

2 LINKER

Overview

The VisualDSP++ linker, `linker.exe`, *links* (maps into memory, resolves references among) object and library files to produce executable (`.DXE`) files, shared memory (`.SM`) files, and overlay (`.OVL`) files, which can be loaded onto the target. It can also produce map files and other output, containing information to be used by the VisualDSP++ 2.0 debugger.

The linker reads the target hardware information to determine appropriate addresses for code and data. This address information is specified in the assembly code. The linker generates a memory image file containing a single executable program which may be loaded into a simulator or emulator for testing.

All information to be used by the VisualDSP++ debugger is provided in the `.DXE` file.

The linker accomplishes these tasks according to settings that you can choose under the direction of a Linker Description File.



Most code examples in this manual correspond to the use of the ADSP-TS001 DSP from the TigerSHARC family of Analog Devices processors.

Overview

This chapter contains the following information on the linker:

- [“Linker Guide” on page 2-13](#)—describes how to use the linker for producing executable files
- [“Linker Command-Line Reference” on page 2-25](#)—lists linker command line switches and syntax
- [“Linker Description File Reference” on page 2-38](#)—describes linker description file syntax and programming techniques
- [“LDF Programming Examples” on page 2-76](#)—provides a series of LDF programming examples for different types of systems
- [“Linker Glossary” on page 2-112](#)—provides definitions of linker related terms

Mapping Files To Memory With An LDF

Whether you want to link a C/C++ function or an assembly routine, the mechanism is the same. All projects must locate their code and data in the DSP's memory (internal or external) to execute. To map portions of code or data to specific memory segments, you use a *Linker Description File* (LDF) to direct the linking operation. This section explains the overall function of the LDF.

Therefore, they must all specify the linking process with an LDF, even if you don't include one. See [“Default LDF and Object Code Placement”](#) for a discussion of what happens if there is no LDF in your project.

The LDF consists of *Commands*, specifying the relevant components of the code and the target system. You place the information needed to link your code in the text of these commands. Several simple examples are shown later in this section.

You can write your own LDF, using information in this chapter and in other VisualDSP++ documentation, or modify an existing LDF, which is often the easier alternative if you are not dealing with large changes in your system's hardware or software.

For an introduction to the LDF commands, and the ways to construct an LDF, refer to [“Linker Guide” on page 2-13](#).

Default LDF and Object Code Placement

If you neither write nor import an LDF into your project, the VisualDSP++ IDE uses a *Default LDF* to link your code. This file is packaged with your DSP tool kit distribution, in a subdirectory specific to your target processor's family. There is a separate default LDF for each target architecture supported by your VisualDSP++ installation. The default LDF reflects your target environment's memory structure, as selected when you set your project's options, and locates the program (and

Mapping Files To Memory With An LDF

data) portions of your object code according to the `-Dprocessor` command line switch, where *processor* is your target architecture.

 The default compiler section names are `program code`, `data1` and `data2` for data; additional section names are defined in LDF files for use by the linker. For more information about compiler sections, see the *VisualDSP++ 2.0 C/C++ Compiler & Library Manual for TigerSHARC DSPs*.

For more information on LDF syntax, see [“LDF Expressions and Conventions” on page 2-40](#). For sample LDFs and related source files, see the samples included with your VisualDSP++ software.

Inputs—C/C++ and Assembly Sources

The first step towards gaining an understanding of the LDF is to first understand the makeup of the files involved in building a DSP executable.

The process starts with source files. These files contain code written in either C/C++ or Assembly. The first step towards producing an executable is to produce object files by compiling or assembling these sources. The development software names object files with the extension `.DOJ`.

The code in object files produced by the compiler and assembler consists of various *sections*, referred to as *Input Sections*. Each type of Input Section holds a particular type of compiled/assembled source code. For example, an Input Section could hold program opcodes or data (i.e., variables). Some of these Input Sections can also hold debug information.

Each Input Section has a unique name you specify in the source code. Depending on whether the source is C, C++ or assembly, there are different conventions for naming an Input Section.

Input Section Directives in Assembly Code

Linker directives are obligatory in assembly code. Each code and data object must be placed explicitly or assembly will fail.

In an assembly source, code that goes into a particular Input Section appears directly after a `.SECTION <name>` directive in the source file. An example of this is as follows:

```
.section data1;          // declares section data1
.var input[3];          // declares data buffer in data1

.section program;        // declares section program
    k6=k31+output_imag; // code in section program
    j6=j31+output_real; k5=k31+0;;
    LC0=N/4; k7=k31-2;;
```

In this example, the Input Section `data1` contains the array `input`, and Input Section `program` contains the 3-lines of code.

Input Section Directives in C/C++ Source Files

In a C/C++ source file, you use the `section("section_name")` C language extension to define Input Sections. A fragmentary example follows:

```
segment("ext_data") int temp;

segment("ext_code") void func1(void) {
    int x = 1;
}

void func2(void) {
    int i = 0;
}
```

Because this code uses the `section("seg_name")` extension, as the compiler processes the source, the compiler stores the code generated from `func1` in its own separate Input Section of the `.DOJ` file named `ext_code`. Also during compilation, the compiler puts the variable `temp` in the Input Section `ext_data`.

Mapping Files To Memory With An LDF

Note that the `section ("seg_name")` extension is optional. As shown in the example, the `func2` function does not use `section ("seg_name")` syntax. If your code does not specify an Input Section name, the compiler uses a set of default names. In this case, the compiler puts the code from `func2` in an Input Section for program code, which has the default Input Section name `program`. For more information on the compiler's default Input Section names, see the *C/C++ Compiler & Library Manual for TigerSHARC DSPs*.

The linker uses the Input Sections' contents to make executable files. The Input Section names are the labels the linker uses to find the Input Sections within object files.



It is important to identify the difference between Input Section names and executable file names, because both types of names appear in the linker description file.

Outputs—DSP Executables

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

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

Input Section and Output Section names occupy different namespaces. They are independent and may be replicated in an LDF. Your LDF can use Output Section names that are exactly the same as Input Section names. Using the same name for different items—although legal—is poor programming style and tends to make the LDF more difficult to understand.

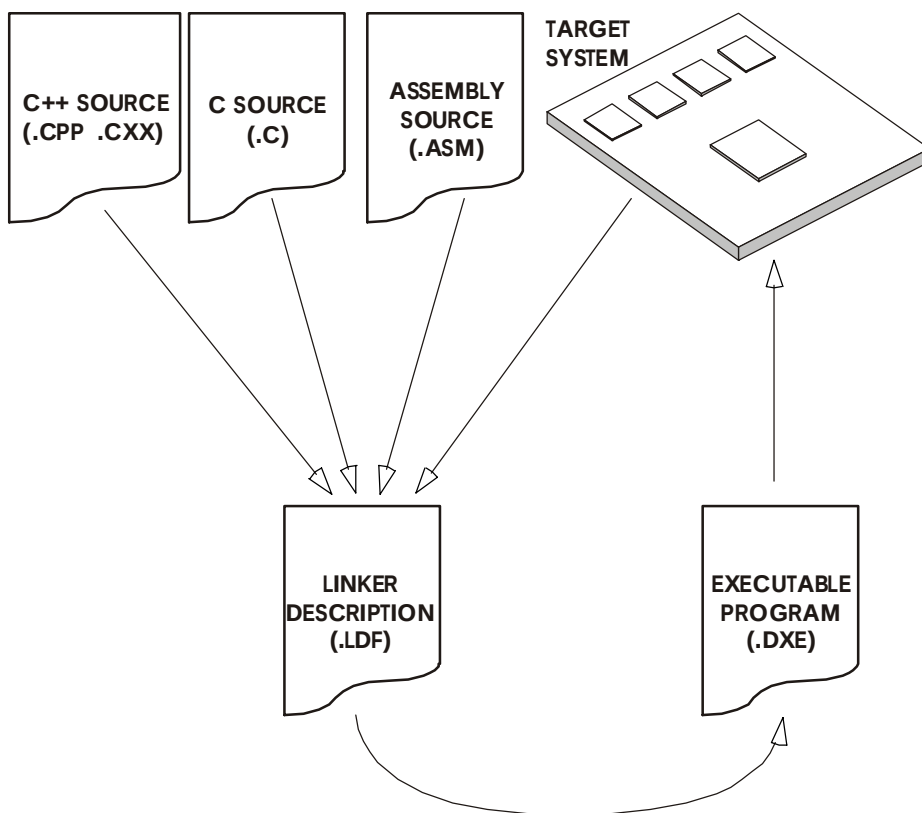


Figure 2-1. The LDF and the Linking Process



It is important to understand the function of the `.DXE` executable file. It is neither loaded into the DSP nor burned into an EPROM. The `.DXE` contains the raw code and data from the object files, along with additional information, used by utilities (such as the debugger) to locate code in the target (DSP, simulator, ICE, ...).

The Linking Process and the LDF

The linker's job is to take the Input Sections in object files and place them in Output Sections in the executable file using the commands in the LDF file. The LDF defines the DSP's memory and where within that memory the linker puts the Input Sections.

- Your source code specifies one or more Input Sections as destinations for its compiled/assembled object(s).
- The compiler and assembler produce object code, with labels directing which portions are allocated to which Input Sections. More than one code item may be placed in the same Input Section, but not vice versa.
- The linker maps each Input Section in the object code to an Output Section, as directed by the LDF. Again, more than one Input Section can be placed in any Output Section.
- The linker maps each Output Section into a *Memory Segment*, a contiguous range of memory on the target*, as specified by the LDF. More than one Output Section may map to a single Memory Segment.

Figure 2-1 shows how the LDF file combines information, directing the linker to place program sections in an executable according to the memory available in the DSP system. The LDF file describes the target system and maps your program code within the system memory and processors. The target system memory map and defined segments in your source file are used by the LDF to create an executable program.

Listing 2-1 shows a sample LDF (formatted for easy reading). The LDF file contains two commands (`MEMORY` and `SECTIONS`) that combine program and system information.

* Memory Segments have uniform width. Contiguous addresses on different-width hardware must be in different segments.

Listing 2-1. Sample Linker Description File - Example 1

```

ARCHITECTURE(ADSP-TS001)
SEARCH_DIR( $ADI_DSP\Ts\lib )
$OBJECT1 = main.doj, $COMMAND_LINE_OBJECTS;
/* see Note 1, 2 and 3 on page 2-10 */

MEMORY{
/* see Note 4 on page 2-11 */

    M0Code{ TYPE(RAM) START(0x000000) END(0x00FFFF) WIDTH(32)}
    M1Data{ TYPE(RAM) START(0x080000) END(0x08FFFF) WIDTH(32)}
    M2Data{ TYPE(RAM) START(0x100000) END(0x10FFFF) WIDTH(32)}

} /* End memory command */

PROCESSOR p0{
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

SECTIONS{
/* see Note 2 on page 2-11 */

    code{
        FILL(0xb3c00000)
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS( $OBJECTS( program ))
    } >M0Code
    data1 {INPUT_SECTIONS( $OBJECTS( data1 ))} >M1Data
    data2 {INPUT_SECTIONS( $OBJECTS( data2 ))} >M2Data

    } /*End sections command for processor po */
} /* End processor command */

```

As noted at the beginning of this chapter, you can run the linker from the VisualDSP++ IDE or from the command line. Therefore, all LDF commands (and arguments) have command-line equivalents.

In the following discussion, the salient commands for connecting your program to the target DSP are MEMORY and SECTIONS.

Mapping Files To Memory With An LDF

Notes on Example 1

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

1. `ARCHITECTURE(x)` names the target architecture. It thereby specifies possible memory widths and address ranges, the register set, and other structural information for use by the debugger, linker, loader, splitter and utility software. The target architecture must be installed in VisualDSP++.
2. `SEARCH_DIR` specifies path name(s) to search for libraries and object files. This example shows one search directory, the single argument to the `SEARCH_DIR` command (refer to [page 2-51](#)).

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

If the LDF contains no `SEARCH_DIR` command, the search sequence is the directory from which the linker is called, followed by the VisualDSP++ installation directory and its `lib` subdirectory.

3. `$OBJECTS` expands to a comma-delimited list of object files to be linked together. The LDF supports string macros, making it easier to read; you can substitute short macros for long text strings.

`$ADI_DSP` is the home directory for VisualDSP++. You can also define macros in the LDF. The string macro feature is independent of preprocessor macro support (`#defines`), also available in the LDF.

To prepend `foo.doj` to the string `$OBJECTS`, rewrite the line defining `$OBJECTS` as:

```
$OBJECTS=foo.doj, $COMMAND_LINE_OBJECTS;
```

`$COMMAND_LINE_OBJECTS` (refer to [page 2-48](#)) is another LDF command-line macro, which expands to a list of all the input object file names on the linker's command line. Remember that all linker invocations from the VisualDSP++ IDDE have command-line equivalents.



The link order is determined by the order of the objects. In the default case, when you use default LDFs that ship with the VisualDSP++ tools, all the sections commands use the `$OBJECTS` macro to determine the link order.

The `$OBJECTS` macro includes the `$COMMAND_LINE_OBJECTS` macro that corresponds to the objects specified on the linker command line, in the order specified. The IDDE lists the objects in alphabetical order. However, you may customize the LDF to link objects in any order. Rather than use the `$OBJECTS` macro as the defaults do, each `INPUT_SECTIONS` command could have one or more object names. You can also build your own object list macros in the LDF for use in the `INPUT_SECTIONS` commands.

4. The `MEMORY` command (on [page 2-54](#)) defines the target system's physical memory. Its argument list partitions memory into Memory Segments and assigns labels to each, specifying start and end addresses, memory width, and memory type (program, data, ...). It thereby connects your program to the target system.

Memory Segment names are unique and occupy different namespaces from Input Section and Output Section names.

5. The `OUTPUT()` command (on [page 2-65](#)) directs the linker to produce an executable (`.EXE`) file, specifying the filename. In this listing, the argument to the `OUTPUT` command is the

Mapping Files To Memory With An LDF

`$COMMAND_LINE_OUTPUT_FILE` macro (on [page 2-49](#)) . Therefore, the linker names the executable according to the text following its `-o` switch.

```
linker ... -o outputFile
```

6. `SECTIONS` (on [page 2-66](#)) defines the placement of code and data in physical memory. Based on the three parameters that appear on each line for this command, the linker takes Input Sections as inputs, places them in Output Sections, and maps Output Sections to the Memory Segments declared in the `MEMORY` command.

The `INPUT_SECTIONS` statement specifies the object file that the linker uses to resolve the mapping. The line:

```
dx_e_isr{ INPUT_SECTIONS ($OBJECT1 (isr.tbl) ) } > mem_isr
```

directs the linker to take the `isr.tbl` Input Section, place it in the `dx_e_isr` Output Section, and map it to the `mem_isr` Memory Segment.

For more information on this process, see [“Linker Guide” on page 2-13](#). For information on other LDF commands and syntax, see [“Linker Description File Reference” on page 2-38](#).

Linker Guide

The linker (`linker.exe`) processes your object, library, linker description, and linker command files, producing one or more output files. Linker operations depend on two types of controls: linker options and linker commands.

Linker options let you control how the linker processes your object and library files, letting you select features such as search directories, map file output, and symbol removal among others. These options come from linker command-line switches or (when used within the VisualDSP++ environment) from settings in the `Link` dialog box of the environment's `Project Options` dialog box. These `Link` dialog box settings correspond to command line switches.

Linker commands, in your project's linker description file (LDF), define the memory map of your DSP system and the placement of your program's sections within DSP memory.



The VisualDSP++ environment treats the LDF as a source file in the project's file display, but this file acts only as command input to the linker.

There are a number of operations required for using the linker. All software developers using the linker should be familiar with the following operations:

- [“Describing the Link Target” on page 2-14](#)
- [“Placing Code on the Link Target” on page 2-20](#)
- [“Using Link Features” on page 2-22](#)
- [“Setting Linker Options” on page 2-23](#)
- [“Interpreting Link Errors” on page 2-23](#)

Describing the Link Target

Before defining your DSP system's memory and program placement with linker commands, you must analyze the target DSP system and describe it in terms that the linker can process.

