECTIONS ( Fft_1st_last.doj(program) )
   } > ovl_live   // Overlay to live in section ovl_live

OVERLAY_INPUT
   {
      OVERLAY_OUTPUT (fft_two.ovl)
      INPUT_SECTIONS ( Fft_mid.doj(program) )
   } > ovl_live   // Overlay to live in section ovl_live
} > ovl_run
```

The two defined overlays (fft_one.ovl and fft_two.ovl) live in memory segment ovl_live (defined by the MEMORY{} command), and run in section ovl_run. All instruction and data defined in the program memory segment within the Fft_1st_last.doj file are part of the fft_one.ovl overlay. All instructions and data defined in program within the file Fft_mid.doj are part of overlay fft_two.ovl. The result is two functions within each overlay.

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

To verify whether an overlay is in internal memory, place the overlay ID of this overlay into a register (for example, `P0`) and compare this value to the overlay ID of each loaded overlay. This is done by loading these overlay values into a register (for example, `R1`).

```
                 /* Is overlay already in internal memory? */
CC = p0 == p1;
                 /* If so, do not transfer it in. */
if CC jump skipped_DMA_setup;
```

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

The following code segment calls the four FFT functions.

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

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

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

Register `R0` contains the overlay ID with the referenced symbol, and register `R1` contains the run-time address of the referenced symbol. The final instruction jumps 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 jumps to the address of the referenced symbol stored in `R1`.

Linker-Generated Constants

The following constants, which are 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 4-1 shows the linker-generated constants and examples of processor-specific addresses.

Table 4-1. Linker-Generated Constants and Processor Addresses

Constant	Blackfin Processors
`_ov_startaddress_1`	0x00000000
`_ov_startaddress_2`	0x00000010
`_ov_endaddress_1`	0x0000000F
`_ov_endaddress_2`	0x0000001F
`_ov_word_size_run_1`	0x00000010
`_ov_word_size_run_2`	0x00000010
`_ov_word_size_live_1`	0x00000010
`_ov_word_size_live_2`	0x00000010
`_ov_runtimestartaddress_1`	0xF0001000
`_ov_runtimestartaddress_2`	0xF0001000

The overlay manager places the constants in arrays as shown in Figure 4-3. 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 (M# for SHARC and Blackfin processors), and the beginning address of the array is stored in the Index register (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 4-3 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.

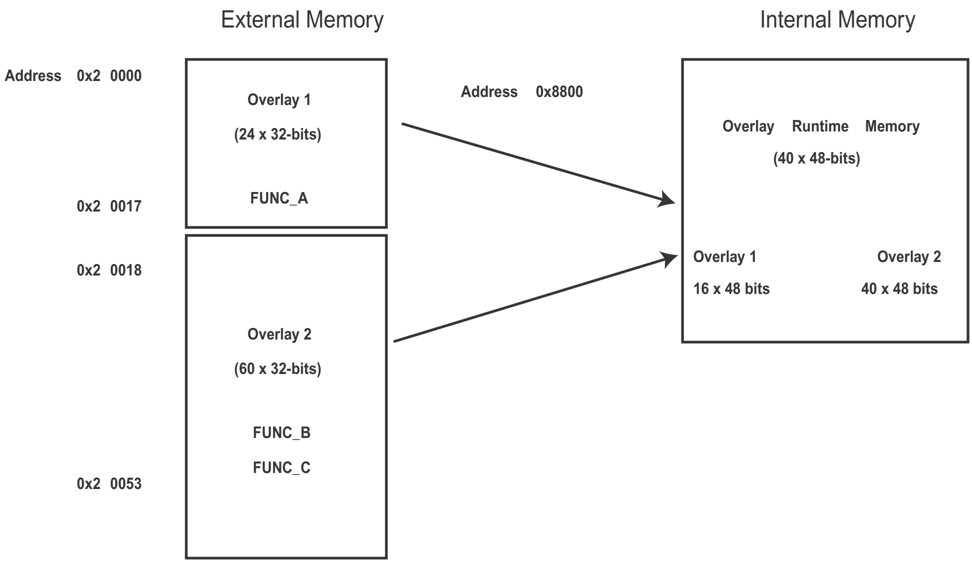

Figure 4-3. SHARC Overlay Live and Run Memory Sizes

Storing Overlay ID

The overlay manager 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:

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

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

Reducing Overlay Manager Overhead

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

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

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

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

```
     [P0] = R0;
call fft_last_stage;
```

The next set of instructions represents a load function call.

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

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

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

```
OVERLAY_GROUP1 {
   OVERLAY_INPUT
     {
       ALGORITHM(ALL_FIT)
       OVERLAY_OUTPUT(fft_one.ovl)
       INPUT_SECTIONS( Fft_ovl.doj (program) )
   } >ovl_code   // Overlay to live in section ovl_code
   OVERLAY_INPUT
     {
       ALGORITHM(ALL_FIT)
       OVERLAY_OUTPUT(fft_three.ovl)
       INPUT_SECTIONS( Fft_ovl.doj (program) )
  } >ovl_code   // Overlay to live in section ovl_code
```

```
} > mem_code

OVERLAY_MGR {
   INPUT_SECTIONS(ovly_mgr.doj(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 (last) 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.

> Specific applications may require specific code 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 4-32 for basic syntax information. Refer to "Introduction to Memory Overlays" on page 4-4 for detailed information on overlays.

Figure 4-4 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 4-4 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.

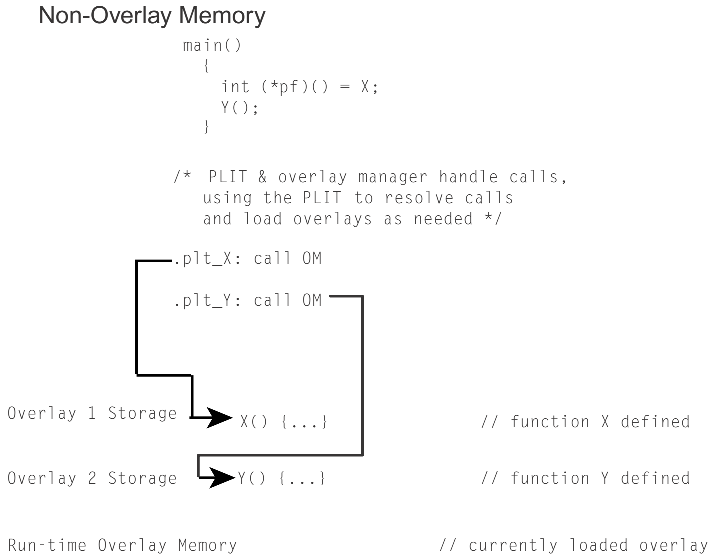

Figure 4-4. PLITs and Overlay Memory; main() Calls to Overlays

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

When the reference and the definition reside in the same executable file, the linker does not generate PLIT code. However, you can force the linker to output a PLIT, even when all references can be resolved locally. The PLIT code 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 resolve inter-processor overlay calls, as shown in Figure 4-5, for systems that permit one processor to access the memory of another processor.

When one processor calls into another processor's overlay, the call increases the size of the .plit section in the executable file that manages the overlay.

The linker resolves all references to variables in overlays, and the PLIT lets an overlay manager 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 referenced.

Inter-Processor Calls

PLITs resolve inter-processor overlay calls, as shown in Figure 4-5, for systems that permit one processor to access the memory of another processor.

