# 2 LINKER

## Overview

You can use the VisualDSP++ linker, `linker.exe`, to maximize DSP performance by controlling the location of frequently used code and data.

The linker consumes library and object files and produces 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. All information to be used by the VisualDSP++ debugger is provided in the `.DXE` file.

You can supply a linker description file which defines the target memory and the desired mapping of code and data into that memory, or the linker can use a default mapping for the selected DSP chip. The linker generates a memory image file containing a single executable program which may be loaded into a simulator or emulator for testing.

You specify linker options via VisualDSP++ Integrated Development and Debugging Environment (IDDE) in the Project Options dialog box or via DOS command-line inputs.

Most code examples in this manual are written for the ADSP-21062 DSP.

This chapter contains the following information on the linker:

- "Mapping Files To Memory with an LDF" on page 2-3 — introduces the Linker Description File, describing its inputs and outputs, and how it enables your code to run in your target environment
- "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 2-84—provides a series of LDF programming examples for different types of systems
- "Linker Glossary" on page 2-122—provides definitions of linker related terms

# Mapping Files To Memory with an LDF

Whether you link a C/C++ function or an assembly routine, the mechanism is the same --- you use the *Linker Description File* (LDF) to direct linking operation by mapping code or data to specific memory segments. This section explains the overall function of the LDF.

Each DSP project must locate its code and data in the DSP's memory (internal or external) to execute. The LDF specifies the linking process.

You can write your own LDF, using information in this chapter, 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. 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. For an introduction to the LDF commands, and the ways to construct an LDF, refer to "Linker Guide" on page 2-13.

## Linking Process Overview

Using commands in LDF, the linker reads the Input Sections in object files and places them in Output Sections in the executable file, using the commands in the LDF. The LDF defines the DSPs memory and indicates where within that memory the linker has to place the Input Sections.

### Default LDF and Object Code Placement

If you neither write nor import an LDF into your project, VisualDSP++ 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.

One default LDF is provided for each target architecture supported by your VisualDSP++ installation. The default LDF reflects your target environment's memory structure, as specified by your project's options, and locates the program (and data) portions of your object code according to the `-D`*`processor`* command-line switch (`-proc` switch), where *`processor`* is your target architecture.

For more information on LDF syntax, see "Command-Line Syntax" on page 2-25. For sample LDFs and related source files, see the samples included with your VisualDSP++ software.

## The Linking Process and the LDF

Figure 2-1 on page 2-5 shows how the LDF combines information, directing the linker to place program sections in an executable according to the memory available in the DSP system.

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.

The linker maps your program code (and data) within the system memory and processor(s). The linker uses the target system's memory map and segments defined in your source file to create an executable program.

The linking process operates as follows:

- 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. Each Input Section may contain multiple code items, but a code item may appear in one Input Section only.

- The linker maps each Input Section from the object code to an Output Section, as directed by the LDF. More than one Input Section can be placed in an Output Section.