You then produce an LDF for your project. The LDF describes the following:

- Your DSP system's physical memory map
- Your program's placement within your DSP system's memory map



If you do not include an LDF, the linker uses a default linker description file that matches the `-DPROCESSOR` switch on the linker's command line.

Internal Memory

The TigerSHARC contains three blocks of two Mbits each of on-chip, 128-bit wide SRAM. Each memory block is organized as 64K words of 32 bits each. The accesses are pipelined to meet one clock cycle access time needed by the core, DMA or by the external bus. Each access can be of up to four words.

Memories (and their associated buses) are a resource that must be shared between the compute blocks, the IALUs, the sequencer, the external port and the link ports. In general, if during a particular cycle more than one unit in the processor attempts to access the same memory, one of the competing units is granted access, while the other is held off for further arbitration until the following cycle. Refer to the *TigerSHARC Hardware Reference Manual* for more information. This type of conflict only has a small impact on performance due to the very high bandwidth afforded by the internal buses.

An important benefit of large on-chip memory is that by managing the movement of data on- and off-chip with DMA, a system designer can realize high levels of determinism in execution time. Predictable and deterministic execution time is a central requirement in DSP and real-time systems.

Writing a Linker MEMORY Command

The linker's `MEMORY { }` command defines the memory architecture of your DSP system. This lets the linker place your executable file within memory. The steps for writing a linker `MEMORY { }` command are as follows:

1. List the ways that your program uses memory in your system. Some typical uses for memory segments are interrupt table, initialization data, program code, data, heap space, and stack space.
2. List the types of memory in your system and the address ranges of each type of memory. A memory type has several characteristics: RAM, ROM, or PORT; and word width.
3. Using the information in these two lists, write linker `MEMORY { }` command(s), declaring the memory segments in your system.

The example in [Figure 2-2](#) and the steps that follow shows an example TigerSHARC system that uses a C program and details the above steps.

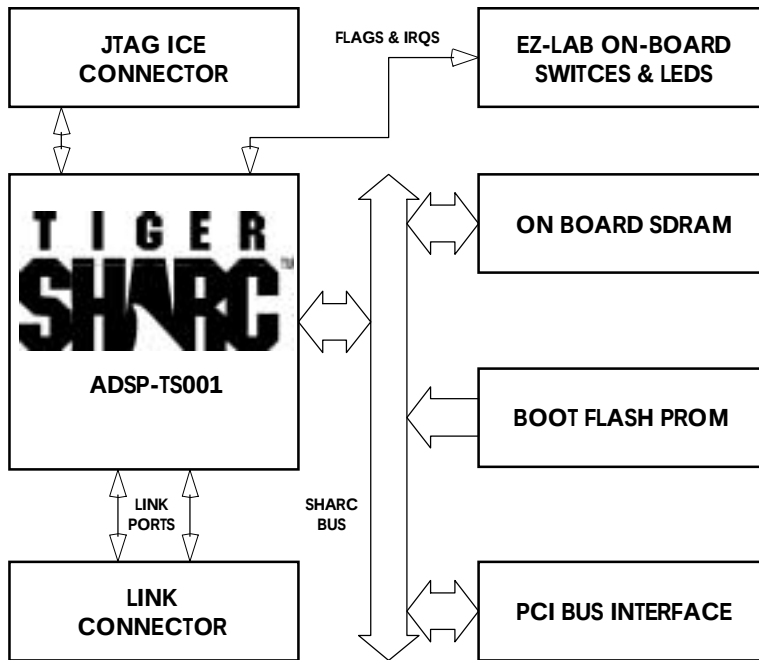


Figure 2-2. Example System for Linker Description

Memory Map

The `MEMORY{}` command defines the memory architecture of your DSP system. This information lets the linker place your executable file in the system's memory.

1. **Memory usage** — Input section names are generated by the compiler or specified by the developer in the source code. Both the memory section names and the output section names are defined in the LDF. While the memory labels are used primarily within the LDF, the outputs section labels can be important to downstream tools. For example, one can invoke `elfdump` to dump the contents of the "data1" section of an executable file.

The memory names are defined in the MEMORY() command; the output section names are defined in the SECTIONS() command. The default LDF has been constructed to handle all the sections that might be generated by the compiler (the column "Input Section"). The produced .dxe file has appropriate "Output Section(s)" for which material (the corresponding "input section(s)") was found in the inputs.

The correspondences used in the TigerSHARC default LDF (ADSP-TS001.ldf) are:

<u>Input Section</u>	<u>Output Section</u>	<u>Memory</u>
program	code	M0Code
data1	data1	M1Data
data2	data2	M2Data
heaptab	heaptab	M1Data
M1Stack	jstackseg	M1Stack
M2Stack	kstackseg	M2Stack
M1Heap	defheapseg	M1Heap
ctor	ctor	M1Data
ctor0	ctor	M1Data
ctor1	ctor	M1Data
ctor2	ctor	M1Data
ctor3	ctor	M1Data

2. Memory characteristics — In the example system, the ADSP-TS001 DSP has internal memory addresses from 0x0 to 0x17ffff, and the characteristics of this memory appear in [Table 2-1](#).

Table 2-1. ADSP-TS001 Memory Structure

Block	Range	Word Size
M0 Memory Block	0x00000000 - 0x0000ffff	32-bit instructions or 16-, 32, or 40-bit data
	0x00010000 - 0x0007ffff	Reserved
M1 Memory Block	0x00080000 - 0x0008ffff	32-bit instructions or 16-, 32, or 40-bit data
	0x00090000 - 0x0009ffff	Reserved
M2 Memory Block	0x00100000 - 0x0010ffff	32-bit instructions or 16-, 32, or 40-bit data
	0x00110000 - 0x0017ffff	Reserved
Internal Registers	0x0018 0000 - 0x0018 07ff	Control, status, and I/O registers
	0x0018 0800 - 0x01bf ffff	Reserved
	0x01c0 0000 - 0x03ff ffff	Broadcast and multiprocessor (not used in LDF)
SDRAM	0x0400 0000 - 0x07ff ffff	32-bit instructions or 16-, 32, or 40-bit data

3. Linker `MEMORY{}` Command — referring to steps 1 and 2, you specify the TigerSHARC memory with the `MEMORY{}` command in [Listing 2-2](#).

Listing 2-2. Linker `MEMORY{}` Command Code -- Example 2

```
/* start TS001_memory.ldf file that is referred to in
the “LDF Programming Examples” on page 2-76 */

// Internal memory blocks are 0x10000 (64k)

// MEMORY { /* (this line commented for include purposes) */

M0Code {TYPE(RAM) START(0x00000000) END(0x0000FFFF) WIDTH(32)}
M1Data {TYPE(RAM) START(0x00080000) END(0x0008BFFF) WIDTH(32)}
M1Stack{TYPE(RAM) START(0x0008C000) END(0x0008FFFF) WIDTH(32)}
M2Data {TYPE(RAM) START(0x00100000) END(0x0010BFFF) WIDTH(32)}
M2Heap {TYPE(RAM) START(0x0010C000) END(0x0010C7FF) WIDTH(32)}
M2Stack{TYPE(RAM) START(0x0010C800) END(0x0010FFFF) WIDTH(32)}
SDRAM {TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)}
MS0 {TYPE(RAM) START(0x08000000) END(0x0BFFFFFF) WIDTH(32)}
MS1 {TYPE(RAM) START(0x0C000000) END(0x0FFFFFFF) WIDTH(32)}
HOST {TYPE(RAM) START(0x10000000) END(0xFFFFFFFF) WIDTH(32)}

// end TS001_memory.ldf file
```

Notes on Example 2

[Listing 2-2](#) applies to the preceding discussion of how to write a `MEMORY` command, and to the following discussion of the `SECTIONS` command. The `SECTIONS` command is not atomic: it can be interspersed with other directives, including location counter information. You can define new symbols within the LDF; this example defines the starting stack address, the highest possible stack address, and the heap’s starting location and size. These newly-created symbols will be entered in the executable’s symbol table.

Placing Code on the Link Target

The `SECTIONS{}` command lets you place program `.SECTIONS` within the memory of a DSP system that you defined with the linker's `MEMORY{}` command. You must let the linker know the type of link to perform by embedding any `SECTIONS{}` commands within the linker's `PROCESSOR{}` or `SHARED_MEMORY{}` commands. These commands tell the linker to place the code in memory for a single processor, shared among multiple processors, or shifted between overlays. The steps for writing a linker `SECTIONS{}` command are as follows:

1. List all the `.SECTIONS` in your DSP program, identifying their memory type and noting when location is critical to their operation. (These `.SECTIONS` include interrupt tables, data buffers, and on-chip code or data).
2. Compare this list with the segments you defined in your `MEMORY{}` command, identifying the segments in which `.SECTIONS` must be placed.
3. Using the information in these two lists, write one or more linker `SECTIONS{}` command(s).

The TigerSHARC system in [Figure 2-2](#) and any simple C program provide a generic example for exercising this analysis technique. The following is an example system description.

1. Program sections — The TigerSHARC C/C++ compiler produces code that uses memory in predefined ways and has predefined section labels.
2. Linker `MEMORY{}` Command — A `MEMORY{}` command for the TigerSHARC DSP appears in [Listing 2-2](#). This command declares segments that correspond to the output sections produced by the compiler.

3. Linker `SECTIONS{}` Command — Combining the information from steps 1 and 2, you could specify how to place code for the TigerSHARC EZ-LAB with the `SECTIONS{}` command in [Listing 2-3](#). Also see “[LDF Programming Examples](#)” on page 2-76.

Listing 2-3. Linker `SECTIONS{}` Command Example

```
// SECTIONS {

// begin output sections

// places code (instructions) in M0 (internal memory)
code{
    FILL(0xb3c00000)
    INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS( $OBJECTS(program))
} >M0Code

// places data in M2 (internal memory)
data1{
    INPUT_SECTIONS( $OBJECTS(data1))
} >M2Data

// places data in M1 (internal memory)
data2{
    INPUT_SECTIONS( $OBJECTS(data2))
} >M1Data

// places c rtl initializers in M2 (internal memory)
ctor{
    INPUT_SECTIONS( $OBJECTS(ctor0))
    INPUT_SECTIONS( $OBJECTS(ctor1))
    INPUT_SECTIONS( $OBJECTS(ctor2))
    INPUT_SECTIONS( $OBJECTS(ctor3))
    INPUT_SECTIONS( $OBJECTS(ctor))
} >M2Data

// places c rte heap table in M2 (internal memory)
heaptab{
    INPUT_SECTIONS( $OBJECTS(heaptab))
} >M2Data
```

Linker Guide

```
// places c rte J IALU stack in M2 (internal memory)
jstackseg{
    ldf_jstack_limit = .;
    ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
} >M2Stack

// places c rte K IALU stack in M1 (internal memory)
kstackseg{
    ldf_kstack_limit = .;
    ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
} >M1Stack

// places c rte heap in M2 (internal memory)
defheapseg{
    ldf_defheap_base = .;
    ldf_defheap_size = MEMORY_SIZEOF(M2Heap);
} >M2Heap

// end sections
```

Using Link Features

The two previous sections, [“Describing the Link Target” on page 2-14](#) and [“Placing Code on the Link Target” on page 2-20](#), provide an overview of how to link executables for single processor systems. The linker’s advanced features support linking executables for systems with multiprocessor memory, shared memory, and overlay memory. For more information on these topics, see the following sections:

- For information on multiprocessor memory, see the command syntax for [“MPMEMORY{ }” on page 2-56](#) and the example in [“Linking for Multi-Processor and Shared Memory” on page 2-80](#).
- For information on shared memory, see the command syntax for [“SHARED_MEMORY{ }” on page 2-73](#) and the example in [“Linking for Multi-Processor and Shared Memory” on page 2-80](#).

- For information on overlay memory, see the command syntax for [“SECTIONS{ }” on page 2-67](#) and [“PLIT{ }” on page 2-61](#). Examples of overlay linking appear in [“Linking for Overlay Memory” on page 2-86](#) and [“Using a Procedure Linkage Table” on page 2-90](#).

The flexible nature of the linker’s command language lets you make linking as simple or as complex as fits your needs. To write simple, maintainable linker description files, use the linker’s predefined macros for file searches, input, and output. [For more information, see “LDF Macros” on page 2-47.](#)

Setting Linker Options

When developing within the VisualDSP++ environment, you can set default tool settings for all files in a project. You may find it useful to modify the linker’s default option settings in the VisualDSP++ environment. For more information, see the *VisualDSP++ User’s Guide for TigerSHARC DSPs*.

The linker also has command-line switches that correspond to the `Link` dialog’s options. For more information on the link operation, see [“Linker Command-Line Reference” on page 2-25](#).

Interpreting Link Errors

The linker reports link warnings and errors to standard out in the command line version of the linker (or in the VisualDSP++ environment Output Bar). Linker warning or error messages describe errors that the linker encountered when processing the linker description file.

A linker *warning* message indicates a processing error which does not keep the linker from producing a valid output file.

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

brief description of the error condition. The following is an example error message:

```
[Error E2005] my_file.ldf:177 Expected token was not found.:  
Expected 'Eof' Before '3'
```

This error indicates that the linker encountered the character 3 on line 177 of the `my_file.ldf` file, indicating that this character appeared where the linker expected the end-of-file.

When developing within the VisualDSP++ environment, the Output window displays project build status and error messages. In most cases, you can double click on a message or error number and the IDE jumps to the source file and the line number that contains the error. However, some build errors—such as bad or missing cross reference to an object or executable file—do not correlate directly to source files. For more information, see the *VisualDSP++ User's Guide for TigerSHARC DSPs*.



Often, errors relating to missing cross references stem from omissions in the LDF file. For example, a segment from the objects is not placed by the LDF, creating a cross reference error in every object that refers to labels in the missing segment. Accounting in the LDF file for all segments that need placement can help solve this sort of error.

Linker Command-Line Reference

This section provides reference information on the linker command-line switches. A list of all switches appears in [Table 2-3 on page 2-30](#), and a description of each switch appears in [“Linker Switch Option Descriptions” on page 2-32](#).

The linker generates an executable program by linking together one or more assembled object (`.DOJ`) files. The linker’s primary output is one or more executable (`.DXE`) files. To make an executable file, the linker processes data from a linker description file and one or more object files.

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

 When using the linker within the VisualDSP++ environment, the settings in the IDDE correspond to linker command-line switches. The IDDE calls the linker with those settings when you link your code. For more information, see the *VisualDSP++ User’s Guide for TigerSHARC DSPs*.

Command-Line Syntax

The general format for a linker command line is shown below.

```
linker sourcefile [sourcefile ...] -switch [-switch ...]
```

Only `linker` (the command itself) and at least one `object` (an object filename) are obligatory. Other switches are optional, and some commands are mutually exclusive. Some other information, such as `architecture`, must be provided for the link to proceed, but that information is almost always subsumed in the *LdfFilename* following the `-T` switch.

For the ADSP-TS001DSP, the linker command may look like this:

```
linker -DADSP-TS001 p0.doj p1.doj -T target.ldf -t -o program.dxe
```

Object Files in the Linker Command Line

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

- The standard object file is produced by the assembler and has a `.DOJ` extension.
- It may also be an archive (library) of one or more files, with a `.DLB` extension. These may be sourced from the VisualDSP++ libraries, or from your own code.
- It may also be an executable (`.EXE`) file to be linked against*.



Object file names are not case-sensitive, but linker switches *are* case-sensitive; for example, `linker -t` is not the same as `linker -T`.

An object filename has the following characteristics:

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

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

[Table 2-2 on page 2-29](#) lists the allowed extensions and matching linker operations.

* “Link Against” is described on [page 2-47](#), under `$COMMAND_LINE_LINK_AGAINST`

Switch Format in the Linker Command Line

The linker has many optional switches. They select the operations and modes for the compiler and other tools. The standard linker switch syntax is as follows:

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

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

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

calls the linker as shown below. Note the difference between the `-T` and `-t` switches:

- `p0.doj`, `p1.doj` and `p2.doj` — Links object files together into an executable
- `-T target.ldf` — Selects a linker description file, controlling executable program placement
- `-t` — Turns on trace information, echoing each link object's name as it is processed
- `-o program.dxe` — Selects a name for the linked, executable output

Filenames on the Linker Command Line

Many linker switches take a file name as an optional parameter. [Table 2-2 on page 2-29](#) lists the type of extensions that the linker expects on file-name arguments. The linker supports relative and absolute path names when searching for default or user-selected directories.