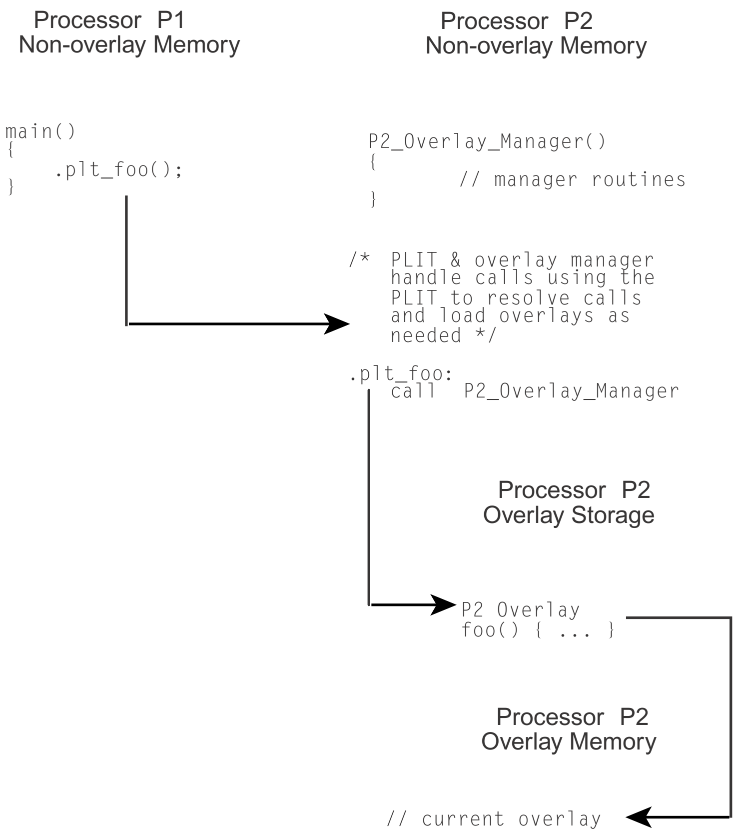

Figure 4-5. PLITs and Overlay Memory – Inter-Processor Calls

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.

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, and shared memory features.

- "OVERLAY_GROUP{}" on page 4-27
- "PLIT{}" on page 4-32

For detailed information on multiprocessor-related LDF commands, refer to "Linking Multiprocessor Systems" on page 4-36.

OVERLAY_GROUP{}

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

```
[Warning li2534] More than one overlay group or explicit
OVERLAY_GROUP command is detected in the output section
'seg_data1'. Create a separate output section for each group
of overlays.
```

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 4-7 for a detailed description of overlay functionality.

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()
        }
    }
```

In the simplified examples in Listing 4-4 and Listing 4-5, 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 4-4, 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 4-6 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 the `program` section.

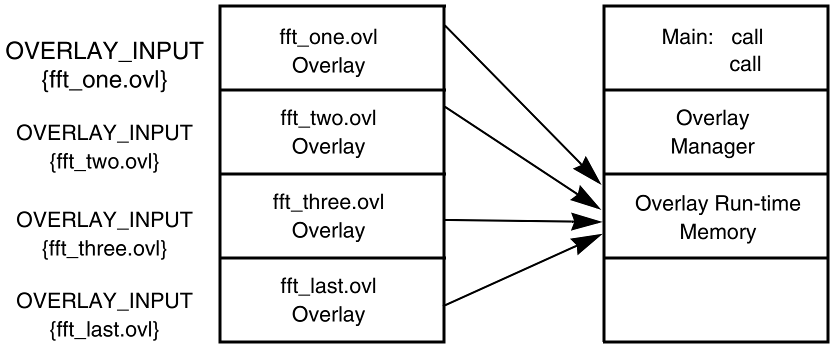

Figure 4-6. Example of Overlays – Not Grouped

Listing 4-4. 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

Figure 4-7 demonstrates grouped overlays.

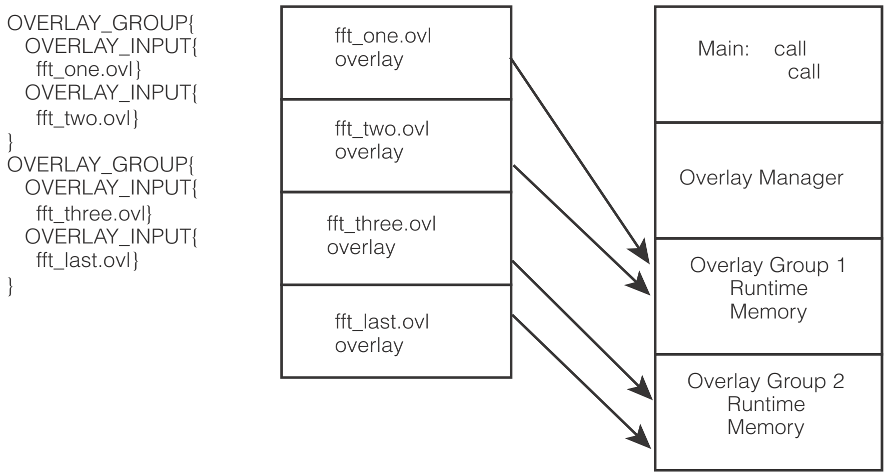

Figure 4-7. Example of Overlays – Grouped

Listing 4-5 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 4-5. 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{}` 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 4-7 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 4-23.

When writing the `PLIT{}` command in the `.ldf` file, 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 4-8 shows the general syntax of the `PLIT{}` command and indicates how the linker handles a symbol (`symbol`) local to an overlay function.

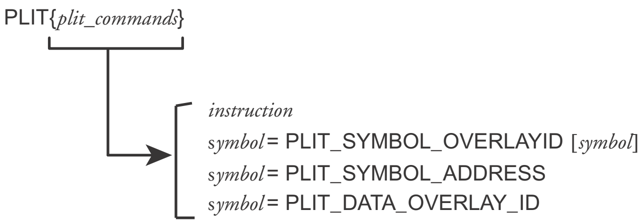

Figure 4-8. PLIT{} Command Syntax Tree

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` file 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.

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 `P0` and `P1` are either safe or irrelevant. For example,

```
PLIT
{
    P0 = PLIT_SYMBOL_OVERLAY_ID;
    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` file. **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.

Linking Multiprocessor Systems

The linker has several commands that can be used to build executable images for multiprocessor systems. Selecting the right multiprocessor linking commands and using them depend on the system you are building and the Analog Devices processor in your system.

The linker will only support linking for homogeneous multiprocessors (that is, the system must use the same kind of processor throughout). If you are building a heterogeneous multiprocessing environment, you will need to build the system with more than one link step, using an `.ldf` file for each kind of processor in your system.

A homogeneous multiprocessor system can be linked with a single `.ldf` file. The `.ldf` file will have a `PROCESSOR{}` command that describes which object files and libraries are to be linked into the memory for each processor. Every `PROCESSOR{}` command will produce a separate executable file (`.dxe`).

For processors that can access the local memory of other processors (for example, through link ports), the `MPMEMORY{}` command can be used to define the offset of each processor's physical memory. The MPMEMORY{} command is described below.

It is possible to specify the code and data that is to be placed into memory that is shared between processors. Two commands are available for placing objects and libraries into shared memory: SHARED_MEMORY{} and COMMON_MEMORY{}. Which of these commands you use will depend on how you intend to use the shared memory and the limitations of the processor architecture. The SHARED_MEMORY{} command can be used if the shared memory in the system does not contain any references to memory that is internal to an individual processor, or if the processor architecture supports addressing the internal memory of other processors.

For other processors, such as ADSP-BF561 processors, where one processor can not access the internal memory of the other processor, use the COMMON_MEMORY{} command. These commands and their usage are described in more detail below.

This section describes the following features and LDF commands:

- “Selecting Code and Data for Placement”
- “Mapping by Section Name” on page 4-40
- “Mapping Using Attributes” on page 4-41
- “Mapping Using Archives” on page 4-41
- “MPMEMORY{}” on page 4-43
- “SHARED_MEMORY{}” on page 4-44
- “COMMON_MEMORY{}” on page 4-50

Regardless of the linker commands that you use, you will have to make decisions regarding which code is going to run on which processor, where data will be placed, and what processors have access to what data. Once you have a partitioning of your code and data you can use the `.ldf` file to instruct the linker on code/data placement.

Selecting Code and Data for Placement

There are many ways to identify code and data objects for placement in a multiprocessor system. The methods are the same methods used when being selective about placement of objects in internal or external memory. There are advantages and disadvantages for each of the methods, and an `.ldf` file may combine many of these methods.