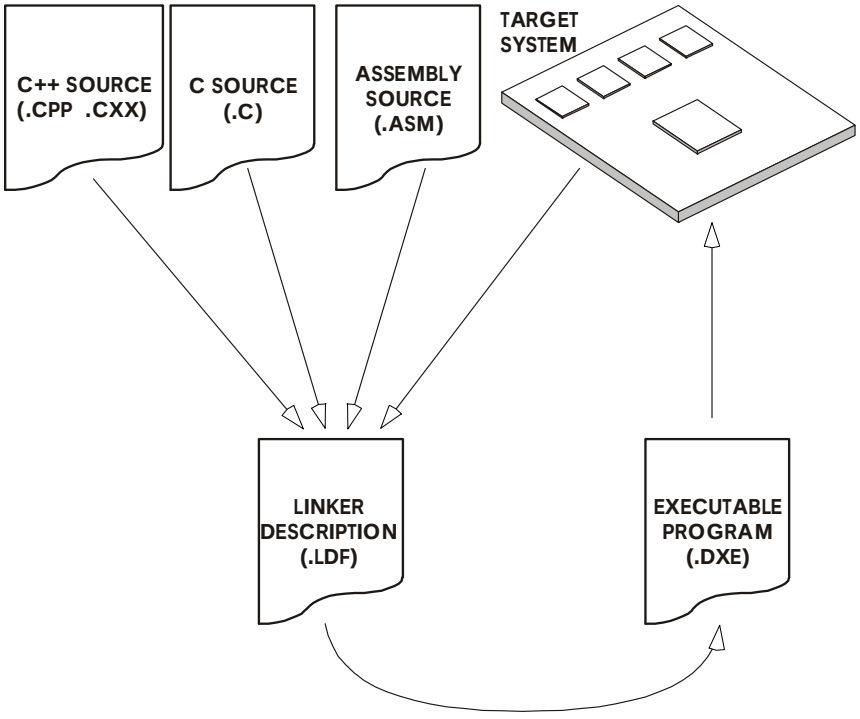


Figure 2-1. The LDF File and Linking Process

The linker may output warning messages and error messages. Be sure to resolve errors to enable the linker to produce valid output.

- The linker maps each Output Section to a *Memory Segment, which is* a contiguous range of memory on the target, as specified by the LDF. Memory Segments have uniform width. Contiguous addresses on different-width hardware must be in different segments. More than one Output Section may map to a single Memory Segment.

Listing 2-1 shows an example LDF (formatted for easy reading). Note that the LDF file includes two commands (`MEMORY` and `SECTIONS`) that combine program and system information.

Listing 2-1. Example Linker Description File

```
ARCHITECTURE(ADSP-21062)          /* see Note 1 on page 2-7 */
SEARCH_DIR( $ADI_DSP\21k\lib )    /* see Note 2 on page 2-7 */
$OBJECT1 = main.doj, $COMMAND_LINE_OBJECTS;
                                  /* see Note 3 on page 2-7 */
MEMORY{                           /* see Note 4 on page 2-8 */
   mem_isr{
     TYPE(PM RAM) START(0x00008000) END(0x000080ff) WIDTH(48)}
   mem_pmco{
     TYPE(PM RAM) START(0x00008100) END(0x000087ff) WIDTH(48)}
   mem_pmda{
     TYPE(PM RAM) START(0x00009000) END(0x00009fff) WIDTH(32)}
   mem_dmda{
     TYPE(DM RAM) START(0x0000c000) END(0x0000dfff) WIDTH(32)}
}
PROCESSOR p0{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
                                  /* see Note 5 on page 2-9 */
   SECTIONS{                      /* see Note 6 on page 2-9 */
      dxe_isr{ INPUT_SECTIONS ($OBJECT1 (isr_tbl) ) }
         > mem_isr
      dxe_pmco{ INPUT_SECTIONS ($OBJECT1 (seg_pmco) ) }
         > mem_pmco
      dxe_pmda{ INPUT_SECTIONS (main.doj (seg_pmda) ) }
         > mem_pmda
      dxe_dmda{ INPUT_SECTIONS (main.doj (seg_dmda) ) }
         > mem_dmda
   } /*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++ IDDE or from the command line (though not all IDDE options have command-line equivalents). In the following discussion, the salient commands for connecting your program to the target DSP are `MEMORY` and `SECTIONS`.

### 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 supported 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. (For more information, refer to page 2-50.)

   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.

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` expands to 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 2-47) is another LDF command-line macro, which expands to a list of all the input object file names on the linker's command line.

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.

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 currently 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 Segments must have distinct names; however, the memory names occupy different namespaces from Input Section and Output Section names. Therefore, a memory segment and an output section may have the same name.

5. The `OUTPUT()` command (on page 2-73) directs the linker to produce an executable (`.DXE`) file, specifying the filename. In this listing, the argument to the `OUTPUT` command is the

$COMMAND_LINE_OUTPUT_FILE macro (on page 2-47). Therefore, the linker names the executable according to the text following its -o switch.

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

6. SECTIONS (on page page 2-74) 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:

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

   directs the linker to take the isr_tbl Input Section, place it in the dxe_isr Output Section, and map it to the mem_isr Memory Segment.

## Inputs—C, C++ & Assembly Sources

The first step toward understanding the LDF is to understand the makeup of files involved in building a DSP executable. The process starts with source files, which contain code written in either C, C++ or Assembly. The first step towards producing an executable is to compile or assemble these sources into *object files*. The VisualDSP++ development software gives object files a .DOJ extension.

The object files produced by the compiler and assembler consist of various *sections*, referred to as *Input Sections*. Each type of Input Sections contains a particular type of compiled/assembled source code. For example, an Input Section may contain program opcodes (48-bits wide) or data such as

variables (16-, 32-, or 40-bits wide). Some Input Sections also can contain debug information.

Each Input Section in the LDF has a unique name which corresponds to a name that 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

A section directive must precede code or data in an assembly source file.

In an assembly source, the 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 /dm asmdata /* declares section asmdata */
.var my_buffer[3];   /* declares a data buffer in smdata */

.section /pm asmcode /* declares section asmcode */
r0 = 0x1234;         /* code in section asmcode */
r1 = 0x4567;
r2 = r1 + r2;
```