File searches occur as follows:

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

For more information on file search, see [“LDF Macros” on page 2-47](#).

The linker follows the conventions for file name extensions listed in [Table](#) . When you provide an input or output file name as a command-line parameter, use the following guidelines:

- Use a space to delimit file names in a list of input files.
- Enclose long file names within straight quotes, "long file name".
- Append the appropriate file name extension to each file.

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

Table 2-2. File Name 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

Linker Command-Line Options

This section lists and describes the linker's command-line options (switches) in ASCII collation order: upper- and lower-case options are alphabetized separately. A brief description of each option includes information on case sensitivity, equivalent switches, switches overridden/contradicted by the one described, and naming and spacing constraints on parameters.

- Switches may be used in any order on the command line. Items shown in [] are optional; items in *italics* are user-defined and are described with each switch.
- Filenames and pathnames may be relative or absolute.
- Filenames containing white space or colons must be enclosed by double quotation marks, though relative pathnames such as `..\..\foo.dxe` do not.

Linker Command-Line Reference

The switches in Table 2-3 may be used in any order on the command line. “[Linker Switch Option Descriptions](#)” on [page 2-32](#) provides detailed description of each linker switch.

Table 2-3. Linker Switch Summary

Switch Name	Description
<code>objects</code> on page 2-32	Specifies object files involved in the linking; process files that are not parameters to a switch.
<code>@ file</code> on page 2-32	Directs the linker to use the specified file as input on the command line.
<code>-Darchitecture</code> on page 2-33	Specifies the target architecture (processor).
<code>-e</code> on page 2-33	Directs the linker to eliminate unused symbols from the executable.
<code>-es secName</code> on page 2-33	Names sections (<i>secName</i> list) to which elimination algorithm is being applied.
<code>-ev</code> on page 2-33	Eliminate unused symbols from the executable verbosely, displaying all objects that were eliminated.
<code>-help</code> on page 2-33	Outputs the list of command-line switches and exits.
<code>-i path</code> on page 2-33	Includes search directory for preprocessor include files.
<code>-ip</code> on page 2-34	Fills in fragmented memory with individual data objects that fit.
<code>-jcs21</code> on page 2-34	Directs the linker to convert out-of-range short calls and jumps to the longer form.
<code>-keep symName</code> on page 2-34	Keeps unused symbols.
<code>-L path</code> on page 2-35	Adds the path name to search libraries for objects..
<code>-M</code> on page 2-35	Produces dependencies.

Table 2-3. Linker Switch Summary (Cont'd)

Switch Name	Description
-MM on page 2-35	Builds and produces dependencies.
-Map <i>filename</i> on page 2-35	Outputs a map of link symbol information to a file.
-MD <i>macro</i> [= <i>def</i>] on page 2-35	Defines and assigns value <i>def</i> to preprocessor <i>macro</i> .
-o <i>filename</i> on page 2-36	Outputs the named executable file.
-pp on page 2-36	Stops after preprocessing.
-S on page 2-36	Omits debugging symbol information from the output file.
-s on page 2-36	Strips all symbol information from the output file.
-sp on page 2-36	Skips preprocessing.
-T <i>filename</i> on page 2-36	Names the LDF to specify executable program placement.
-t on page 2-37	Directs the linker to output the names of link objects.
-v on page 2-37	Verbose output -- directs the linker to output status information.
-version on page 2-37	Directs the linker to output its version and exit.
-warnonce on page 2-37	Warns only once for each undefined symbol.
-xref <i>filename</i> on page 2-37	Outputs a list of all cross-referenced symbols.

Linker Command-Line Reference

Linker Switch Option Descriptions

objects (filename)

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

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

The linker treats object (.obj) and library (.lib) files that appear on the command line as object files to be linked. For more information on objects, see the \$COMMAND_LINE_OBJECTS linker macro on [page 2-48](#).

The linker treats executable (.dxe) and shared-memory (.sm) files that appear on the command line as executables to be linked against. For more information on executables, see the \$COMMAND_LINE_LINK_AGAINST linker macro on [page 2-48](#). If you do not specify any link objects on the command line or in the linker description file, the link fails.

<null>

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

@ file

Uses *file* as input to the linker command line.
This switch lets you circumvent environmental command-line length restrictions. The *file* may not start with “linker” (it cannot be a linker

command line). Any whitespace in *file* serves to separate tokens, including a newline.

-Darchitecture

The **-D** (default processor) switch specifies target architecture/processor. Valid processors are ADSP-TS001 and ADSP-TS101.

No whitespace is permitted between **-D** and *architecture*.

PROCESSOR name is case-sensitive, and must be available in your

VisualDSP++ installation. This option must be used if no LDF is specified on the command line (see **-T** option). It also must be used if the specified LDF does not specify *architecture()*. Architectural inconsistency between this option and an LDF is an error.

-e

Eliminates unused symbols from the executable.

-es secName

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

-ev

Eliminates unused symbols and verboses — reports on each symbol eliminated.

-help

The **-help** (command-line help) switch displays a summary of command-line options and exits.

-i path

The **-i** (include search directory) switch directs the preprocessor to append the directory to the search path (search directory) for include files.

Linker Command-Line Reference

-ip



This feature is not implemented in the current linker release for ADSP-TSxxx DSPs.

Fills in fragmented memory with individual data objects that fit.

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

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

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

The `-noip` option (in assembler) turns off the individual placement option. See the *VDSP++ 2.0 Assembler and Preprocessor Manual for TigerSHARC DSPs*.

-jcs2l

Directs the linker to convert out-of-range short calls and jumps to the longer form.

-keep symName

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

-L *path*

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

-M

-M (produce dependencies) directs the linker to do a dependency check and to output the result to `stdout`.

-MM

-MM (build and produce dependencies) directs the linker do a dependency check and output the result to `stdout`, and also to do a build.

-Map *filename*

-Map (symbol map file) directs the linker to output a map of link symbol information to *file*, which can have any name. Conventionally, the file parameter is obligatory and has a `.MAP` extension, provided by the Linker. The whitespace is obligatory before file. Otherwise, the link fails.

-MDmacro[=*def*]

-MD (define macro) directs the linker to define and assign a value *def* to a preprocessor *macro*. For example, the linker's `-MDF00=BAR...` means the code following `#ifdef F00==BAR` in the LDF is executed (but not the code following `#ifdef F00==XXX`). If `=def` is not included, *macro* is defined and set to "1", so code following `#ifdef F00` is executed. May be repeated.

Linker Command-Line Reference

-o *filename*

-o outputs executable file with the specified *filename*. If the *filename* is not specified, the linker outputs a “.dxe” file in the project’s home directory. Alternatively, you may use the LDF’s **OUTPUT** command to name the output file.

-pp

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

-S

-S (strip debug symbols) omits debugging symbol information (*not* all symbol information) from the output file. Compare with **-s** switch (on [page 2-36](#)).

-s

-s (strip all symbols) strips all symbol information from the output file.

-sp

-sp (skip preprocessing) links without preprocessing the LDF.

-T *file*

The **-T** (linker description file) switch directs the linker to use the file as the linker description file, specifying executable program placement.

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

The *file* must exist and can be found (e.g. via the **-L** option); there must be whitespace before *file*. The file’s name is unconstrained, but must be

valid; for example, `a.b` works if it is a valid LDF. The `.LDF` is a valid extension but not a requirement. Multiple `-T` options accumulate.

-t

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

-v

`-v` (verbose) outputs status information while linking.

-version

Outputs linker's version to `stderr` and exit.

-warnonce

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

-xref *filename*

`-xref` (external reference file) outputs a list of all cross-referenced symbols (and where they are used) in the link to the named file.

Linker Description File Reference

The LDF lets you develop code for any system that contains a DSP. The syntax of the LDF lets you define your system to the linker and specify how the linker processes executable code for your system. This reference describes LDF syntax and provides LDF examples for typical systems. As you read selections from the following topics, refer to the examples to see applications of LDF syntax:

- [“LDF Structure” on page 2-39](#)
- [“LDF Expressions and Conventions” on page 2-40](#)
- [“Miscellaneous LDF Keywords” on page 2-44](#)
- [“LDF Macros” on page 2-47](#)
- [“LDF Commands” on page 2-51](#)
- [“LDF Programming Examples” on page 2-76](#)



Because the linker runs the preprocessor on the LDF file, you can use any of the preprocessor's commands (such as `#define`) within your LDF file. For more information on preprocessor commands, see the *Assembler and Preprocessor Manual for TigerSHARC DSPs*.

LDF Structure

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

- Divide your LDF into a set of `PROCESSOR{ }` commands, one for each DSP in your system.
- Put `MEMORY{ }` commands in the LDF scope that matches your system, defining memory that is unique to a processor within the scope of the corresponding `PROCESSOR{ }` command. Define common memory definitions (shared or multiprocessor memory) in the global LDF scope.
- Place `MPMEMORY{ }` or `SHARED_MEMORY{ }` commands in the global LDF scope if they apply to your system. These commands represent system resources that can apply to multiple processors.

For more information on the LDF structure, see [“Describing the Link Target” on page 2-14](#), [“Placing Code on the Link Target” on page 2-20](#), and [“LDF Programming Examples” on page 2-76](#).

Command Scoping

The two LDF file scopes are global (to the entire LDF) and local to a command (within the LDF). A local scope defines the content for an `OUTPUT( )` command. You can use an `OUTPUT( )` command within the scope of a `PROCESSOR{ }` and `SHARED_MEMORY{ }` command. The global scope occurs outside commands. Commands and expressions that appear in the global scope are available in the global scope and visible in all subsequent scopes. The effects of commands and expressions that appear in the local scopes are limited to those scopes. Note that LDF macros are available globally regardless of the scope where the macro is defined.

Linker Description File Reference

Figure 2-3 demonstrates some scoping issues. For example, the `MEMORY{}` command that appears in the global LDF scope is available in all the local scopes, but the `MEMORY{}` commands that appear in the local scopes are restricted to those scopes.

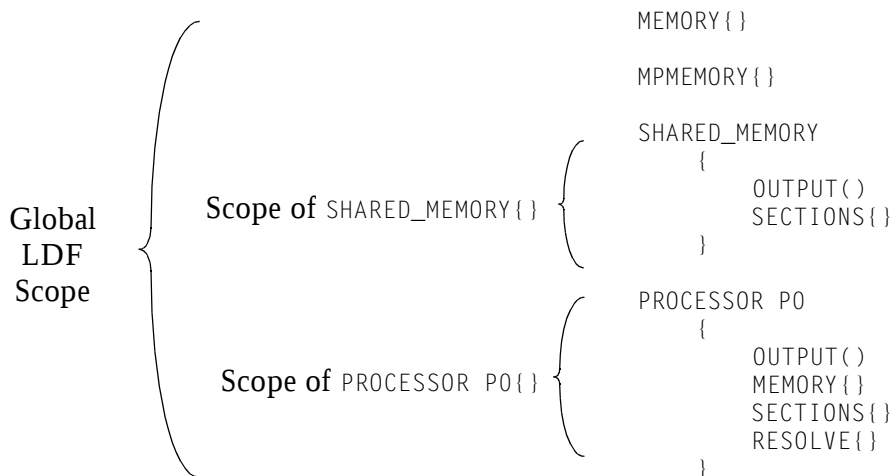


Figure 2-3. LDF Command Scoping Example

LDF Expressions and Conventions

Table 2-4 lists the linker's non-keyword operators and conventions.

Table 2-4. Linker Non-Keyword Operators & Conventions

Convention	Description
.	A dot "." in an address expression refers to the current location pointer.
0xnumber	A "0x" prefix indicates a hexadecimal base number.

Table 2-4. Linker Non-Keyword Operators & Conventions (Cont'd)

Convention	Description
<i>number</i>	A number without a prefix is a decimal base number.
<i>numberk</i> (or <i>K</i>)	A “k” suffix indicates a decimal base <i>number</i> multiplied by 1024.
<code>/* comment */</code>	A “/* */” string encloses C style comments.
<code>// comment</code>	A “//” string precedes single-line C++ style comments.

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

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

Linker Keywords

Descriptions of the linker keywords from [Table 2-5](#) appear in the following sections:

- [“Miscellaneous LDF Keywords” on page 2-44](#)
- [“LDF Operators” on page 2-44](#)
- [“LDF Macros” on page 2-47](#)
- [“LDF Commands” on page 2-51](#)
- [“LDF Commands” on page 2-51](#)

The keywords in [Table 2-5](#) are case-sensitive; the linker only recognizes a keyword when the entire word is in UPPERCASE letters.

Table 2-5. Linker Keywords

ABSOLUTE on page 2-44	ADDR on page 2-45	ALGORITHM on page 2-72
ALIGN on page 2-45	ALL_FIT on page 2-72	ARCHITECTURE on page 2-52
BEST_FIT on page 2-72	BOOT on page 2-44	DEFINED on page 2-45
DM ¹	ELIMINATE on page 2-45	ELIMINATE_SECTIONS on page 2-45
END on page 2-55	FILL on page 2-70	FIRST_FIT on page 2-72
INCLUDE on page 2-51	INPUT_SECTION_ALIGN on page 2-45	INPUT_SECTIONS on page 2-45
KEEP on page 2-46	LENGTH on page 2-55	LINK_AGAINST on page 2-51
MAP on page 2-51	MEMORY on page 2-54	MEMORY_SIZEOF on page 2-46

Table 2-5. Linker Keywords

MPMEMORY on page 2-56	NUMBER_OF_OVERLAYS on page 2-72	
OUTPUT on page 2-65	OVERLAY_ID on page 2-72	OVERLAY_INPUT on page 2-72
OVERLAY_GROUP on page 2-57	OVERLAY_OUTPUT on page 2-72	PACKING ¹
PLIT on page 2-61	PLIT_DATA_OVERLAY_IDS on page 2-64	PLIT_SYMBOL_ADDRESS on page 2-64
PLIT_SYMBOL_OVERLAY ID on page 2-64	PM ¹	PORT on page 2-55
PROCESSOR on page 2-64	RAM on page 2-55	RESOLVE on page 2-66
RESOLVE_LOCALLY on page 2-72	ROM on page 2-55	SEARCH_DIR on page 2-77
SECTIONS on page 2-67	SHARED_MEMORY on page 2-73	SHT_NOBITS on page 2-68
SIZE on page 2-72	SIZEOF on page 2-46	START on page 2-55
TYPE on page 2-55	VERBOSE on page 2-45	WIDTH on page 2-55

¹ These keywords do not apply for TigerSHARC DSP linker description files.

Miscellaneous LDF Keywords

This section describes the linker keywords that are not operators, macros, or commands.

ABSOLUTE—An expression operator that returns the non-relocatable value of an expression.

BOOT—Boot memory, memory from which a TigerSHARC DSP can be booted.

FALSE—A constant with the value of 0.

TRUE—A constant with the value of 1.

XREF—A cross-reference option setting.

For more information about linker keywords that are operators, see [“Miscellaneous LDF Keywords” on page e2-44](#). For more information about linker keywords that are macros, see [“LDF Macros” on page 2-47](#). For more information about linker keywords that are commands, see [“LDF Commands” on page 2-51](#).

LDF Operators

LDF operators in expressions support memory address operations. Expressions that contain these operators terminate with a semicolon, except when you use the operator as a variable for an address. The linker supports the following LDF operators:

- **ABSOLUTE**(*address_expression*)

The linker returns the absolute, non-relocatable address of the *address_expression* on a call to **ABSOLUTE**(). This operator is useful for assigning an absolute address to a symbol.

- `ADDR(section_name)`

The linker returns the absolute, non-relocatable address of the *section_name* on a call to `ADDR()`. This operator is useful for assigning a section's absolute address to a symbol.

- `ALIGN(address_boundary_expression)`

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

- `DEFINED(symbol)`

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

- `ELIMINATE()`

The linker turns on object elimination, eliminating symbols from the executable if they are not called. If the `VERBOSE` keyword is added (for example, `ELIMINATE(VERBOSE)`), the linker reports on objects as they are eliminated.