Using LDF Macros for Placement

The easiest way to partition code and data between processors is to explicitly place the object files by name. In the example below, the code that is to be placed in core A are in object files that are explicitly named in the `.ldf` file.

```
    {
    OUTPUT ( $COMMAND_LINE_OUTPUT_DIRECTORY/corea.dxe )
    SECTIONS
        {
        code
            {
            INPUT_SECTIONS (corea.doj(program)
             coreamain.doj(program))
            } > CoreaCode
...
        }
PROCESSOR COREB
    {
    OUTPUT ( $COMMAND_LINE_OUTPUT_DIRECTORY/coreb.dxe )
    SECTIONS
        {
        code
```

```
            {
            INPUT_SECTIONS (coreb.doj(program)
              corebmain.doj(program))
            } > CorebCode
...
        }
```

Doing placement explicitly by object file can be made easier through the use of LDF macros. The example could be simplified with macros for the objects to be placed in each core.

```
$COREAOBJECTS = corea.doj, coreamain.doj;
$COREBOBJECTS = coreb.doj, corebmain.doj;
...
PROCESSOR COREA
{
...
    SECTIONS
    {
        code
    {
    INPUT_SECTIONS ( $COREAOBJECTS(program) )
    } > CoreaCode
}
```

By using an LDF macro, it is much easier to make changes if functionality is going to be moved from one processor to another.

Object files can appear in more than one LDF macro. Depending on the system, the same object file may be mapped to more than one processor.

The main advantages of explicitly naming object files when placing object files to processors is that it is explicit in the `.ldf` file where each object file goes. By using LDF macros, the list of object files can be localized. A disadvantage for explicitly naming object files is that every time a new file is added to your system, the `.ldf` file must be modified to explicitly reference the file. Also, it is not possible to share the `.ldf` file with other projects that are built on the same multiprocessing system.

Mapping by Section Name

Both the compiler and assembler allow you to name sections in object files. In the assembler, this is done using the `.SECTION` directive:

```
.SECTION Corea_Code;
```

The compiler has two ways to name a section. The first method uses the `section()` qualifier:

```
section("Corea_Code") main() {...}
```

The section name can also be specified using the `section` pragma. The use of this pragma is recommended since it is more flexible and results in code that is portable.

```
#pragma section ("Corea_Code")
main() {...}
```

Users can use section names to identify code that is to be placed with a particular processor.

```
PROCESSOR COREA
    {
    OUTPUT ( $COMMAND_LINE_OUTPUT_DIRECTORY/corea.dxe )
    SECTIONS
        {
        code
            {
            INPUT_SECTIONS ( $OBJECTS(Corea_Code) )
            } > CoreaCode
...
        }
```

The advantage of mapping by section name is that the `.ldf` file can be made generic and reused for other projects using the same multiprocessor. The disadvantage is that it requires making changes to C and assembly source code files to make the mapping. Also, it may not be possible to modify source code for some libraries or code supplied by third parties.

Mapping Using Attributes

The linker now supports mapping by attributes. When compiling and assembling, users can assign attributes to object files. These attributes can then be used to filter object files for inclusion (or exclusion) during mapping. Users can assign attributes to object files that identify a core that the object files should be mapped to, a core that an object file should not be mapped to, code that is safe to be shared by all processors, and so on.

The run-time libraries are built using attributes so it possible to select areas within the run-time libraries for placement. For example, it is possible to select the objects in the run-time libraries that are needed for I/O and place them only in external memory.

An advantage of using attributes is that the `.ldf` file can be made generic and reused for other projects using the same multiprocessor. The disadvantage is that changing where an object is placed requires rebuilding the object file in order to change the attributes. Also, if all of the object files are being built in the same project, it can be inconvenient to use file-specific build options. Also, it may not be possible to rebuild the object for some libraries.

Mapping Using Archives

Another way to partition files is to build an object archive or library.

As an example, you could create a project just for building the object files to be placed in core A. The target of the project would be an archive named `corea.dlb`. The project that actually links the multiprocessor system would include `corea.dlb`. In fact, it is easiest to build a project group in which the linking project would have dependencies on the projects that build the archives it depends on. The `.ldf` file would then use the archive for linking:

```
PROCESSOR COREA
    {
    OUTPUT ( $COMMAND_LINE_OUTPUT_DIRECTORY\corea.dxe )
    SECTIONS
        {
        code
            {
            INPUT_SECTIONS ( corea.dlb(program) )
            } > CoreaCode
...
    }
```

The disadvantage of using archives for mapping is that it requires organizing more than one project. The advantage is that it can be easy to add, delete, or move objects from one processor to another. Removing an object from a project will remove it from the archive when the project is rebuilt. Adding a file to a project that builds an archive will automatically add the file to the link without needing to make changes to source. This flexibility makes it easy to create an `.ldf` file that can be shared by users building for the same architecture.

The `COMMON_MEMORY{}` command requires archives when mapping objects into memory that is shared between processors. This command is described in more detail in "COMMON_MEMORY{}" on page 4-50.

MPMEMORY{}

> ⓘ The `MPMEMORY{}` command is not used with Blackfin processors.

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 4-7 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 4-9 shows `MPMEMORY{}` command syntax.

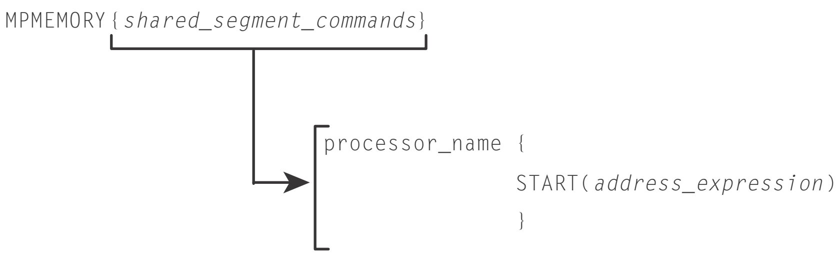

Figure 4-9. MPMEMORY{} Command Syntax Tree

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 3-16.

- *`processor_name{placement_commands}`* – Applies the *`processor_name`* offset for multiprocessor linking. Refer to "PROCESSOR{}" on page 3-49 for more information.

The `MEMORY{}` command specifies the memory map for the target system. The `.ldf` file 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 3-39. See "Memory Characteristics Overview" on page 2-25 for memory map descriptions.

SHARED_MEMORY{}

The `SHARED_MEMORY{}` command creates an executable output that maps code and data into a memory space that is shared by multiple processors. The output is given the extension `.sm` for shared memory. The `SHARED_MEMORY{}` command is similar in structure to the `PROCESSOR{}` command. The `PROCESSOR{}` command contains, among other commands, an `OUTPUT()` command that specifies a `.dxe` file for the output, and uses `SECTIONS{}` command to map selected sections from object files into specified sections in processor memory. Similarly, the `SHARED_MEMORY{}` command uses an `OUTPUT()` command and `SECTIONS{}` command to create an `.sm` file.

Figure 4-10 shows the syntax for the SHARED_MEMORY{} command, followed by definitions of its components.

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

Figure 4-10. SHARED_MEMORY{} Command Syntax

The command components are:

- OUTPUT() – Specifies the output file name (*file_name*.sm) of the shared memory executable (.sm) file. An OUTPUT() command in a SHARED_MEMORY{} command must appear before the SECTIONS{} command in that scope.
- SECTIONS() – Defines sections for placement within the shared memory executable (.sm) file.

The .ldf file will have a MEMORY{} command that defines the memory configuration for the multiprocessor. The SHARED_MEMORY{} command must appear in the same LDF scope as the MEMORY{} command. The PROCESSOR{} commands for each processor in the system should also appear at this same LDF scope.

Figure 4-11 shows the scope of SHARED_MEMORY{} commands in the LDF.

The mapping of objects into processors and shared memory is made useful by being able to have processors and shared memory "link against" each other. The LINK_AGAINST() command specifies a .dxe file or .sm file generated by the mapping for another processor or shared memory and makes the symbols in that file available for resolution for the current processor.

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

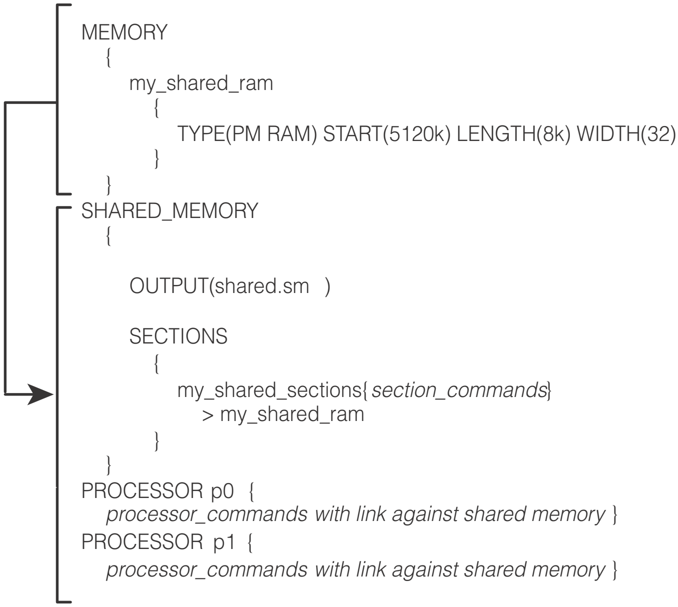

Figure 4-11. LDF Scopes for SHARED_MEMORY{}

When the `.dxe` file or `.sm` file that is named in the `LINK_AGAINST()` command is generated by another `.ldf` file, the linker will read in the executable file just as it reads in object files and archives. When the `.dxe` file of the `.sm` file that is named is being generated in the same `.ldf` file, the linker will use the executable file as it is being generated. When the processor and shared memory appear in the same `.ldf` file, the order that the processor or shared memory commands appear is not important.

For example, consider that the object file `data.doj` contains the global data buffer `DBUF`, and the object file `main.doj` contains code that references that data. Further, the data buffer `DBUF` is placed in shared memory so that it is available to multiple processors, while `main.doj` contains code that is going to be executed from core A. An `.ldf` file that does this mapping would include:

```
SHARED_MEMORY
    {
    OUTPUT("shared_memory.sm")
    SECTIONS
        {
        data_sm
            {
            INPUT_SECTIONS(data.doj(data))
            } > mem_shared_mem
        }
    }
PROCESSOR CoreA
    {
    OUTPUT("corea.dxe")
    LINK_AGAINST("shared_memory.sm")
    SECTIONS
        {
        code_corea
            {
            INPUT_SECTIONS(main.doj(program))
            } > corea_a_mem
        }
    }
```

In the example `.ldf` file, the `SHARED_MEMORY{}` command creates the output file `shared_memory.sm`. The data from the object file `data.doj` is mapped into the output file and placed into the memory named `mem_shared_mem`. (The memory definition is not shown.) Later in the `.ldf` file. the mapping for core A is done with a `PROCESSOR{}` command. In addition to creating the output file (`corea.dxe`) and mapping the program sections from the object file main.doj, it also "links against" the file corea.dxe.

The LINK_AGAINST() command has the following effect:
After all of the objects and sections for processor core A have been mapped, the symbol table in the file shared_memory.sm is used to find any symbols that could not be resolved. In the example, the object file main.doj contains a reference to the DBUF symbol but none of the object files mapped into core A contained that symbol. The symbols in shared_memory.sm are then read and DBUF is found to have been mapped into shared memory. The linker will resolve the reference in core A to be the address in shared memory that DBUF was mapped into by processing the SHARED_MEMORY{} command that produced shared_memory.sm.

The processing order described above is slightly modified if there are symbols that have weak linkage. A symbol with strong linkage in an executable named in a LINK_AGAINST() command will take precedence over a "weak" symbol.

The LINK_AGAINST() command takes effect only after mapping of objects and libraries in the input sections for the processor. Object from libraries will be mapped if needed to resolve references, even if those symbols are available in the shared memory .sm file named in the LINK_AGAINST() command. If the processor and shared memory both map the same library files, it is possible that an object from that library may get mapped into the processor and the shared memory. The multiple mapping is unlikely to make the program incorrect, but it can be a waste of memory.

The LINK_AGAINST() command can also appear within a SHARED_MEMORY{} command. It is possible for a shared memory to link against a processor .dxe file. The LINK_AGAINST() command works in the same way. After mapping objects and libraries that are listed in INPUT_SECTIONS() commands, if there are symbols that have not been resolved, the .dxe file (or .sm file) specified in the LINK_AGAINST() will be used.

It is possible for more than one `LINK_AGAINST()` command to appear in the same processor or shared memory. The `.dxe` files or `.sm` files that are named will be searched in the order they appear to resolve references.

It is also possible to have a processor link against a shared memory and have the same shared memory link against that processor. The bidirectional link against can allow code in the processor memory to call code that exists in shared memory that can then call code that is in the processor memory. As mentioned above, linking behavior does not depend on the order that processors and shared memory appear in the `.ldf` file. This order independence is still true with a bidirectional link against.

Note that references from shared memory into processor memory may not be supported by all processors. For example, for a multi-core Blackfin processor like the ADSP-BF561 processor, it is not possible for code executing in one core to access memory that is in internal memory of the other processor.

If there is code in shared memory that references internal memory of core A, that code can only be executed on core A. If core B executes the code, once core B tries to reference the internal memory on core A, the part will halt because of a hardware exception.

Also note that on parts where processors can access the internal memory of the other processors, that access may be slow and affect the performance of your program.

If you don't have `LINK_AGAINST()` commands within a `SHARED_MEMORY{}` command then there won't be any references from shared memory back to internal memory of any of the cores. If your system needs to have references from shared memory back to processors it is best to use the `COMMON_MEMORY{}` command. If there are references from shared memory back to processor internal memory for the Blackfin processors, `COMMON_MEMORY{}` is required.

One solution is to partition shared memory into a section reserved for core A, a section reserved for core B, and a section that is memory shared between the two processors. The partitioning is managed by using the MEMORY{} command. Then the PROCESSOR{} command for core A will map into the core A internal memory and into the section of shared memory reserved for core A. It will also typically link against the shared memory. The PROCESSOR{} command for core B will map into the core B internal memory and into the section of shared memory reserved for core B, and link against the shared memory. The SHARED_MEMORY{}} command is used to map the program and data that is common to both processors.

COMMON_MEMORY{}

The COMMON_MEMORY{} command provides another way to map objects into memory that is shared by more than one processor. The mapping is done in the context of the processors that will use the shared memory; these processors are identified as a "master" of the common memory. The COMMON_MEMORY{} command will also manage references from the shared memory back to internal memory of the processors so that each processor will not reference memory that is in another processor's internal memory. The COMMON_MEMORY{} command looks like the PROCESSOR{} and SHARED_MEMORY{} commands in that it uses INPUT_SECTIONS() commands for mapping. A restriction is that within a COMMON_MEMORY{} command, only archives may be mapped and not individual object files.

The following example shows the basic components of the COMMON_MEMORY{} command.

```
COMMON_MEMORY
    {
    OUTPUT("common_memory.cm")
    MASTERS(CoreA, CoreB)
    SECTIONS
        {
        data_cm
            {
            INPUT_SECTIONS(common.dlb(data))
```

```
            } > mem_common_mem
        }
    }
PROCESSOR CoreA
    {
    OUTPUT("corea.dxe")
    SECTIONS
        {
        code_corea
            {
            INPUT_SECTIONS(main.doj(program))
            } > corea_a_mem
        }
    }
PROCESSOR CoreB
    {
    OUTPUT("coreb.dxe")
    SECTIONS
        {
        code_corea
            {
            INPUT_SECTIONS(main.doj(program))
            } > corea_a_mem
        }
    }
```

The `COMMON_MEMORY{}` command uses the `OUTPUT()` to name the file that will hold the result of the mapping. The command uses the `.cm` extension for the file. The `COMMON_MEMORY{}` command also uses the `SECTIONS{}` command to map files into memory segments. However, the only files that can be mapped are archive (`.dlb`) files. Individual object files cannot be mapped from inside of a `COMMON_MEMORY{}` command.