- `ELIMINATE_SECTIONS( sectionList )`

The linker turns on section elimination, eliminating the listed sections from the executable if they are not called. The *sectionList* is the comma delimited list of sections.

- `INPUT_SECTION_ALIGN( address_boundary_expression )`

The `INPUT_SECTION_ALIGN()` operator is only valid within the scope of an output section. For more information, see [“Command Scop-](#)

Linker Description File Reference

ing” on page 2-39. For more information on output sections, see the syntax description for “SECTIONS{ }” on page 2-67.)

The linker fills any “holes” created by the `INPUT_SECTION_ALIGN()` instructions with zeroes (by default) , or with the value specified with the preceding `FILL` command valid for the current scope. For more information on `FILL`, see page 2-70.

The linker aligns each input section (instruction or data) placed in an output section with the address specified by the *address_boundary_expression*. The address boundary expression (must be a power of 2) is a word boundary (address). Legal values for this expression depend on the word size of the segment that receive the output section being aligned. For ADSP-TS001 DSPs, the linker aligns instructions lines with quad-word boundaries when the `code` output section is set to `INPUT_SECTION_ALIGN(4)`.

- `KEEP( keepList )`

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

- `MEMORY_SIZEOF(segment_name)`

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

- `SIZEOF(section_name)`

The linker returns the size, in 8-bit bytes, of the section, *section_name*. This operator is useful when knowing a section’s size helps with moving the current location counter to an appropriate location.

- The current location counter (`.`)

The linker treats a `.` (period) surrounded by spaces as the symbol for the current location counter. Because the “`.`” only refers to a location in an output section, this operator may only appear within the `SECTIONS{}` command. Note that the `.` operator starts at the start address of the target memory segment and increments through the segment’s addresses; it may not be decremented, or moved backwards, except for the `OVERLAY_INPUT()` portion of the `SECTIONS{}` command. Observe the following rules when manipulating the `.` operator:

- Use the `.` operator anywhere that a symbol is allowed in expressions.
- Note that assigning a value to the `.` operator moves the location counter, leaving voids or gaps in memory.

LDF Macros

Macros are names of text strings. They may be assigned values (textual or procedural) used to substitute the macro reference(s). Programs encountering these identifiers in their input files can operate on macros in several ways:

- Substitute the string value for the name. Normally the string value is longer than the name, so the macro “expands” to its textual length.
- Perform actions that are conditional on the existence or value of the macro.
- Assign a value to the macro, possibly as the result of a procedure, then use that value in further processing.

The linker supports all three treatments of macros it encounters in the LDF. Some macros are built in, with predefined procedures or values,

Linker Description File Reference

which may be system-specific. These are called linker (or LDF) macros, and are described below. Others, user macros, are user-defined.

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

LDF macros funnel the input from the linker command line into pre-defined macros and provide support for user-defined macro substitutions. Linker macros are available globally in the LDF regardless of where they are defined. For more information on these topics, see [“Command Scoping” on page 2-39](#) and [“LDF Macros & Command Line Interaction” on page 2-50](#).



The LDF macros are independent of preprocessor macro support (`#defines`), also available in the LDF. Preprocessor macros (or other preprocessor commands) are placed by the preprocessor into source files. Preprocessor macros are useful for repeating instruction sequences in your source code. These macros facilitate text replacement, file inclusion, and conditional assembly and compilation. In addition, the `#define` preprocessor commands define symbolic constants.

LDF Macro List

The linker supports the following LDF macros:

- `$COMMAND_LINE_OBJECTS`

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

- `$COMMAND_LINE_LINK_AGAINST`

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

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

- `$COMMAND_LINE_OUTPUT_FILE`

The linker expands this macro into the output executable (`.DxE` or `.SM`) file name, which is set with the linker's `-o` switch. You should use this macro only once in your LDF for file name substitution within an `OUTPUT()` command.

- `$ADI_DSP`

The linker expands this macro into the path to the installation directory. This macro is useful when controlling how the linker searches for files.

- `$macro = list_of_files ;`

The linker supports user-defined macros for file lists. Using the syntax above, you can define the `$macro` as a comma delimited `list_of_files`. After you define the `$macro`, the linker substitutes the `list_of_files` for the `$macro` where it subsequently appears in the LDF. Terminate a `$macro` declaration with a semicolon. The linker processes the files in the order that they appear.

LDF Macros & Command Line Interaction

Whether you run the linker within the VisualDSP++ environment or from a command line, the linker gets its commands through a command-line interface. Many (but not all) linker operations, such as input, output, and link against files can be controlled through the command line. Using LDF macros, you can apply these command-line inputs throughout your LDF file. [For more information, see “LDF Macros” on page 2-47.](#)

Whether you should use the command-line inputs in the LDF *or* control the linker with LDF code depends on the following two results.

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

The command-line interface for the linker does not let you name the multiple outputs that you need for multiprocessor, shared memory, or overlay memory.

2. Writing an LDF that *does not use* command-line inputs can produce a more specific LDF that you can use with more complex linker features. The environment interface for the linker lets you name the multiple outputs that you need for multiprocessor, shared memory, or overlay memory.

LDF Commands

Commands in the LDF define the target system and specify the order in which the linker is processing the output for that system. Linker commands operate within a scope, influencing the operation of other commands that appear within the range of that scope. [For more information, see “Command Scoping” on page 2-39.](#) This section describes the following LDF commands:

- [“ARCHITECTURE\(\)” on page 2-52](#)
- [“INCLUDE\(\)” on page 2-52](#)
- [“LINK_AGAINST\(\)” on page 2-53](#)
- [“MAP\(\)” on page 2-53](#)
- [“MEMORY{” on page 2-54](#)
- [“MPMEMORY{” on page 2-56](#)
- [“OVERLAY_GROUP{” on page 2-57](#)
- [“PACKING\(\)” on page 2-60](#)
- [“PLIT{” on page 2-61](#)
- [“PROCESSOR{” on page 2-64](#)
- [“SEARCH_DIR\(\)” on page 2-66](#)
- [“SECTIONS{” on page 2-67](#)
- [“SHARED_MEMORY{” on page 2-73](#)

ARCHITECTURE()

Specifies the processor in your target system. Your LDF may contain only one `ARCHITECTURE()` command, and this command must appear with global scope, applying to the entire LDF.

Syntax: `ARCHITECTURE(processor)`



The `ARCHITECTURE()` command is case-sensitive. Hence, `ADSP-TS001` is a legal value, but `adsp-TS001` is not.

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



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

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

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

INCLUDE()

The linker's `INCLUDE()` command specifies an additional file that the linker processes while processing the current linker description file. You may specify any number of additional files. Indicate one include file per `INCLUDE()` command and enclose long file names within straight quotes, "long file name".

LINK_AGAINST()

To link multiprocessor programs for ADSP-TSxxx processors, you must use the `LINK_AGAINST()` command in your linker description file (`.ldf`).

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

`LINK_AGAINST()` is an optional part of the `PROCESSOR{}`, `SHARE_MEMORY{}`, `OVERLAY_INPUT{}` command.

The syntax for the `LINK_AGAINST()` command is:

```
PROCESSOR processor_name {
    LINK_AGAINST (executable_file_names)
}
```

where:

- `n` is the processor name (typically 0,1,...)
- `executable_file_names` is a list of one or more executable (`.DxE`) or shared memory (`.SM`) files. If a list of file names is given, the names are separated by whitespace.

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

MAP()

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

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

MEMORY{}

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

Your linker description file may contain a `MEMORY{}` command that applies to each scope of your LDF, but there is no limit to the number of segments you can declare within each `MEMORY{}` command. [For more information, see “Command Scoping” on page 2-39.](#) The `MEMORY{}` command should be followed by the `SECTIONS{}` command. Use the memory segment names for placing program `.SECTIONS`. Only memory segment declarations can appear within the `MEMORY{}` command.

If you do not specify the target processor's memory map with the `MEMORY{}` command, the linker cannot link your program. If the combined sections directed to a segment require more space than is in the segment, the linker issues an error message and halts the link. The syntax for the `MEMORY{}` command appears in [Figure 2-4](#).

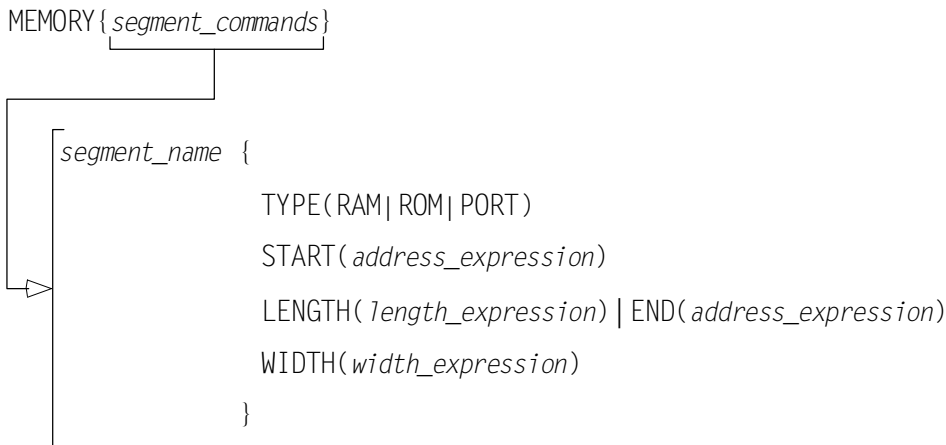


Figure 2-4. Syntax Tree of the `MEMORY{}` Command

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

- *segment_commands*

Declares your target processor's memory segments. While your linker description file may contain only one `MEMORY{ }` command that applies to each scope of the LDF, there is no limit to the number of segments that you can declare within each `MEMORY{ }` command. Each segment declaration must contain the following parameters: a *segment_name*, a `TYPE( )` command, a `START( )` command, a `LENGTH( )` or `END( )` command, and a `WIDTH( )` command.

- *segment_name*

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

- `TYPE( RAM | ROM | PORT )`

Identifies the architecture-specific type of memory within the segment. (Note: this is only for documentation and the specified string does not have any special meaning.) The linker stores this information in the executable for use by other development tools. A valid `TYPE( )` command specifies the memory type: `RAM`, `ROM`, or `PORT`.

- `START( address_expression )`

Identifies the start address of the memory region. The *address_expression* must be an absolute address or an expression that evaluates to an absolute address.

- `LENGTH( length_expression ) | END( address_expression )`

Identifies the length of the memory region in bytes or sets the end address of the memory region. If stating the length, the *length_expression* must be the number of addressable words within the region or an expression that evaluates to the number of words.

Linker Description File Reference

If stating the end address, the *address_expression* must be an absolute address or an expression that evaluates to an absolute address, such as: `START + LENGTH = END`

- `WIDTH(width_expression)`

Identifies the width in bits of memory words within the segment. The *width_expression* must be a number or an expression that evaluates to that number. Note: For TigerSHARC DSPs, this number must be 32.

MPMEMORY{}

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

Your linker description file may contain only one `MPMEMORY{}` command, and the number of processors that you can declare is processor specific.

The `MPMEMORY{}` command should be followed by `PROCESSOR processor_name{}` commands containing each processor's `MEMORY{}` and `SECTIONS{}` commands. The syntax for the `MPMEMORY{}` command appears in [Figure 2-5](#).

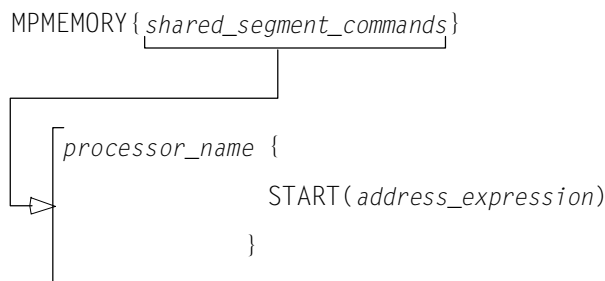


Figure 2-5. Syntax Tree of the `MPMEMORY{}` Command

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

- *shared_segment_commands*
Contains *processor_name* declarations with a `START{ }` address for each processor offset in multiprocessor memory. Processor names follow the same rules as any linker label. [For more information, see “LDF Expressions and Conventions” on page 2-40.](#)
- `PROCESSOR processor_name {placement_commands}`
Applies the *processor_name* offset for multiprocessor linking. [For more information, see “PROCESSOR{ }” on page 2-64.](#)

OVERLAY_GROUP{ }

Overlays are sets of program data or instructions that reside (“live”) off chip. When needed, they are brought on chip into “run-time memory”, under the control of an assembly language *overlay manager* routine.

Overlaying lets you improve performance by placing currently executing code in your fastest memory, though the entire program may be too large to fit in that memory. Performance is most enhanced when code executes in phases, with relatively long residence in certain portions of the program. An FFT is a good example of such code.

Overlays can be *grouped* or *ungrouped*. Ungrouped overlays execute sequentially from a single starting address in “live” memory. The `OVERLAY_GROUP` command lets you group overlays, so that the group is brought into run-time memory together, and its constituents run from different starting addresses. [Figure 2-6 on page 2-58](#) and [Figure 2-7 on page 2-58](#) show a comparison of ungrouped versus grouped overlays.

Linker Description File Reference

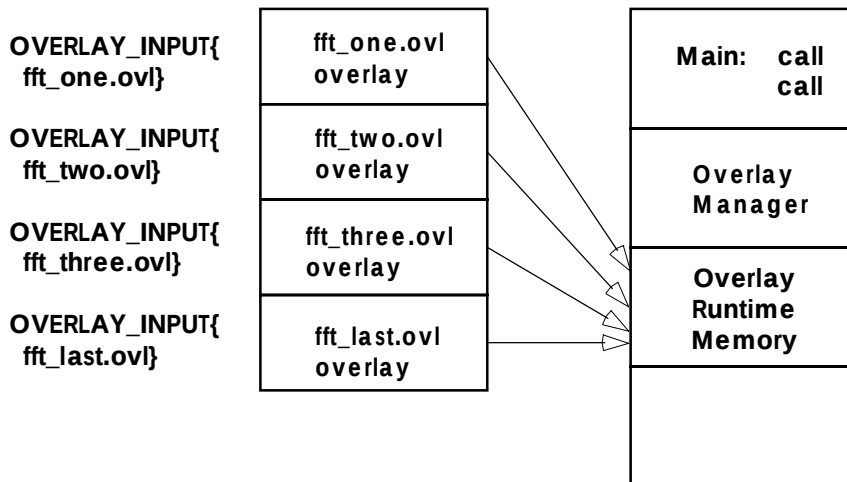


Figure 2-6. Overlays, Not Grouped

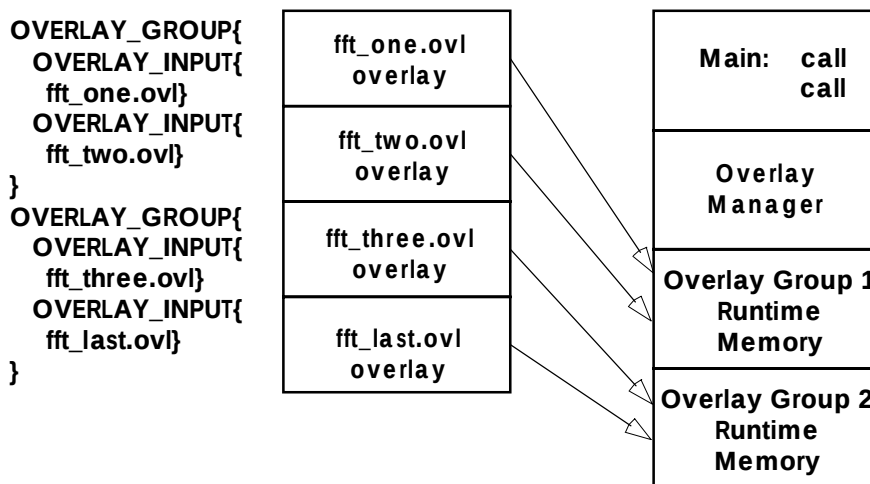


Figure 2-7. Overlays, Grouped

Listing 2-4. LDF Overlays, Not Grouped

```

/*
Declare which functions reside in which overlay. The overlays
have been split up into either different segments if in the
same file or different files. The overlays declared in this
section, Code, will run in segment M0Code.
*/

OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_1st.doj( program) )
} >M0_ovly // Overlay to live in section M0_ovly
OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_two.ovl)
    INPUT_SECTIONS( Fft_mid.doj( program ) )
} >M0_ovly // Overlay to live in section M0_ovly
OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS( Fft_last.doj(program1) )
} >M0_ovly // Overlay to live in section M0_ovly
OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_last.ovl)
    INPUT_SECTIONS( Fft_last.doj( program2 ) )
} >M0_ovly // Overlay to live in section M0_ovly

```

Listing 2-5. LDF Overlays, Grouped

```

/*
Declare which functions reside in which overlay. The overlays
have been split up into either different segments if in the
same file or different files. The overlays declared in this
section, Code, will run in segment M0Code.
*/

```

Linker Description File Reference

```
OVERLAY_GROUP {
    OVERLAY_INPUT {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_one.ovl)
        INPUT_SECTIONS( Fft_1st.doj( program ) )
    } >M0_ovly // Overlay to live in section M0_ovly
    OVERLAY_INPUT {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_two.ovl)
        INPUT_SECTIONS( Fft_mid.doj( program ) )
    } >M0_ovly // Overlay to live in section M0_ovly
}
OVERLAY_GROUP {
    OVERLAY_INPUT {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_three.ovl)
        INPUT_SECTIONS( Fft_last.doj( program1 ) )
    } >M0_ovly // Overlay to live in section M0_ovly
    OVERLAY_INPUT {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_last.ovl)
        INPUT_SECTIONS( Fft_last.doj( program2 ) )
    } >M0_ovly // Overlay to live in section M0_ovly
}
```

PACKING()

The `PACKING()` command specifies the order the linker uses to place bytes in a memory. This order correlates to the order that the DSP may use as it unpacks the data when the DSP transfers data from external memory into its internal memory. The processor's unpacking order can relate to the transfer method.



TigerSHARC DSPs do not require byte packing for 32- or 64-bit bus systems, and therefore, the `PACKING()` syntax does not apply to these DSPs.

PLIT{}

The linker's `PLIT{}` command lets you add Procedure Linkage Table (PLIT) commands to your LDF. These commands are assembly instructions, which are specific to an overlay, that the linker generates whenever a symbol resolves to a function in overlay memory. These instructions handle a call to a function in overlay memory by calling an overlay memory manager. A `PLIT{}` command may appear in the global LDF scope within a `PROCESSOR{}` command or within an `INPUT_SECTIONS{}` command.

What is a PLIT?

A PLIT is a template of instructions for loading an overlay. It may include saving registers or stacking context information.

The linker does not accept a PLIT without any arguments (`PLIT{}`). If you do not want the linker to redirect function calls in overlays, you should omit the PLIT commands entirely.

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.

PLIT Functions

In order to work correctly, a PLIT must enable the executable code to perform the following functions. These descriptions list the standard variable names for performing each function:

- Return the absolute address of the resolved symbol in “live” memory: `PLIT_SYMBOL_ADDRESS`.
- identify the overlay it must bring in and run when it encounters the symbol resolving an address in it: `PLIT_SYMBOL_OVERLAYID`.
- If necessary, return a null-terminated array of overlay IDs containing any data for the overlay function: `PLIT_DATA_OVERLAY_IDS`. Data can be overlaid, just like code.

Linker Description File Reference

- Save the target’s state on the stack or in memory before loading and executing an overlay function, so it continues correctly on return. **Note:** Your program may not need this information saved.
- Initiate (jump to) the routine that transfers the overlay code to internal memory, given the previous information about its identity, size and location: `_OverlayManager` “Smart” overlay managers first check whether the overlay function is already in the internal memory; otherwise, avoid reloading it.

Allocating Space for PLITs

Your LDF must allocate space in memory to hold any PLITs your linker builds. Typically, that memory resides in the program code (M0Code*) Memory section. A typical LDF declaration for that purpose appears below.

```
// ... In the SECTIONS command for Processor P0
// Plit code is to reside and run in PROGRAM section
.plit {} > M0Code
```

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

- There is no Input Section associated with the `.plit` Output Section: the LDF is allocating space for linker-generated routines, not containing any of your (input) data objects.
- This segment allocation does not take any parameters. You write the structure of this command (see [“PLIT Syntax” on page 2-63](#)). The linker creates instances 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 Memory Segment.

* Whatever you name your program code Memory Segment.

PLIT Syntax

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

The general syntax for the `PLIT{}` command appears in [Figure 2-8](#), which shows how the linker handles a symbol local to an overlay function.

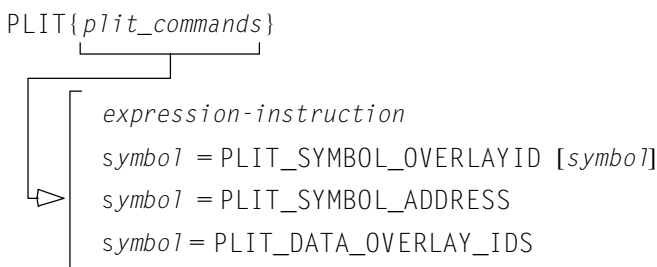


Figure 2-8. Syntax Tree of the `PLIT{}` Command

The linker first evaluates the *plit_commands*, a sequence of assembly code. Each line is passed to a processor-specific assembler, which supplies values for the symbols and expressions. After evaluation, the linker puts the returned bytes in the `.plit` Output Section. It also manages addressing in that Output Section.

- *expressions* are assembly instructions or expressions. There may be none, one, or more. They may occur in any reasonable order in the command structure, and may precede or follow the symbols discussed in the next few paragraphs. Within the `PLIT{}` command, you may use any valid assembly instruction.

The next two symbols contain information about *symbol* and the overlay where it occurs. In most cases, you must supply instructions to handle that information:

Linker Description File Reference

- The `PLIT_SYMBOL_OVERLAYID` command directs the linker to return the overlay ID of the resolved symbol.
symbol_1 is typically a register name or memory location, which is loaded with that overlay ID.
- The `PLIT_SYMBOL_ADDRESS` command directs the linker to return the absolute address of the resolved symbol in run-time memory.
symbol_2 is typically a register name or memory location, which is loaded with that address.

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

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

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

- `jump (_OverlayManager)` is normally the last instruction in the PLIT.

PROCESSOR{}

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

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

sors that you can declare is processor-specific. The syntax for the `PROCESSOR{}` command appears in [Figure 2-9](#).

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

Figure 2-9. Syntax Tree of the `PROCESSOR{}` Command

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

- *processor_name*
Assigns a *processor_name* to the processor. Processor names follow the same rules as any linker label. [For more information, see “LDF Expressions and Conventions” on page 2-40.](#)
- `OUTPUT(file_name.DXE)`
Selects the output file name for the executable (`.DXE`). Note that an `OUTPUT()` command in an LDF scope must appear before a `SECTIONS{}` command in that scope.
- `MEMORY{segment_commands}`
Defines memory segments that apply only to this processor. Using LDF command scoping, you can define these segments outside the `PROCESSOR{}` command. [For more information, see “Command Scoping” on page 2-39, and “MEMORY{” on page 2-54.](#)

Linker Description File Reference

- `PLIT{plit_commands}`

Defines Procedure Linkage Table (PLIT) commands that apply only to this processor. [For more information, see “PLIT{ }” on page 2-61.](#)

- `SECTIONS{section_commands}`

Defines sections for placement within the executable (.DxE). [For more information, see “SECTIONS{ }” on page 2-67.](#)

- `RESOLVE(symbol_name, resolver)`

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



If you resolve a C/C++ variable, prefix it with an underscore in the `RESOLVE()` statement (i.e., `_symbol_name`).

You can override the search order for a specific variable or label by using the `RESOLVE()` command, which directs the linker to ignore a `LINK_AGAINST()` for a specific symbol. [For more information, see Listing 2-9 on page 2-81.](#)

SEARCH_DIR()

The linker's `SEARCH_DIR()` command specifies one or more directories that the linker searches for input files. You may specify multiple directories within `SEARCH_DIR` commands, delimiting each path with a semicolon (;) and enclosing long directory names within straight quotes, "long directory name". The search order follows the order in which directories appear. This command appends search directories to the directory selected with the `-L` linker command-line switch.

Place this command at the beginning of the LDF, so the linker applies the command to all file searches.

SECTIONS{}

The linker's `SECTIONS{}` command specifies the placement of your program's `.SECTIONS` in memory, using segments defined with the linker's `MEMORY{}` command.

Your linker description file may contain a `SECTIONS{}` command within each `PROCESSOR{}` and `SHARED_MEMORY{}` command. The `SECTIONS{}` command must be preceded by a `MEMORY{}` command, defining the memory segments in which the linker places the sections. The syntax for the `SECTIONS{}` command appears in [Figure 2-10](#).

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

- *section_commands* or *expression*

Defines *expressions* or output sections (*section_name*). Use *expressions* to manipulate symbols or position the current location counter. Use output section commands to declare your program's sections. While your linker description file may contain only one `SECTIONS{}` command within each LDF scope, there is no limit to the number of input sections that you can declare within each `SECTIONS{}` command. [For more information, see “Command Scoping” on page 2-39.](#)

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

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

Linker Description File Reference

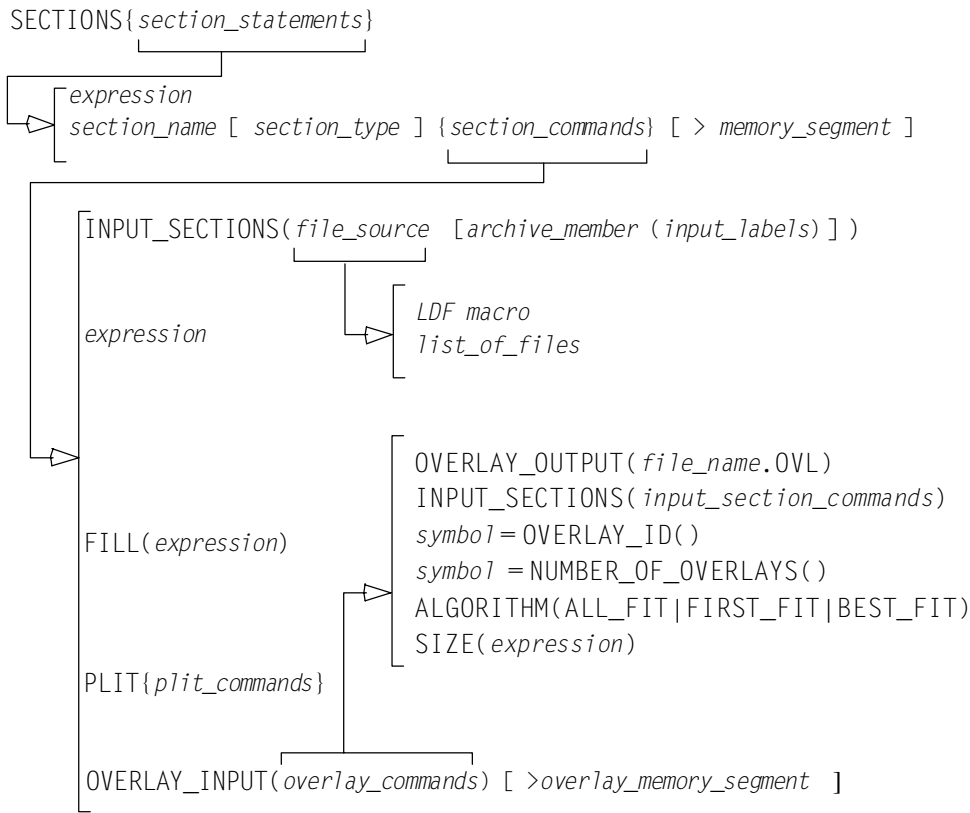


Figure 2-10. Syntax Tree of the `SECTIONS{}` Command

- `section_type` is optional and assigns an ELF section type. The only legal section type keyword is `SHT_NOBITS`. This section type is a section containing uninitialized data. For an example of how to use `SHT_NOBITS`, see [Listing 2-8](#).

- *section_commands* may contain any combination of the following commands: an `INPUT_SECTIONS()` command, an *expression*, a `FILL()` command, a `PLIT{}` command, or an `OVERLAY_INPUT()` command.
- `> memory_segment` at the end of a section definition declares whether the section is placed in the specified memory segment. It is optional. Some sections, such as those for debugging, do not need to be included in the memory image of the executable, but are needed for other development tools that read the executable file. By omitting a memory segment assignment for a section, you direct the linker to keep the section in the executable, but mark the section for exclusion from the memory image of the executable.
- `INPUT_SECTIONS()`

This part of the syntax in an *output_section_command* identifies the parts of your program to place in the executable with *input_section_commands*. When placing an input `.SECTION`, specify the *file_source*, *archive_member* (if the *file_source* is an archive), and *input_labels* of the sections. An `INPUT_SECTIONS()` command has the following syntax rules:

- *file_source* may be a list of files or any LDF macro that expands into a file list, such as the `$COMMAND_LINE_OBJECTS` macro. The list may contain object or archive files. Use commas to delimit files within the list and enclose long file names within straight-quotes, "long file name".
- *archive_member* names the source-object file within an archive. The *archive_member* parameter and the left/right brackets, `[ ]`, are only required if the *file_source* of the *input_label* is an archive.

Linker Description File Reference

- *input_labels* come from the run-time `.SECTION` labels in your assembly program. Use commas to delimit `.SECTION` names within the list. The linker places `.SECTIONS` in the order that you list them.

- *expression*

In a *section_command*, manipulates symbols or positions the current location counter specified by a period. It is an assembly directive.

- `FILL(expression)`

In a *section_command*, fills with the *expression* any gaps that you create by aligning or advancing the current location counter. By default, the linker fills these gaps with the *expression* “0”. Specify only one `FILL()` command per output section.

- `PLIT{plit_commands}`

In a *section_command*, declares a locally* scoped Procedure Linkage Table (PLIT). It contain its own labels and expressions. [For more information, see “PLIT{ }” on page 2-61.](#)

- `OVERLAY_INPUT(overlay_commands)`

In a *section_command*, it identifies parts of your program to place in an overlay executable with *overlay_commands*. [For more information, see “Linking for Overlay Memory” on page 2-86.](#) The *overlay_commands* part of the syntax must contain at least one of the following commands: an `INPUT_SECTIONS()` command, and `OVERLAY_ID()` command, a `NUMBER_OF_OVERLAYS()` command, a `OVERLAY_OUTPUT()` command, an `ALGORITHM()` command, a `RESOLVE_LOCALLY()` command, or `SIZE()` command. Note that you can increment the current location counter only within the

* In that section only.

`OVERLAY_INPUT()` command; this is the only context where it is illegal to decrement the counter.

The *memory_segment_name* determines whether the section is placed in an overlay segment and is optional. Some overlay sections, such as those loaded from a host, do not need to be included in the overlay memory image of the executable, but are needed for other development tools that read the executable file. By omitting an overlay memory segment assignment for a section, you direct the linker to keep the section in the executable, but mark the section for exclusion from the overlay-memory image of the executable.

An `OVERLAY_INPUT()` command has the following syntax rules:

- The `OVERLAY_OUTPUT()` command directs the linker to output an overlay file (`.OVL`) for the overlay with the name specified. Note that an `OVERLAY_OUTPUT()` command in an `OVERLAY_INPUT()` command must appear before any `INPUT_SECTIONS()` for that overlay.
- The `INPUT_SECTIONS()` command has the same syntax within an `OVERLAY_INPUT()` command as when it appears within a *output_section_command*, except that you can not place the `.plt` section in the overlay memory. [For more information, see “INPUT_SECTIONS\(\)” on page 2-69.](#)
- The `OVERLAY_ID()` command directs the linker to return the overlay ID of the resolved symbol.
- The `NUMBER_OF_OVERLAYS()` command directs the linker to return the number of overlays that the current link generates when using the `FIRST_FIT` or `BEST_FIT` overlay-placement `ALGORITHM()`.
- The `ALGORITHM()` command directs the linker to use the specified overlay linking algorithm. Valid linking algorithms include `ALL_FIT`, `FIRST_FIT`, and `BEST_FIT`.