The biggest syntactic difference in the `COMMON_MEMORY{}` command is the `MASTERS()` command. This command explicitly identifies the processors that are going to share the memory. The processor names are the name used in the PROCESSOR{} commands also appearing in the same .ldf file. Within the PROCESSOR{} command, there is no need for a LINK_AGAINST() command specifying the common memory. The MASTERS() command describes the connection.

The mapping of the archives in the COMMON_MEMORY{} command is really done when the mapping is done for the masters named in the MASTERS() command. While mapping for each of the processors named as a master, the linker will treat each INPUT_SECTIONS() command in the common memory as if they appeared within the PROCESSOR{} command. Since only archives are allowed, only the objects within the archive that are needed to satisfy references for the processor will be mapped. The mapping will be into the memory sections in the common memory.

For example, the effect of the previous example will be as if the INPUT_SECTIONS() in the COMMON_MEMORY{} were part of the PROCESSOR{}:

```
// NOT ACTUAL LDF - EFFECT OF COMMON_MEMORY{}
PROCESSOR CoreA
    {
    OUTPUT("corea.dxe")
    SECTIONS
        {
        code_corea
            {
            INPUT_SECTIONS(main.doj(program))
            } > corea_a_mem
// when mapping CoreA, the input sections from
// the common memory are mapped as if they were
// part of this PROCESSOR{} because CoreA is
// listed as a MASTER
        data_cm
        {
            INPUT_SECTIONS(common.dlb(data))
```

```
            } > mem_common_mem
        }
    }
```

Of course, by specifying with the `COMMON_MEMORY{}` command, the same mapping for the objects in `common.dlb` will also be done for core B, and the objects that are shared by the two processors will only be mapped once into the shared memory space.

The mapping will be done for each of the processors named as a master. Some symbols will be needed for each processor, and in simple cases the common memory will share the code or data between the processors. If an object is mapped into common memory that has a reference that goes back into internal memory of a processor, if necessary, the linker will make a copy of the object file so that both cores can safely use common memory. This behavior is described in the example below.

To demonstrate the complexities of multiprocessing linking, the example has several dependencies. The abbreviated C examples show the dependencies for several object files.

```
// file mainA.doj
void mainA() {
// the main code in CoreA references 2 common functions
    commonfunc1();
    commonfunc2();
}
// file mainB.doj
void mainB() {
// the main code in CoreB references 3 common functions
    commonfunc1();
    commonfunc2();
    commonfunc3();
}
// file func1.doj
void commonfunc1() {
```

```
// a common function with a reference to a library
    libfunc1();
    }
// file func2.doj
void commonfunc2() {
// a common function with a reference to a library
    libfunc2();
    }
// file func3.doj
void commonfunc3() {
// no further references
    }
// file libfunc1.doj and libfunc2.doj have no further references
// create archives for common files
elfar -c common.dlb func1.doj func2.doj func3.doj
elfar -c commonlib.dlb libfunc1.doj libfunc2.doj
```

Each of the processors has its own `main` function. Each `main` function makes calls to common functions. Some of the common functions make further calls to library functions. The common functions have been placed in an archive named `common.dlb`, and the library files have been placed in an archive named `commonlib.dlb`.

The `.ldf` file to build the multiprocessor system is shown below.

```
COMMON_MEMORY
    {
    OUTPUT("common_memory.cm")
    MASTERS(CoreA, CoreB)
    SECTIONS
        {
        data_cm
            {
            // the common libraries are mapped into common
             //  memory
```

```
            INPUT_SECTIONS(common.dlb(program)
              commonlib.dlb(program))
            } > mem_common_mem
        }
    }
PROCESSOR CoreA
    {
    OUTPUT("corea.dxe")
    SECTIONS
        {
        code_corea
            {
            INPUT_SECTIONS(mainA.doj(program))
            // for performance reasons map
             // libfunc1.doj into this core
            INPUT_SECTIONS(libfunc1.doj(program))
            } > corea_a_mem
        }
    }
PROCESSOR CoreB
    {
    OUTPUT("coreb.dxe")
    SECTIONS
        {
        code_coreb
            {
            INPUT_SECTIONS(mainB.doj(program))
            } > corea_b_mem
        }
    }
```

Notice that processor core A explicitly maps `libfunc1.doj` into its internal memory. Core B does not map a version of `libfunc1.doj`. Both processors link against the common memory that does mapping against the archives that contain common functions.

To understand the operation of `COMMON_MEMORY{}`, let's walk through the mapping of the objects into memory, beginning with core A. The `INPUT_SECTIONS()` commands for core A will map `mainA.doj` and `libfunc1.doj` into the memory `corea_a_mem`. The references to `commonfunc1` and `commonfunc2` will cause the object files `func1.doj` and `func2.doj` to be pulled out of the archive `common.dlb` and they will be mapped into the common memory `mem_common_mem`. The object file `func1.doj` has a reference to `libfunc1`. This symbol was already mapped when `libfunc1.doj` was mapped into the core memory. The object file `func2.doj` has a reference to `libfunc2` so the object `libfunc2.doj` will be pulled out of the archive `commonlib.dlb` and it will also be mapped into `mem_common_mem`. Note that this mapping only considers the files required for core A so `commonfunc3` is not considered.

The mapping for core B will be similar. The `INPUT_SECTIONS()` command for core B will map `mainB.doj` into the memory `coreb_b_mem`. The references to the common functions will cause the object files `func1.doj`, `func2.doj`, and `func3.doj` to be pulled out of the archive `common.dlb` and be mapped into `mem_common_mem`. The references in the common functions to the library functions will cause the library objects to be pulled from the `commonlib.dlb` so `libfunc1.doj` and `libfunc2.doj` will be mapped into the common memory `mem_common_mem`. Note that this mapping only considers the files for core B and the common memory. In particular, the fact that `libfunc1.doj` was mapped into core A memory is not considered for this mapping.

Now the linker ensures that all the objects mapped into common memory can be shared; for those files that cannot be shared, it will fix them by making duplications. Those object files mapped into common memory that do not have any further references (the leaf functions `func3.doj`, `libfunc1.doj`, and `libfunc2.doj`) are fine as they are. The function `commonfunc2` references `libfunc2.doj` (which is only mapped into common memory), so it is also fine. The function `commonfunc1` references `libfunc1.doj`. In the context of core A, `func1.doj` will call the version of `libfunc1` that is mapped into core A internal memory. In the context of core B, `func1.doj` will call the version of `libfunc1` that is mapped into common memory. To resolve this problem, the linker will create a copy of `func1.doj`. The `mainA` function will call the version that references back to the version of `libfunc1` that is in core A memory while `mainB` will call the version that references back to the version of `libfunc1` that is in common memory.

It is rare that an object mapped into common memory will be duplicated. When an object is duplicated, the linker will only duplicate the minimal amount needed to keep integrity. The duplication will only happen in cases where using the `SHARED_MEMORY{}` command would have resulted in a run-time exception, because a processor was accessing memory in another processor's internal memory.

5 ARCHIVER

The archiver (`elfar`) combines object (`.doj`) files into library files, which serve as reusable resources for code development. The 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:

- "Introduction" on page 5-1
  Introduces the archiver's functions
- "Archiver Guide" on page 5-2
  Describes the archiver's functions
- "Archiver Command-Line Reference" on page 5-12
  Describes archiver operations by means of command-line switches

Introduction

The `elfar` 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 object files of an existing library file to `stdout`
- Replaces file(s) in an existing library file
- Encrypts symbol(s) in an existing library file
- Embeds 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, 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.

Archiver Guide

The `elfar` utility combines and indexes object files (or any other files) to produce a searchable library file. This section describes the following archiver functions:

- "Creating a Library" on page 5-2
- "Making Archived Functions Usable" on page 5-3
- "Archiver Symbol Name Encryption" on page 5-10

Creating a Library

To create an archive, use the `-c` switch when invoking the archiver from the command line (as shown in "Archiver Command-Line Reference" on page 5-12). The command line should include the name of the archive being created and the list of objects files to be added.

Example:

```
elfar -c my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

If the objects files were created using the C/C++ compiler, it is recommended that the compiler driver and the compiler's `-build-lib` switch are used to build the library (the compiler driver invokes `elfar` to build the library). Refer to the appropriate *C/C++ Compiler and Library Manual* for more information.

Example:

```
ccblkfn -build-lib -o my_lib.dlb fft.doj sin.doj cos.doj tan.doj
```

It is possible to build a library from within the IDE. CrossCore Embedded Studio writes output to `<projectname>.dlb`.

To maintain code consistency, use the conventions in Table 5-1.

Table 5-1. 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:

- Creating *library routines*, functions that can be called from other programs, and library data, *variables*, that can be referenced from programs
- Accessing library routines and data from your code

Writing Archive Routines: Creating Entry Points

A library routine (or function) in code can be accessed by other programs. Each routine must have a globally visible start label (*entry point*). Library data must be given a visible label. Code that accesses that routine must declare the entry point's name as an external symbol in the calling code.

To create visible external symbol:

1. Declare the start label of each routine and each variable as a global symbol with the assembler's `.GLOBAL` directive. This defines the entry point.

   The following code fragment has a visible entry point for the function `dIriir` and creates a visible symbol for the variable `FAE`.

```
...
.global dIriir;
.section data1;
.byte2 FAE = 0x1234,0x4321;

.section program;
.global FAE;
dIriir: R0=N-2;
P2 = FAE;
```

2. Assemble the files into object files containing the global segments.
3. You can also write library functions in C and C++. Functions declared in your C/C++ file will be given globally visible symbols that can be referenced by other programs. Use the C/C++ compiler to create objects files, and use the compiler driver and its `-build-lib` switch to create the library.

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-BF533.ldf`), and creates the executable file (`main.dxe`).

```
linker -DADSP-BF533 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 `.dxe` file.

Specifying Object Files

The list of object files on the command line is used to specify objects to be added to the archive. Such commands are `-c` (create), `-a` (add), or `-r` (replace). The list can also be used to specify objects in the library to be extracted using the `-e` (extract) command.

When the list refers to object files to be added to the archive, the file name is specified the way the file names are specified for the host operating system. The file name can include path information – relative or absolute. If path information is not included, the archiver will look for the file in the current working directory.

When the list refers to object files already in the archive, the file names should not include any path information. The archiver only saves the base file name for the object files in the archive.

The archiver accepts the wildcard character "*" in the specification of the object file names. On Windows systems, the archiver does all interpretation of the wildcard character. When it appears in a list of object files to be added, the archiver searches the file system for files that match this specification. When a wildcard appears in a list of objects already in the library, the archiver will search through the object files in the library for matches.

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 5-2), 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` command. 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 5-9).

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
```

The archiver adds “::” to the front of the version information to highlight it.

User-Defined Version Information

You can provide any number of user-defined version values by supplying a text file that contains those values. The text file can have any number of entries. Each line in the file begins with a name (a single token with no 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 5-2) 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
object.doj
```

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 5-2) prints any version information. Two forms of the `-p` switch can be used to examine version information.

The -pv switch prints version information only, 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 adding version information, 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 version information was added to the archive.

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 5-2), 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 5-2) 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. 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 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 5-2) 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 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.

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"

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 5-2 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 5-1 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 5-10 for more information.

Archiver Parameters and Switches

Table 5-2 describes each archiver part of the `elfar` command. Switches must appear before the name of the archive file.

Table 5-2. Command-Line Switches and Entries

Item	Description
exclude_file	Specifies the name of a text file containing a list of symbols not to be encrypted.
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.
type-letter	The initial letter of *type-letter* provides the initial letter of all encrypted symbols.
`-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 5-2. Command-Line Switches 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 5-10.
-t *verno*	Tags the library with version information in string
-tx *filename*	Tags the library with full 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	Disables 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 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, or s) 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.

6 MEMORY INITIALIZER

CrossCore Embedded Studio includes a memory initializer 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 runtime 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 6-16).

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 that the section contains instructions, data, or both), and extracts the data and instructions from them to make a callback initialization stream. When this stream is built up, the callback `.dxe` files are processed in the order 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 6-15
- "-Init Initcode.dxe" on page 6-16
- "-NoAuto" on page 6-16
- "-NoErase" on page 6-17
- "-Section Sectionname" on page 6-17

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 memory initializer command-line switches.

By taking multiple input files, the memory initializer supports systems that 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 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, each containing a piece of the callback executable
- The initialization data and code from the primary application

Figure 6-1 shows the basic structure of the initialization stream.

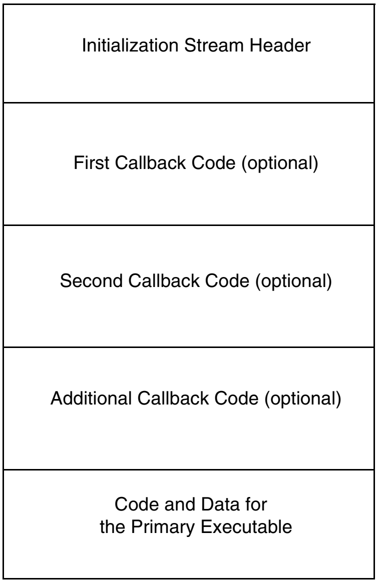

Figure 6-1. Memory Initializer Basic Initialization Stream Structure

RTL 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, copies "callback" code into RAM and then executes it. This is performed piece-by-piece and continues 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 Memory Initializer

There are several reasons why it may be beneficial to use the memory initializer:

- The system needs to initialize RAM memory from 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 (`ZERO_INIT` and `RUNTIME_INIT`). Sections marked with `ZERO_INIT` may contain only "zero-initialized" data, and sections marked with `RUNTIME_INIT` may contain the data with any initialization values.

Refer to the `SECTIONS` description (on page 3-56) for detailed information on these qualifiers.

The following example shows how to use the `ZERO_INIT` and `RUNTIME_INIT` qualifiers in an `.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 file sections according to those flags, and produces a modified output .dxe file.

The memory initializer is able to identify the .dxe file 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.

Note that two memory sections are specified in default .ldf files, which also serve the memory initializer: bsz_init and .meminit. The bsz_init section holds the pointer generated by the memory initializer, which points to the start address of the initialization stream, and the section .meminit holds the actual initialization stream generated by the memory initializer. Although other sections may be selected as alternatives (using the appropriate command-line switches), this is not recommended.

Preparing the Source Files

The sections marked with the ZERO_INIT and RUNTIME_INIT qualifiers must 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 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 Memory Initializer

The memory initializer is invoked from a command line or from the IDE.

Invoking meminit From the Command Line

The simplest command line to invoke the memory initializer 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. Memory initializer command-line switches are listed in Table 6-1.

> Users of SHARC processors that have been using `mem21k` to invoke the memory initializer from a command line can continue to do so. However, invoking `meminit` accomplishes the same results, since `meminit` passes the command to `mem21k` when used with a SHARC processor.

Invoking meminit From the 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-BF533 main.doj -meminit -o project1.dxe
```

Invoking meminit From the 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-BF533 -mem main.c -o output.dxe
```

Invoking meminit From the IDE

Following the **Properties** > **C/C++ Build** > **Settings** > **Tool Settings** path in the IDE, choose the **Additional Options** node under the linker node (see Figure 6-2). Click on the **Add** button in the **Additional options for linking** field. Type `-meminit` in the **Enter Value** box. Click **OK** and then click **Apply**. When the project is built, the linker calls the memory initializer.