Linker Description File Reference

- 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.
- For `FIRST_FIT`, the linker splits the input sections mentioned in the `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.
- For `BEST_FIT`, the linker splits the input sections mentioned in the `OVERLAY_INPUT()` into a set of overlays that can each overlay the *output_section*'s runtime memory segment, but splits these overlays to optimize memory usage.

Note that when using the `FIRST_FIT` or `BEST_FIT` algorithm the overlay ID of the next overlay equals:

`Next ID = OVERLAY_ID() + NUMBER_OF_OVERLAYS()`

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

SHARED_MEMORY{ }

The linker produces two types of executable output: `.DXEs`, which run in a single processor's address space, and Shared Memory executable (`.SM`) files that reside in the shared memory of a system. These are produced by the `SHARED_MEMORY{ }` command.

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

Your LDF may contain any number of `SHARED_MEMORY{ }` commands, but the number of processors that can access a shared memory is processor-specific.* The `SHARED_MEMORY{ }` command must appear in the same LDF scope of a `MEMORY{ }` command that describes the shared memory. The `PROCESSOR{ }` commands that declare the processors that share this memory must also appear within this same LDF scope.

The syntax for the `SHARED_MEMORY{ }` command appears in [Figure 2-11](#), followed by definitions of its components.

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

Figure 2-11. Syntax Tree of the `SHARED_MEMORY{ }` Command

* For ADSP-TS001 TigerSHARC shared memory systems, you may have up to 6 processors accessing the shared memory without adding external logic for bus arbitration. This limit may be exceeded if you add bus arbitration logic to the system.

Linker Description File Reference

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

- `OUTPUT(file_name.SM)`

Selects the output file name for the Shared Memory executable (`.SM`). An `OUTPUT()` command in a `SHARED_MEMORY{}` command must appear before the `SECTIONS{}` command in that scope.

- `SECTIONS{section_commands}`

Defines sections for placement within the Shared Memory executable (`.SM`). [For more information, see “SECTIONS{ }” on page 2-67.](#)

An example of this command scoping appears in [Figure 2-12](#). [For more information, see “Command Scoping” on page 2-39.](#)

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

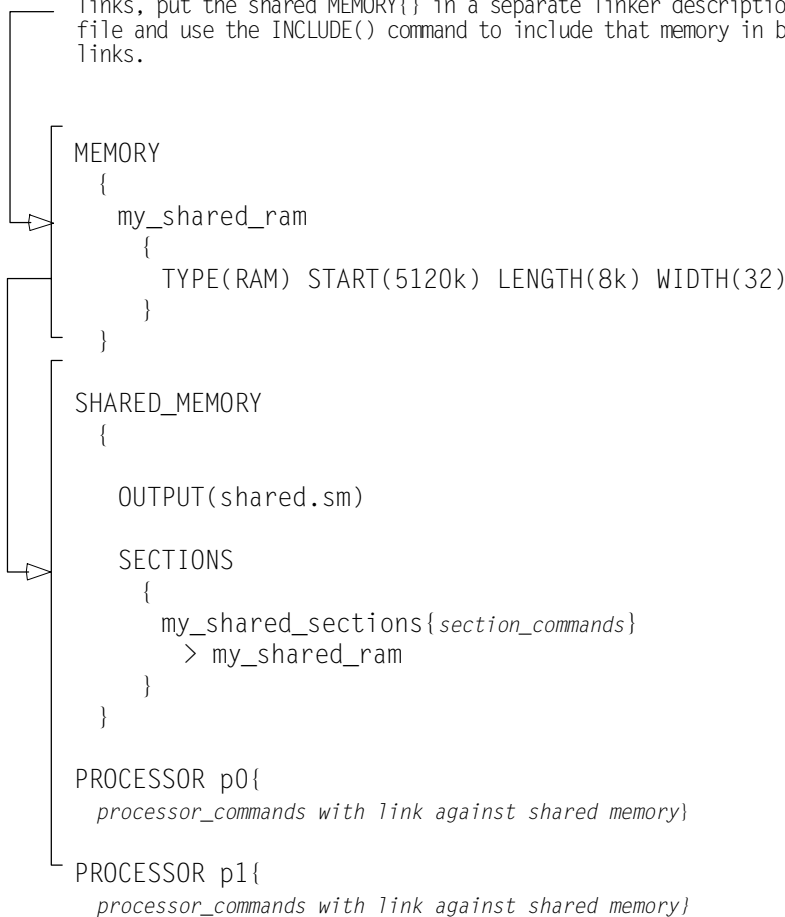


Figure 2-12. LDF Scoping for the SHARED_MEMORY{} Command

LDF Programming Examples

This section shows LDF examples for many typical system models. As you modify these examples, refer to the syntax descriptions in “[LDF Commands](#)” on page 2-51. The examples in this section include the following:

- “[Linking for Single-Processor Memory](#)” on page 2-77
- “[Linking Large Uninitialized Variables](#)” on page 2-79
- “[Linking for Multi-Processor and Shared Memory](#)” on page 2-80
- “[Linking for Overlay Memory](#)” on page 2-86
- “[Using a Procedure Linkage Table](#)” on page 2-90
- “[Managing Overlays](#)” on page 2-97
- “[Managing Two Overlays](#)” on page 2-102
- “[Reducing Overlay Manager Overhead](#)” on page 2-107



The source code for a variety of example programs comes with your development software. Each example program includes an LDF file. For working examples of the linking process, examine the LDF files that come with the examples. For the TigerSHARC DSPs, these examples are in the directory:

```
InstallPath\Analog Devices\VisualDSP++\Ts\examples
```



A variety of per processor default LDF files come with the development software, providing an example LDF for each processor’s internal memory architecture. For the TigerSHARC DSPs, these default LDFs are in the directory:

```
InstallPath\Analog Devices\VisualDSP++\Ts\ldf
```

Linking for Single-Processor Memory

When linking an executable file for a single-processor system, the LDF describes the processor's memory and places code for that processor. The example LDF in [Listing 2-6](#) shows a single processor LDF. Note the following items in this LDF:

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

Listing 2-6. Single-Processor System LDF Example

```
// Link for the ADSP-TS001

ARCHITECTURE(ADSP-TS001)

// Generate a MAP file

MAP(SINGLE-PROCESSOR.MAP)

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

SEARCH_DIR( $ADI_DSP\Ts\lib )

$LIBS = libc.dlb;
```

LDF Programming Examples

```
// 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
// object(s) (.doj files) and archives (.dlb files) that
// appear on the command line. In this example,
// $COMMAND_LINE_OBJECTS = single.doj

// ts_hdr.doj is the standard initialization file for TSxxx

$OBS = $COMMAND_LINE_OBJECTS, ts_hdr.doj;

// A linker project to generate a DXE file.

PROCESSOR P0 {
    // The name of the output file is specified with the
    // OUTPUT command
    OUTPUT( .\SINGLE.DXE )
    // Processor specific memory command
    MEMORY {
        INCLUDE( "TS001_memory.ldf" )
        // For content, see Listing 2-2 on page 2-19.
    }
    // Specify the output sections
    SECTIONS {
        INCLUDE( "TS001_sections.ldf" )
        // For content, see Listing 2-3 on page 2-21.
    }
    // end P0 sections
} // end P0 processor
```

Linking Large Uninitialized Variables

When linking an executable file that contains large uninitialized variables, you can reduce the size of the file using the `SHT_NOBITS` section type syntax. This is a useful technique, especially for ADSP-TSxxx users with large SDRAM systems.

A variable defined in a source file normally takes up space in an object and executable file even if that variable is not explicitly initialized when defined. For large buffers this can result in large executables filled mostly with zeros. Such files take up excess disk space and can incur large download times when using the emulator.

The LDF can specify that an output section is omitted from the output file. The LDF output section type `SHT_NOBITS` directs the linker to omit data for that section from the output file. [Listing 2-8](#) shows an example using the `SHT_NOBITS` section to avoid initialization of a segment.



The `SHT_NOBITS` technique corresponds to using the `/UNINIT` segment qualifier in previous (`.ACH`) development tools.

Even if you do not use the `SHT_NOBITS` technique, the boot loader removes variables initialized to zeros from the `.ldr` file and replaces them with instructions for the loader kernel to zero out the variable. This reduces the file size, but still requires execution time for the processor to initialize the memory with zeros.

Listing 2-7. Using Large Uninitialized Variables: Assembly Source

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

LDF Programming Examples

Listing 2-8. Using Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-TS001)

// Libraries & objects from the command line

$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY{
    SDRAM{TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)}
} // end memory
PROCESSOR P0 {
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    // SHT_NOBITS section isn't written to the output file
    SECTION {
        sdram_output SHT_NOBITS {
            INPUT_SECTIONS( $OBJECTS ( sdram_area ) )
        } >SDRAM
    } //end section
} // end processor P0
```

Linking for Multi-Processor and Shared Memory

When linking executable files for a multiprocessor memory and shared memory system, the LDF describes the multiprocessor memory offsets, shared memory, each processor's memory; and places code for each processor. The example LDF in [Listing 2-9](#) shows a multiprocessor memory and shared memory LDF. Note the following items in this LDF:

- `ARCHITECTURE()` command defines the processor type (note that only one processor type can be defined within an LDF)
- `SEARCH_DIR()` commands add the `lib` and current working directory to the search path
- `$OBJ`s and `$LIB`s macros get object (`.DOJ`) and library (`.DLB`) file input

- `MPMEMORY{ }` command defines each processor's offset within multi-processor memory
- `SHARED_MEMORY{ }` command identifies the output for the shared memory items
- `MAP( )` command outputs map files
- `MEMORY{ }` command defines memory for the processors
- `PROCESSOR{ }` and `SECTIONS{ }` commands define each processor and places program sections for each processor's output file, using the memory definitions
- `LINK_AGAINST( )` commands resolve symbols within multiprocessor memory

Listing 2-9. Multi-Processor System LDF Example

```

ARCHITECTURE(ADSP-TS001)
SEARCH_DIR( $ADI_DSP\Ts\lib )
//      Multiprocessor memory space is represented in the
//      LDF via the MPMEMORY command. The values represent
//      an "addend" that the linker will use when
//      it resolves undefined symbols in one DXE to symbols
//      defined in another DXE. That is, the addend is added
//      to the defined symbols value.
//
// For example,
//      PROCESSOR project PSH0 contains the undefined symbol
//      "buffer"
//      PROCESSOR project PSH1 defined the symbol "buffer" at
//      address 0x22000
//
//      the linker will "fix up" the reference to "buffer" in
//      PSH0's code to address
//      0x22000 + MPMEMORY(PSH1) =
//      0x22000 + 0x2400000      =
//      0x2422000

```

LDF Programming Examples

```
MPMEMORY {
    PSH0 { START (0x2000000) }
    PSH1 { START (0x2400000) }
}
MEMORY {
    //This memory description is to be used for all
    //processors
    //Alternatively, a PROCESSOR could describe its
    //own MEMORY
    INCLUDE( "TS001_memory.ldf" )
    // For content, see Listing 2-2 on page 2-19.
}

$LIBRARIES = libc.dlb;

//There will be three link projects specified in this
//one LDF file.
//The first link project is a shared memory link project
//against which the PROCESSOR projects will be linked
//

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

$SHARED_OBJECTS = shared.doj;

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

OUTPUT(shared.sm)

//shared.c has only data declarations. No need to
//specify any output section other than "data2"

SECTIONS {
    data2{
        INPUT_SECTIONS( $SHARED_OBJECTS(data2))
    } >M1Data
    } // end shared sections
} // end shared memory
```

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

PROCESSOR PSH0 {
    $PSH0_OBJECTS = psh0.doj, ts_hdr.doj;
    LINK_AGAINST(shared.sm)
    OUTPUT( psh0.dxe )
    SECTIONS {
        // places code (instructions) in M0 (internal memory)
        code{
            FILL(0xb3c00000)
            INPUT_SECTION_ALIGN(4)
            INPUT_SECTIONS(
                $PSH0_OBJECTS(program) $LIBRARIES(program)
            )
        } >M0Code

        // places data in M2 (internal memory)
        data1{
            INPUT_SECTIONS(
                $PSH0_OBJECTS(data1) $LIBRARIES(data1)
            )
        } >M2Data

        // places data in M1 (internal memory)
        data2{
            INPUT_SECTIONS(
                $PSH0_OBJECTS(data2) $LIBRARIES(data2)
            )
        } >M1Data

        // places c rtl initializers in M2 (internal memory)
        ctor{
            INPUT_SECTIONS( $LIBRARIES(ctor0))
            INPUT_SECTIONS( $LIBRARIES(ctor1))
            INPUT_SECTIONS( $LIBRARIES(ctor2))
            INPUT_SECTIONS( $LIBRARIES(ctor3))
            INPUT_SECTIONS( $LIBRARIES(ctor))
        } >M2Data
    }
}
```

LDF Programming Examples

```
// places c rte heap table in M2 (internal memory)
heaptab{
    INPUT_SECTIONS(
        $LIBRARIES(heaptab) $PSH0_OBJECTS(heaptab)
    )
} >M2Data

// places c rte J IALU stack in M2 (internal memory)
jstackseg{
    ldf_jstack_limit = .;
    ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
} >M2Stack

// places c rte K IALU stack in M1 (internal memory)
kstackseg{
    ldf_kstack_limit = .;
    ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
} >M1Stack

// places c rte heap in M2 (internal memory)
defheapseg{
    ldf_defheap_base = .;
    ldf_defheap_size = MEMORY_SIZEOF(M2Heap);
} >M2Heap
} // end PSH0 sections
} // end PSH0 processor
//
//The third and final link project defined in this LDF
//file is another DXE project. This PROCESSOR project will
//be linked against both the SHARED project defined above
//and the PSH0 DXE project also defined above.
//

PROCESSOR PSH1 {
    $PSH1_OBJECTS = psh1.doj, ts_hdr.doj;
    LINK_AGAINST(shared.sm, psh0.dxe)
    OUTPUT(psh1.dxe)
```

```

SECTIONS {
// places code (instructions) in M0 (internal memory)
code{
    FILL(0xb3c00000)
    INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS(
        $PSH1_OBJECTS(program) $LIBRARIES(program)
    )
} >M0Code

// places data in M2 (internal memory)
data1{
    INPUT_SECTIONS(
        $PSH1_OBJECTS(data1) $LIBRARIES(data1)
    )
} >M2Data

// places data in M1 (internal memory)
data2{
    INPUT_SECTIONS(
        $PSH1_OBJECTS(data2) $LIBRARIES(data2)
    )
} >M1Data

// places c rtl initializers in M2 (internal memory)
ctor{
    INPUT_SECTIONS( $LIBRARIES(ctor0))
    INPUT_SECTIONS( $LIBRARIES(ctor1))
    INPUT_SECTIONS( $LIBRARIES(ctor2))
    INPUT_SECTIONS( $LIBRARIES(ctor3))
    INPUT_SECTIONS( $LIBRARIES(ctor))
} >M2Data

// places c rte heap table in M2 (internal memory)
heaptab{
    INPUT_SECTIONS(
        $LIBRARIES(heaptab) $PSH1_OBJECTS(heaptab)
    )
} >M2Data

```

LDF Programming Examples

```
// places c rte J IALU stack in M2 (internal memory)
jstackseg{
    ldf_jstack_limit = .;
    ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
} >M2Stack

// places c rte K IALU stack in M1 (internal memory)
kstackseg{
    ldf_kstack_limit = .;
    ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
} >M1Stack

// places c rte heap in M2 (internal memory)
defheapseg{
    ldf_defheap_base = .;
    ldf_defheap_size = MEMORY_SIZEOF(M2Heap);
} >M2Heap
} // end PSH1 sections
} // end PSH1 processor
```

Linking for Overlay Memory

When linking executable files for an overlay memory system, the LDF describes the overlay memory, the processors that use the overlay memory, and each processor's unique memory. The LDF places code for each processor and the special `.plit` section. The example LDF in [Listing 2-10](#) shows an overlay memory LDF. For more information on this LDF, see the comments in the listing.