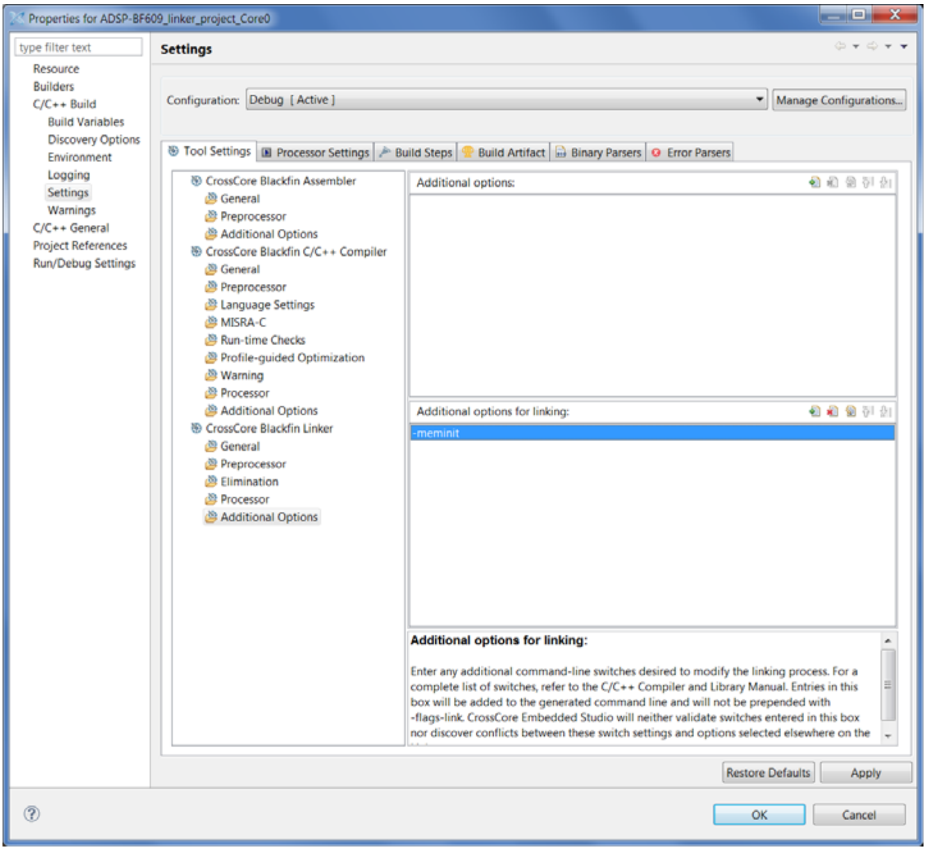

Figure 6-2. Invoking the Memory Initializer From the IDE

Invoking meminit With Callback Executables

To directly invoke the memory initializer from a command line, use the `-Init` switch for each "callback" executable as shown below:

```
  meminit Input.dxe -o Output.dxe -Init Callback1.dxe
-Init Callback2.dxe
```

From the IDE, choose **Properties > C/C++ Build > Settings > Tool Settings** and select the **Additional Options** node under the linker node. Use the **Additional options for linking** 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, click on the **Add** button in the **Additional options for linking** field and type:

```
-meminit -flag-meminit -Init callback1.dxe -Init callback2.dxe
```

in the **Enter Value** box. Click **OK** and then click **Apply** (see Figure 6-3).

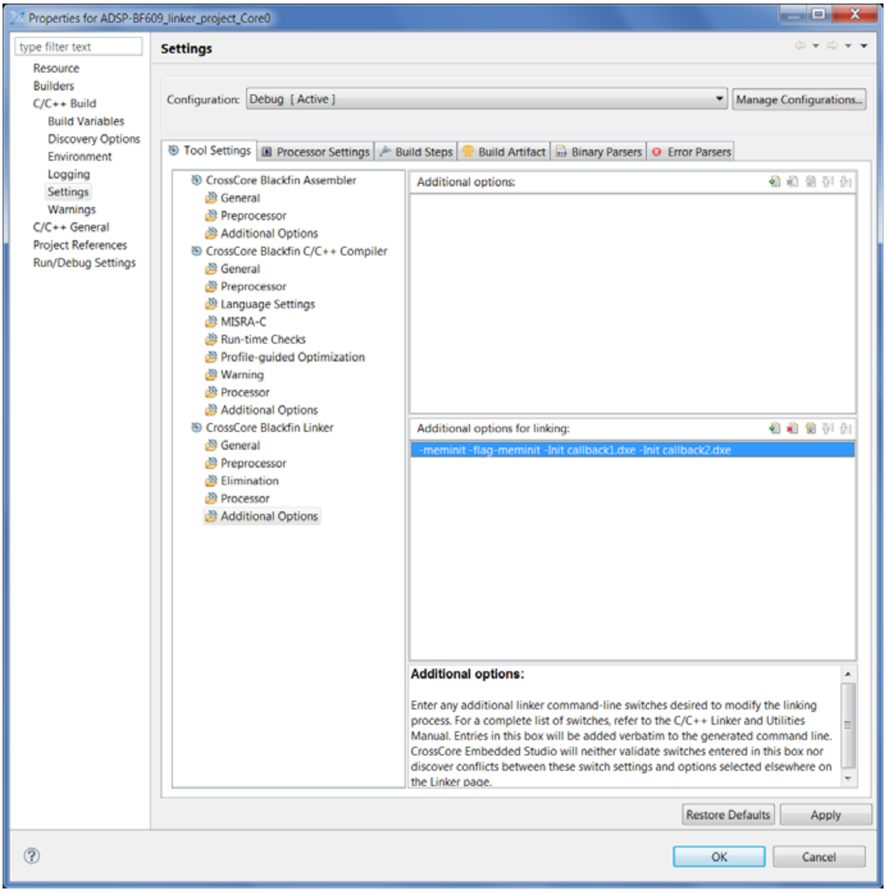

Figure 6-3. Invoking Callback Executable From the IDE

Memory Initializer Command-Line Switches

Table 6-1 summarizes the memory initializer switches. It is followed by a detailed description of each switch.

Most of the listed switches are 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 initialized properly in the source files, most of these optional switches may not be required. By default, the memory initializer automatically handles everything needed to create an initialization stream.