Listing 2-10. Overlay-Memory System LDF Example

```
ARCHITECTURE(ADSP-TS001)
SEARCH_DIR( $ADI_DSP\Ts\lib )

MAP(overlay.map)

/*
This simple overlay example uses internal memory as overlay
storage memory. Typically, overlays would never be stored in
internal memory; they would just be loaded there at runtime.
```

```

*/

// Internal memory blocks are 0x10000 (64k)

MEMORY {
M0Code {TYPE(RAM) START(0x00000000) END(0x00007FFF) WIDTH(32)}
M0_ovly{TYPE(RAM) START(0x00008000) END(0x0000FFFF) WIDTH(32)}
M1Data {TYPE(RAM) START(0x00080000) END(0x0008BFFF) WIDTH(32)}
M1Stack{TYPE(RAM) START(0x0008C000) END(0x0008FFFF) WIDTH(32)}
M2Data {TYPE(RAM) START(0x00100000) END(0x0010BFFF) WIDTH(32)}
M2Heap {TYPE(RAM) START(0x0010C000) END(0x0010C7FF) WIDTH(32)}
M2Stack{TYPE(RAM) START(0x0010C800) END(0x0010FFFF) WIDTH(32)}
SDRAM {TYPE(RAM) START(0x04000000) END(0x07FFFFFF) WIDTH(32)}
MS0 {TYPE(RAM) START(0x08000000) END(0x0BFFFFFF) WIDTH(32)}
MS1 {TYPE(RAM) START(0x0C000000) END(0x0FFFFFFF) WIDTH(32)}
HOST {TYPE(RAM) START(0x10000000) END(0xFFFFFFFF) WIDTH(32)}
}
// end memory; the M0_ovly segment is for overlay storage
/*
Processor and application specific assembly language
instructions. These instructions are generated for each symbol
that is resolved in overlay memory.
*/

PLIT {
    j4 = PLIT_SYMBOL_OVERLAYID;
    // Assign the overlay ID of the resolved symbol to j4

    j5 = PLIT_SYMBOL_ADDRESS;
    // Assign the "run" address of the resolved symbol to j5

    dm(_overlayID) = j4;
    dm(_pf) = j5;
    JUMP _OverlayManager;
}

$LIBRARIES = libc.dlb;
$OBJECTS = ts_hdr.doj;

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

    OUTPUT(mgrovly.dxe)
}

```

LDF Programming Examples

```
SECTIONS {
    // .text output section
    Code {
        FILL(0xb3c00000)
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS(
            $PO_OBJECTS(program) $LIBRARIES(program) )
    }
    /*
    Specify the first overlay. This overlay is stored in the memory
    defined by "MO_ovly". It runs in the memory space defined by
    "M0Code".
    */
    OVERLAY_INPUT {
    /*
    The output archive file "overlay1.ovl" contains the code and
    symbol table for this overlay.
    */
        OVERLAY_OUTPUT( overlay1.ovl )
    /*
    Only take the code from the file overlay1.doj. If there is data
    that this code needs it must be either the INPUT of a data
    overlay or the INPUT to non-overlay data memory.
    */
        INPUT_SECTIONS( overlay1.doj ( program) )
    /*
    Tell the linker that all of the code must fit into the "run"
    memory all at once. ALGORITHM(FIRST_FIT) would allow the linker
    to break the code into several overlays as necessary (in the
    event that not all of the code fit).
    */
        ALGORITHM( ALL_FIT )
        SIZE(0x100)
    } > MO_ovly
}
```

```

/*
This is the second overlay. Note that these OVERLAY_INPUT
command must be contiguous in the LDF file in order for them
to occupy the same runtime" memory.
*/
    OVERLAY_INPUT {
        OVERLAY_OUTPUT( overlay2.ovl )
        INPUT_SECTIONS( overlay2.doj (program) )
        ALGORITHM( ALL_FIT )
        SIZE( 0x100)
    } > MO_ovly
} > M0Code

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

        // linker inserts instructions here

    } > M0Code
data1 {
    INPUT_SECTIONS(
        $PO_OBJECTS(data1) $LIBRARIES(data1) )
    } > M2Data
data2 {
    INPUT_SECTIONS(
        $PO_OBJECTS(data2) $LIBRARIES(data2) )
    } > M1Data

// places c rtl initializers in M0 (internal memory)
ctor{
    INPUT_SECTIONS( $LIBRARIES(ctor0))
    INPUT_SECTIONS( $LIBRARIES(ctor1))
    INPUT_SECTIONS( $LIBRARIES(ctor2))
    INPUT_SECTIONS( $LIBRARIES(ctor3))
    INPUT_SECTIONS( $LIBRARIES(ctor))
    } >M2Data

```

LDF Programming Examples

```
// places c rte heap table in M0 (internal memory)
heaptab{
    INPUT_SECTIONS(
        $LIBRARIES(heaptab) $PSH0_OBJECTS(heaptab)
    )
} >M2Data

// places c rte J IALU stack in M0 (internal memory)
jstackseg{
    ldf_jstack_limit = .;
    ldf_jstack_base = . + MEMORY_SIZEOF(M2Stack);
} >M2Stack

// places c rte K IALU stack in M1 (internal memory)
kstackseg{
    ldf_kstack_limit = .;
    ldf_kstack_base = . + MEMORY_SIZEOF(M1Stack);
} >M1Stack

// places c rte heap in M0 (internal memory)
defheapseg{
    ldf_defheap_base = .;
    ldf_defheap_size = MEMORY_SIZEOF(M2Heap);
} >M2Heap
} // end P0 sections
} // end P0 processor
```

Using a Procedure Linkage Table

The linker resolves function calls, both direct and indirect, across overlays. This support implies that the linker must generate some extra code in order to transfer control to a user defined routine (an overlay manager) that handles the loading of overlays. The linker overlay manager code goes in a special section of the executable, which has the section name `.plit`. For more details on placing `.plit` code, see the description of [“PLIT{ }” on page 2-61](#).

The `PLIT{ }` command lets you insert assembly instructions that handle calls to functions in overlays. The assembly commands are specific to an

overlay and are executed each time a call to a function in that overlay is detected.

[Figure 2-13 on page 2-92](#) shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker generates some special symbols for overlays. These overlay symbols include: `_ov_startaddress_#`, `_ov_endaddress_#`, `_ov_size_#`, and `_ov_runtimestartaddress_#`. The `#` in these symbols indicates the overlay number.

In this figure, the two functions are on different overlays. By default, the linker generates PLIT code only when an unresolved function reference is resolved to a function definition in overlay memory.

The main function calls functions `X()` and `Y()`, which are defined in overlay memory. Because the linker cannot resolve these functions locally, the linker replaces the symbols `X` and `Y` with `.plit_X` and `.plit_Y`. Any unresolved references to `X` and `Y` are resolved to `.plit_X` and `.plit_Y`. In cases where both the reference and the definition reside in the same executable, the linker does not generate PLIT code. You can force the linker to output a PLIT, however, even when all references can be resolved locally.

Using PLITs lets you resolve inter-overlay calls, as shown in [Figure 2-14 on page 2-93](#). You should structure your LDF so the PLIT code that the linker generates for inter-overlay function references is part of the `.plit` section for `main()`, which is stored in non-overlay memory.



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

A PLIT also lets you resolve inter-processor-overlay calls as shown in [Figure 2-15 on page 2-94](#). When one processor makes a call into another processor's overlay, the call increases the size of the `.plit` section in the executable that manages the overlay.

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

LDF Programming Examples

Non-Overlay Memory

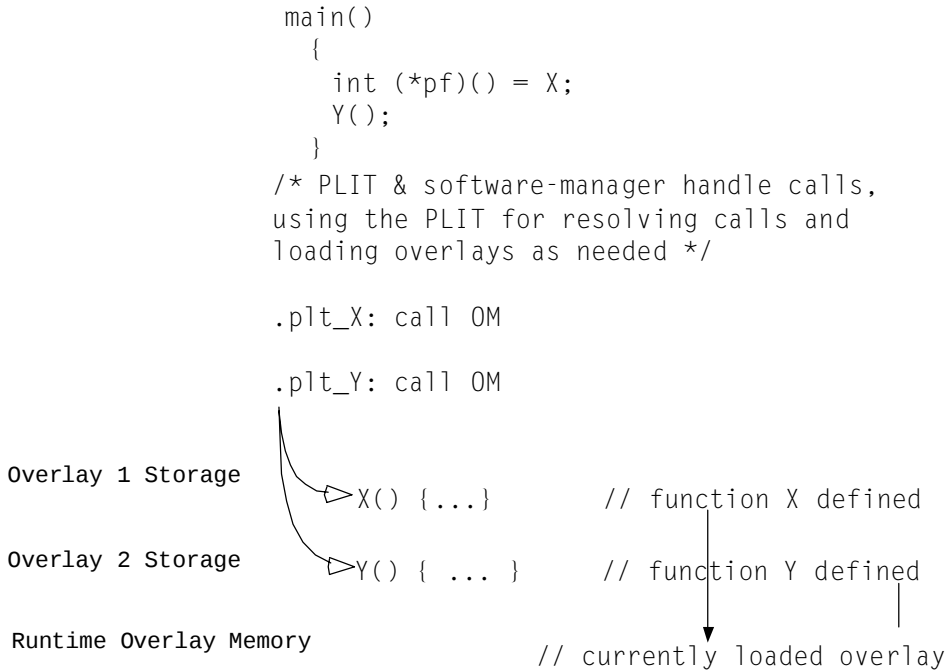


Figure 2-13. PLITs & Overlay Memory; main() Calls to Overlays

lays. One way to optimize overlays is to not put global variables in overlays. This avoids the difficulty of making sure the proper overlay is loaded before a global gets called.

Non-Overlay Memory

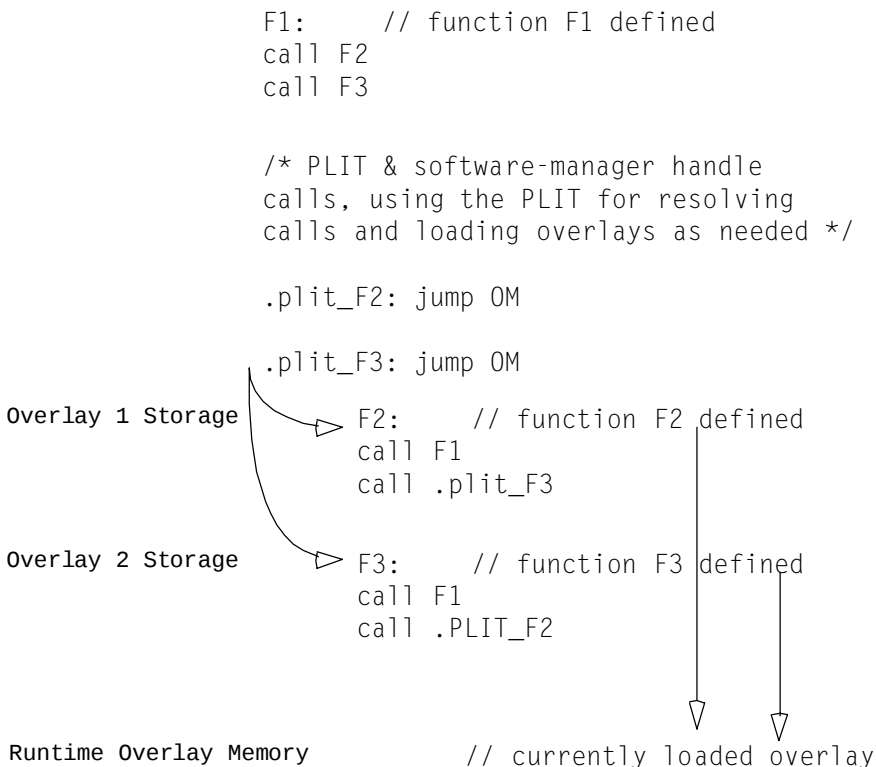


Figure 2-14. PLITs & Overlay Memory; Inter-Overlay Calls

LDF Programming Examples

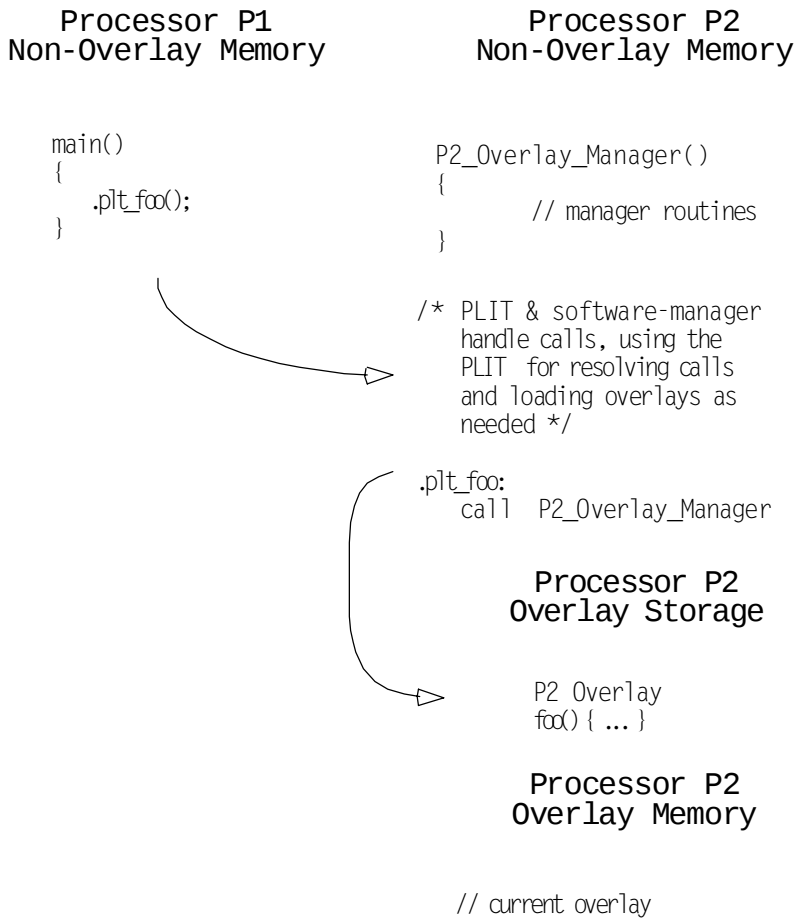


Figure 2-15. PLITs & Overlay Memory; Inter-Processor Calls

Using An Overlay Memory Manager

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

This appendix discusses the concept of memory overlays and how they are used with Analog Devices' 32-bit TigerSHARC DSPs. The following topics and examples are discussed:

- [“Managing Overlays” on page 2-97](#)
- [“Managing Two Overlays” on page 2-102](#)
- [“Reducing Overlay Manager Overhead” on page 2-107](#)

All of the code segments used in the following discussion are parts of the two example programs that appear at the end of this appendix.

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

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

[Figure 2-16](#) demonstrates the concept of memory overlays. There are two memory spaces: internal and external. The external memory is partitioned into five overlays. The internal memory contains the main program, an

LDF Programming Examples

overlay manager function and two segments reserved for execution of overlay program instructions.