Table 6-1. Summary of Command-Line Options and Entries

Item	Description
`-BeginInit` *`Initsymbol`* on page 6-15	Specifies a symbol name for a variable that holds a pointer pointing to the start address of an initialization stream.
`-h[elp]` on page 6-15	Displays the list of memory initializer switches.
`-IgnoreSection` *`Sectionname`* on page 6-16	Directs the memory initializer to NOT process a section selected in the primary input file.
`-Init` *`Initcode.dxe`* on page 6-16	Specifies an executable file to be inserted into the initialization stream and executed as a callback.
`InputFile.dxe` on page 6-16	Specifies a primary input file.
`-NoAuto` on page 6-16	Directs the memory initializer to NOT process sections in the primary input file based on the section header flags. This switch is optional.
`-NoErase` on page 6-17	Directs the memory initializer not to erase the data of the processed sections in the primary executable file.
`-o` *`Outputfile.dxe`* on page 6-17	Specifies an output file.
`-Section` *`Sectionname`* on page 6-17	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.
`-v` on page 6-18	(Verbose) Outputs status information as the memory initializer 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 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 code) is searched, which, by default, is in the `bsz_init` memory section. If this symbol cannot be found in the input primary file, an error message is issued; for example:

```
meminit -BeginInit boggy input.dxe
ERROR: The specified destination section, .meminit, not found in
the input 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 6-17). It also must be able to hold a value that 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 6-16.

-h[elp]

The `-h[elp]` switch displays 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. This switch does not affect a callback executable file specified using the `-Init` *`Sectionname`* 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 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 whether this switch is used. 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 create 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 an output file name from the root of the input file name. For example, if the input file name is `InputFile.dxe`, the output file name is created as `InputFile1.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 `-IgnoreSection` switches cannot be used with the `-Section` switch.

It is not necessary to use this switch to specify sections that 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.

-v

The `-v` or `-verbose` (verbose) switch directs the memory initializer to output status information as it processes files.

A FILE FORMATS

CrossCore Embedded Studio supports 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-1
- "Build Files" on page A-4
- "Debugger Files" on page A-8

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-2
- "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-4

C/C++ Source Files

C and C++ source files are text files (with extensions such as `.c`, `.cpp`, `.cxx`, and so on) that contain 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 Analog Devices 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 Analog Devices extensions to ANSI C, refer to the *C/C++ Compiler and Library Manual* for appropriate target architecture.

Assembly Source Files (.asm)

Assembly source files are text files that contain 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, refer to the *Assembler and Preprocessor Manual.*

[1] With and without built-in function support; a minimal differentiator. There are others.

Assembly Initialization Data Files (.dat)

Assembly initialization data (.dat) files are text files that contain fixed-point 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.

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, refer to the *C/C++ Compiler and Library Manual* for appropriate target architecture. For information on the assembler and preprocessor, see the *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 3-31.

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 2-39.

Build Files

Build files are produced by CrossCore Embedded Studio 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-5
- "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 memory 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. Extended Linear Address Record Example

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. Data Record Example

Field	Purpose
:0402100000FE03F0F9	Example record
:	Start character
04	Byte count of this record
0210	Address
00	Record type
00	First data byte
F0	Last data byte
F9	Checksum

Table A-4. End-of-File Record Example

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 *Loader and Utilities 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 *Loader and Utilities 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 CrossCore Embedded Studio.

- 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/`and search on the text string ELF.

- (1993) *Debugging Information Format (DWARF) V3.0* from the DWARF Standards Committee.

  Go to: `http://dwarfstd.org`

B UTILITIES

The CrossCore Embedded Studio software includes the following utilities:

- "elfdump – ELF File Dumper" on page B-1
- "elfpatch" on page B-5

elfdump – ELF File Dumper

The executable and linking format (ELF) file dumper (`elfdump`) utility extracts data from ELF-format executable (`.dxe`) and object (`.doj`) files and yields text showing the ELF file's contents.

The `elfdump` utility is often used with the archiver (`elfar`). Refer to "Disassembling a Library Member" on page B-3 for details. Also refer to "Dumping Overlay Library Files" on page B-4 on how to extract and view the contents of overlay library files.

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
`-fh`	Prints the file header
`-arsym`	Prints the library symbol table
`-arall`	Prints every library member
`-help`	Prints the list of elfdump switches to stdout
`-ph`	Prints the program header table
`-sh`	Prints the section header table. This 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
`-c`	Same as `-ost` (deprecated)
`-s`	Same as `-ost` (deprecated)
`-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` switches can have modifier letters 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.
- `s` – Prints contents as list of disassembled machine instructions and also prints labels.

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 5-1.

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
```

Open the text file, scroll through it, and locate the object file you need.

To convert the object file to an assembly listing file with labels, use the following `elfdump` command line, which references the library and the object file in the library.

```
elfdump -ns * libc.dlb (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 clone2.elf
elfdump -all clone2.elf
```

Switches for the `elfdump` commands are case sensitive.

elfpatch

The ELF patch (`elfpatch`) utility allows the bits of an ELF section to be extracted or replaced from a file.

Syntax:

```
elfpatch -get [section-name] -o[output-bits-filename] -text
[input-elf-filename]
```

```
elfpatch -replace [section-name] -o[output-filename] -bits
[input-bits-filename] -text [input-elf-filename]
```

```
elfpatch [help | version]
```

Examples:

```
elfpatch -get _ov_os_overlay_1 -o bytes_bin o1.ovl (overlay1.elf)
```

```
elfpatch -get L1_code -o bytes_txt -text p0.dxe
```

```
elfpatch -replace _ov_os_overlay_1 -o o1_new_from_txt.ovl -bits
bytes_txt -text o1.ovl (overlay1.elf)
```

```
elfpatch -replace L1_code -o p0_new.dxe -bits bytes_bin p0.dxe
```

Extracting a Section in an ELF File

The `elfpatch -get` command dumps the raw contents of a section without any additional formatting. The `input-elf-filename` parameter may be one of the following:

- A stand-alone (non-archive) ELF file containing a section specified by the `section-name` parameter
- A library (filename) combination

The `-text` switch specifies that the output should be a stream of printable text, with each one byte of binary output resulting in two hexadecimal digits in text output. If the `-o` switch for specifying a output file is not given, the output is written to `stdout`.

Replacing Raw Contents of a Section in an ELF File

The `elfpatch -replace` command replaces the raw contents of a section. The replacement bits need not be the same size as the section being replaced.

If the replacement resulted in the replace section clobbering a portion of another section, an error would result in a resolved ELF file.

If the `-bits` switch is not specified, bits are read from `stdin`.

The `input-elf-filename` parameter must exist and be either of the following:

- A stand-alone (non-archive) ELF file containing a section specified by the `section-name` parameter
- A `library (filename)` combination

Ultimately, the `input-elf-filename` parameter must contain a section specified by the `section-name` parameter. If the `-o` switch is not specified, the output (ELF file) is written to `stdout`.

The `-text` switch specifies that the input should be a stream of printable text, with two hexadecimal digits per input byte.

Standard input (`stdin`) and standard output (`stdout`) are used to facilitate piping. Here is an example command line:

```
elfpatch -get code input.dxe | my-transformation | elfpatch
-replace code input.dxe -o output.dxe
```

C LDF PROGRAMMING EXAMPLES FOR BLACKFIN PROCESSORS

This appendix provides several typical `.ldf` files. used with Blackfin processors. As you modify these examples, refer to the syntax descriptions in “LDF Commands” on page 3-31.

This appendix provides the following examples.

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

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:

`<install_path>\Blackfin\Examples`

The development software includes a variety of default `.ldf` files. These files provide an example `.ldf` file for each processor’s internal memory architecture. The default `.ldf` files are in the directory:

`<install_path>\Blackfin\ldf`

Linking for a Single-Processor System

When you link an executable file for a single-processor system, the `.ldf` file describes the processor's memory and places code for that processor. The `.ldf` file in Listing C-1 is for a single-processor system. Note the following commands in this example file.

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

Listing C-1. Example LDF for a Single-Processor System

```
ARCHITECTURE(ADSP-BF533)

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( "BF533_memory.ldf") }

   SECTIONS               // Specify the Output Sections
   { INCLUDE( "BF533_sections.ldf" }
                          // end P0 sections
}                         // end P0 processor
```

Linking Large Uninitialized or Zero-initialized Variables

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

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers, this action can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur long download times when used with an emulator. This situation also may occur when you boot from a loader file (because of the increased file size). Listing C-2 shows an example of assembly source code. Listing C-3 shows the use of the NO_INIT and ZERO_INIT sections to avoid initialization of a segment.

The .ldf file 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 3-56 for more information on the NO_INIT and ZERO_INIT section qualifiers.

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

Listing C-2. Large Uninitialized Variables: Assembly Source

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

Listing C-3. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-BF533)
$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
    SECTIONS {
        extram_output NO_INIT {
            INPUT_SECTIONS( $OBJECTS ( extram_area ) )
            } >mem_extram
        zero_extram_output ZERO_INIT {
            INPUT_SECTIONS ( $OBJECTS ( zero_extram_area ) )
            } >mem_extram
    }    // end section
  }     // end processor P0
```

D LDF PROGRAMMING EXAMPLES FOR SHARC PROCESSORS

This appendix provides several typical .ldf files used with SHARC processors. As you modify these examples, refer to the syntax descriptions in "LDF Commands" on page 3-31.

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 file that come with the examples. Examples are in the following directory:

```
<install_path>\SHARC\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 located in the following directory:

```
<install_path>\SHARC\ldf
```

Linking a Single-Processor SHARC System

When linking an executable for a single-processor system, the .ldf file 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 CrossCore Embedded Studio 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 file 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 3-56 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
    SECTIONS {
        sdram_output NO_INIT {
            INPUT_SECTIONS( $OBJECTS ( sdram_area ) )
              } >mem_sdram
        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.

Here are the major commands in an `.ldf` file. For examples, examine the `.ldf` files included with the installation CD.:

- 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.

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` files (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.

I INDEX

D

E

G

H

I

J

K

L

M

N

O

P

T

U

V