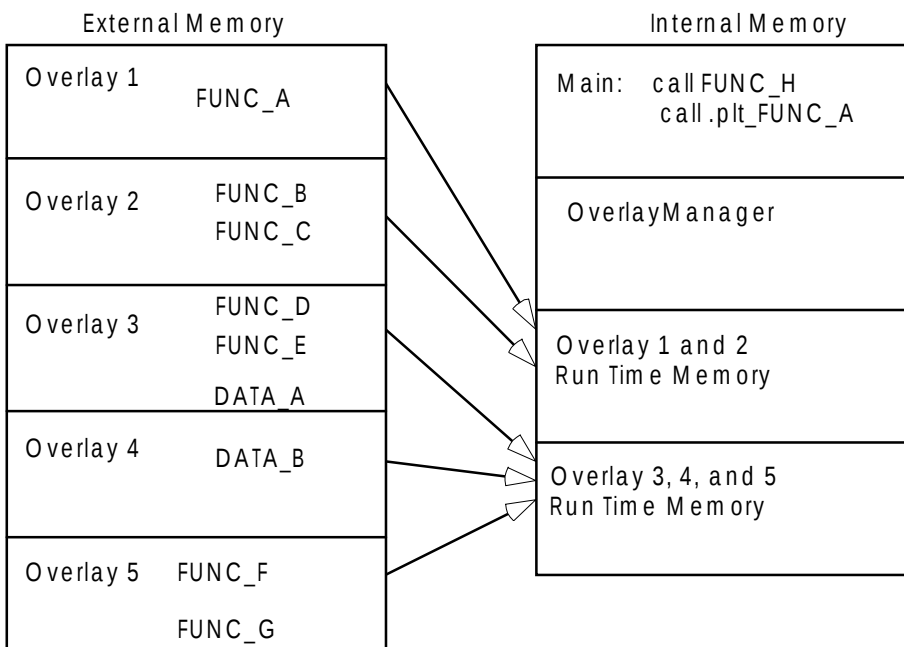


Figure 2-16. Memory Overlays

In this example, Overlay 1 and 2 share the same run time location within internal memory. Overlays 3, 4 and 5 also share a common run-time memory. When `FUNC_B` is required, the overlay manager loads Overlay 2 in the location within internal memory where the overlay is designated to run. When `FUNC_D` is required, the overlay manager loads Overlay 3 into its designated run-time memory. The overlay manager is a user-defined function responsible for insuring that a required symbol (function or data) within an overlay is in the run-time memory when it is needed.

The transfer occurs via DMA. The overlay manager can also handle more advanced functionality, such as checking , if the requested overlay is already in the run-time memory, executing another function while loading an overlay, and tracking recursive overlay function calls.

Managing Overlays

The overlay support provided by the 32-bit tools includes the following:

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

The overlay support is partially designed by the user. The user specifies which overlays share run time memory and which memory segments establish the live and run space. [Listing 2-11](#) shows the section of an LDF defining two overlays.

Listing 2-11. Overlay Declaration in LDF

```
Code{
    FILL(0xb3c00000)
    INPUT_SECTION_ALIGN(4)
    OVERLAY_INPUT{
        OVERLAY_OUTPUT( OVLY_one.ovl )
        INPUT_SECTIONS( FUNC_A.doj (program) )
    }>MO_ovly
    OVERLAY_INPUT{
        OVERLAY_OUTPUT( OVLY_two.ovl )
        INPUT_SECTIONS(
            FUNC_B.doj( program )
            FUNC_C.doj( program )
        )
    }>MO_ovly
}>MOCode
```

LDF Programming Examples

The overlay declaration in [Listing 2-11](#) configures two overlays to share a common run-time memory space:

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

The common run-time location shared by overlays `OVLY_one` and `OVLY_two` is within the memory segment `M0Code`.

The LDF provides the linker with direction on how to configure the overlays as well as the information necessary for the overlay manager routine to load the overlays. The information provided by the linker includes the following constants, where `N` = the Overlay ID:

Listing 2-12. Linker-Generated Overlay Constants

```
N = the Overlay ID
_ov_startaddress_N
_ov_endaddress_N
_ov_size_N
_ov_word_run_size_N
_ov_word_live_size_N
_ov_runtimestartaddress_N
```

Each overlay has a word size and an address which the overlay manager uses to determine where the overlay resides and where it is executed. Exception is the `_ov_size_N` that specifies the total size in bytes.

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



TigerSHARC DSPs do not require byte packing for 32- or 64-bit bus systems.

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

You can define PLIT code in the LDF. This code prepares the overlay manager to handle the overlay containing the referenced symbol. The code generally initializes registers to contain the overlay ID and the referenced symbol's run-time address.

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

```
j4 = [j0 += j3];;
j5 = j4 * j6;;
CALL FUNC_A;;          /* Call to function in overlay */
[j3 += j3] = j5;;
```

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

```
j4 = [j0 += j3];;
j5 = j4 * j6;;
CALL .plt_FUNC_A;;    /* Call to PLIT entry */
[j3 += j3] = j5;;
```

The `.plt_FUNC_A` is the entry in the PLIT containing your defined instructions. These instructions prepare the VisualDSP++ environment for the overlay manager to load the overlay containing `FUNC_A`. The instructions executed in the PLIT are specified within the LDF.

[Listing 2-13](#) is an example PLIT definition from an LDF file. In the example the register `j4` is set to the value of the overlay ID that contains the referenced symbol, and register `j5` is set to the run time address of the referenced symbol. (`PLIT_SYMBOL_OVERLAY_ID` and `PLIT_SYMBOL_ADDRESS` are

LDF Programming Examples

linker key words). The last instruction branches to the overlay manager. The overlay manager uses the initialized registers to determine which overlay to load, and where to jump to execute the overlay function called.

Listing 2-13. PLIT Definition in LDF

```
PLIT{  
    j4 = PLIT_SYMBOL_OVERLAY_ID;  
    j5 = 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 [Figure 2-17](#).

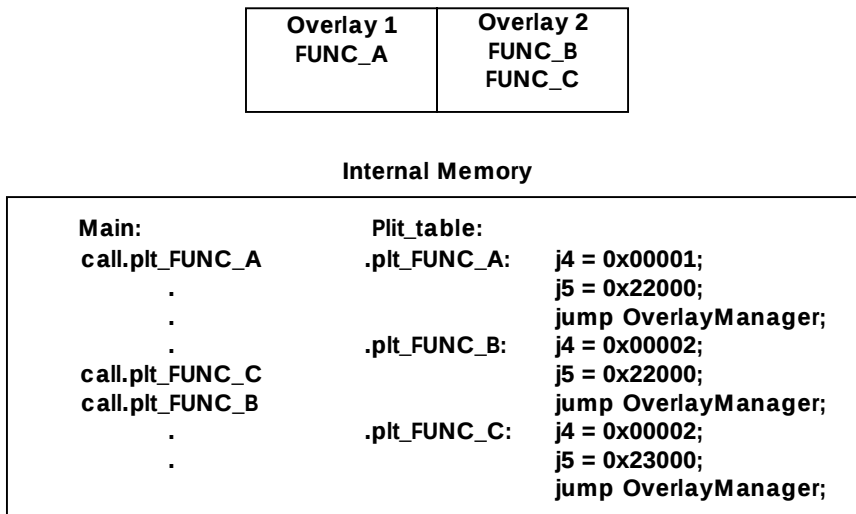


Figure 2-17. Expanded PLITTable

The redirect function calls the PLIT table for overlays 1 and 2 of the example. For each entry the linker replaces the generic assembly instructions with specific instructions (where applicable). For example, the first entry in the PLIT shown in [Figure 2-17](#) is for the overlay symbol `FUNC_A`. The linker replaces the constant name `PLIT_SYMBOL_OVERLAYID` with the ID of the overlay containing `FUNC_A`. The linker also replaces the constant name `PLIT_SYMBOL_ADDRESS` with the run-time address of `FUNC_A`.

When the overlay manager subroutine is called via the jump instruction of the PLIT table, `j4` contains the referenced function's overlay ID, and `j5` contains the referenced function's run-time address. The overlay manager subroutine uses the overlay ID and run-time address to load and execute the referenced function.

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

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

- Set up a stack to store register values.

In some cases, stacks may be corrupted by the overlay.

- check if a referenced symbol has already been transferred into its runtime space as a result of a previous reference

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

LDF Programming Examples

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

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

Managing Two Overlays

This example has two overlays, each of which contain two functions. Overlay 1 contains the functions `fft_first_two_stages` and `fft_last_stage`. Overlay 2 contains functions `fft_middle_stages` and `fft_next_to_last`. For example overlay manager source code, see the examples that come with the development software. In this example the overlay manager:

1. Creates and maintains a stack for the registers it uses
2. Determines if the referenced function is in internal memory
3. Sets up a DMA transfer
4. Flushes the cache and executes the referenced function

The following code segment from the LDF in [Listing 2-14](#) represents the overlay definitions. Several code segments for the LDF and the Overlay Manager are displayed and explained in the text.

Listing 2-14. FFT Overlay (part of `CODE{}` section)

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

```

*/
OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT( fft_two.ovl )
    INPUT_SECTIONS( Fft_mid.doj ( program ) )
} > MO_ovly

```

Two overlays are defined: `fft_one.ovl` and `fft_two.ovl`. Both overlays live in the segment `MO_ovly` defined in the memory section of the LDF and run in section `Code`. All instruction and data defined in sections named `program` within the file `Fft_1st_last.doj` are part of overlay `fft_one.ovl`. All instructions and data defined in sections named `program` within the file `Fft_mid.doj` are part of overlay `fft_two.ovl`. The result is two functions within each overlay.

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

Finally, when the last function `fft_one` is referenced, `fft_one`, overlay `id=1` is again transferred to internal memory for execution. The following code segment calls the four functions of FFT:

```

fftrad2:
    call fft_first_2_stages;;
    call fft_middle_stages;;
    call fft_next_to_last;;
    call fft_last_stage;;

wait:
    idle;
    jump wait;

```

LDF Programming Examples

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

```
PLIT{
    j4 = PLIT_SYMBOL_OVERLAYID;
    j5 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

Register `j4` contains the overlay ID that occupies the referenced symbol and register `j5` contains the run time address of the referenced symbol. The final instructions jump the program counter (PC) to the overlay manager routine. The overlay manager routine uses the overlay ID in conjunction with the overlay constants generated by the linker to transfer the proper overlay into internal memory. Once the transfer is complete, the overlay manager sends the PC to the address of the referenced symbol stored on `j5`.

Linker-Generated Constants

The linker generates the following constants used by the overlay manager:

```
.EXTERN _ov_word_size_run_1;;
.EXTERN _ov_word_size_run_2;;
.EXTERN _ov_word_size_live_1;;
.EXTERN _ov_word_size_live_2;;
.EXTERN _ov_startaddress_1;;
.EXTERN _ov_startaddress_2;;
.EXTERN _ov_runtimestartaddress_1;;
.EXTERN _ov_runtimestartaddress_2;;
```

These constants supply the overlay manager with:

1. The size of the overlays, using both run time word sizes and live word sizes
2. The starting address of the live space
3. The starting address of the run space

Overlay Manager Operations

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

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

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

The overlay manager also stores the ID of an overlay currently in internal memory. When an overlay is transferred to internal memory the overlay manager stores the overlay ID in internal memory in the buffer labeled `ov_id_loaded`. Before another overlay is transferred, the overlay manager compares the required overlay ID with that stored in buffer `ov_id_loaded`. 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.

LDF Programming Examples

The following sample segment creates the run-time stack, stores the overlay ID in a modify register and checks the overlay ID stored in

ov_id_loaded:

```
/*
_overlayID has been defined as j4. j4 is set in the PLIT of
LDF. Set up DMA transfer to internal memory through the
external port. Store values of registers used by the overlay
manager in to the software stack.
*/

[j31+ov_stack]   = j9;;
[j31+ov_stack+1] = j10;;
[j31+ov_stack+2] = jL2;;
[j31+ov_stack+3] = j11;;

/* Use the overlay id as an index (must subtract one) */
j4 = j4 - 1;;      /* Overlay ID -1 */
j10 = j4;;         /* Offset into the arrays containing linker
                    defined overlay constants. */

j11 = [j31+ov_id_loaded];;
j4 = j4 - j11;;
if EQ jump continue;;
[j31+ov_id_loaded] = j10;;

j4 = k9; [j31+ov_stack+4] = j4;;
j4 = k10; [j31+ov_stack+5] = j4;;
j4 = kL2; [j31+ov_stack+6] = j4;;
```

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

In summary, the overlay manager routine does the following:

1. Maintains a run-time stack for registers being used by the overlay manager
2. Compares the requested overlay's ID with that of the previously loaded overlay (stored in buffer `ov_id_loaded`)
3. Sets up the DMA transfer of the overlay (if it is not already in internal memory)
4. 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 following example incorporates the ability to transfer one overlay to internal memory while the core executes a function from another overlay. Instead of the core sitting idle while the overlay DMA transfer occurs, the core enables the DMA then begins executing another function. For examples of overlay manager source code, see the examples that come with the development software.

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

There are several circumstances under which an overlay transfer can be in progress while the core is executing another task. Each circumstance can be labeled as deterministic or non-deterministic. A deterministic circumstance is one where you know exactly when an overlay function is required

for execution. A non-deterministic circumstance is one where you cannot predict when an overlay function is required for execution. For example, a deterministic application may consist of linear flow code except for function calls. A non-deterministic example is an application with calls to overlay functions within an interrupt service routine where the interrupt occurs randomly.

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

Since the linker handles all overlay symbol references in the same way (jump to PLIT table then overlay manager), it is up to the overlay manager to distinguish between a symbol reference requesting the load of an overlay function and a symbol reference requesting the execution of an overlay function. In the example, the overlay manager uses a buffer in memory as a flag to indicate whether the function call (symbol reference) is a load or an execute request. The overlay manager first determines if the referenced symbol is in internal memory. If not, it sets up the DMA transfer. If the symbol is not in internal memory and the flag is set for execution, the core waits for the transfer to complete (if necessary) and then executes the overlay function. If the symbol is set for load, the core returns to the instructions that immediately follow the location of the function load reference.

Every overlay function call requires initializing the load/execute flag buffer. In the example, 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 that is 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 if the function call is a load or an execute.

```
call fft_first_2_stages;;
    j4 = 0;;
    [j31+prefetch] = j4;;
call fft_middle_stages;;
    j4 = 0;;
    [j31+prefetch] = j4;;
call fft_next_to_last;;
    j4 = 0;;
    [j31+prefetch] = j4;;
call fft_last_stage;;
    j4 = 0;;
    [j31+prefetch] = j4;;
```

The next set of instructions represents a load function call:

```
call fft_middle_stages;;
    /* This function call pre loads */
    /* the function into the overlay run memory. */
j4 = 1;;
[j31+prefetch] = j4;;
    /* Set prefetch flag to 1 to indicate a load. */
```

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

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

```
OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_ovl.doj( program ) )
} >M0_ovly // Overlay to live in section M0_ovly
```

LDF Programming Examples

```
OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_ovl.doj( program1 ) )
} >M0_ovly // Overlay to live in section M0_ovly

INPUT_SECTIONS( ovly_mgr.doj ( program ) )

OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_two.ovl)
    INPUT_SECTIONS( Fft_ovl.doj(program2) )
} >M0_ovly // Overlay to live in section M0_ovly

OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS( Fft_ovl.doj( program3 ) )
} >M0_ovly // Overlay to live in section M0_ovly
```

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

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

The overlay manager pushes the appropriate registers on the run-time stack. It checks to see if the requested overlay is currently in internal memory. If not, it sets up the DMA transfer. It then checks to see if the function call is a load or an execution call. If it is a load, it begins the transfer and returns the PC back to the instruction following the call. If it is an execution call, the core is idle until the transfer completes (if the

transfer was necessary) and then jumps the PC to the run-time location of the function.



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

Linker Glossary

LDF Input Sections — parts of object files produced by the compiler and assembler consists of various *sections*, referred to as *Input Sections*. Each type of Input Sections holds a particular type of compiled/assembled source code.

LDF Memory Segment — parts of the DSP memory map that are defined in the linker description file.

LDF Output Sections — parts of an executable file are broken up into sections. These sections are defined by the Executable and Linking Format (ELF) file format that the development software uses for executable files. These sections (or segments) are called *Output Sections*, and these sections have Output Section names.

LDF Sections vs. Segments — the Linker takes Input Sections as inputs, places them in an Output Section, and maps Output Sections in selected Memory Segments. The line:

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

directs the linker to take the `program` *Input Section*, place it in the `code` *Output Section*, and map it to the `M0Code` *Memory Segment*.

Link against — the linker resolves symbols to which multiple executables refer. For instance, shared memory executable files (`.SM`) contain sections of code that other processor executables (`.DXE`) link against. Through this process, the shared item is available to multiple executables without being duplicated.

Link objects — object files (`.DOJ`) that get linked and other items, such as executables (`.DXE`, `.SM`, `.OVL`), that are linked against.

Linker description file — the commands, macros, and expressions that control how the linker arranges you program in memory.

Overlays — sections of code or data that are swapped in and out of run-time memory depending on program execution. The linker produces overlay files (`.OVL`) that your overlay manager swaps in and out of memory.