In this example, the Input Section `asmdata` contains the array `my_buffer`, and Input Section `asmcode` contains code generated for the three lines of assembly instructions.

### Input Section Directives in C/C++ Source Files

In a C/C++ source file, you can use the optional `section (name)` C language extension to define Input Sections. Because this code uses the `section("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 `extern`. Also during compilation, the compiler puts the variable `temp` in the Input Section `ext_data`.

A fragmentary example follows:

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

section("extern") void func1(void) {
   int x = 1;
}
void func2(void) {
   int i = 0;
}
```

Note that the `section ("name")` extension is optional. As shown in the example, the `func2` function does not use `section ("name")` syntax. If your code does not specify an Input Section name, the compiler uses a default name. The default compiler section names are `seg_pmco` (for code), `seg_dmda` (for `DM` data), and `seg_pmda` (for `PM` data); additional section names are defined in LDF files for use by the linker. In this case, the compiler puts the code from `func2` in an Input Section for program code, which has the default Input Section name `seg_pmco`. For more information on LDF sections, refer to "Specifying the Memory Map" on page 2-16.

For more information about compiler sections, see the *VisualDSP++ C/C++ Compiler & Library Manual for ADSP-21xxx DSPs*.

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

## Outputs—DSP Executables

After you have compiled or assembled source files into object files, use the linker to combine the object files, creating an executable file. By default, the development software gives executable files a `.DXE` extension.

Like object files, the executable is partitioned into *Sections* with their own names. These output 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. Therefore, they are independent and may be replicated in an LDF.

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

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, ...).

# Linker Guide

The linker (`linker.exe`) processes your object, library, and linker description 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, specifying 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 tab of the Project Options dialog box.

*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.

## Linker Operations

All software developers using the linker should be familiar with the following operations:


- "Describing the Link Target" on page 2-14
- "Representing Memory Architecture" on page 2-14
- "Placing Code on the Target" on page 2-20
- "Specifying Linker Options" on page 2-23
- "Linker Error and Warning Messages" on page 2-23

## Describing the Link Target

Before specifying 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 `-D`*`PROCESSOR`* switch (or `-proc` *`processor`* switch) on the linker's command line.

### Representing Memory Architecture

You use the `MEMORY` command to represent the memory architecture of your DSP system. The linker uses this information to place your executable file in the system's memory. The steps for writing a linker `MEMORY` command are as follows:

1. List the ways that your program uses memory in your system. Typical uses for Memory Segments include interrupt tables, initialization data, PM code, PM data, DM data, heap space, and stack space. Refer to "Specifying the Memory Map" on page 2-16.
2. List the types of memory in your system and the address ranges of each type of memory and word width. For ADSP-21xxx DSPs, memory can be qualified as either `PM` or `DM`; and either `RAM` or `ROM`.
3. Construct a `MEMORY{}` command that combines the information in these two lists to declare the Memory Segments in your system. Use Listing 2-2 as code example.

The example in Figure 2-2 is showing an example SHARC system that uses a C program; the information that follows is describing the above steps.

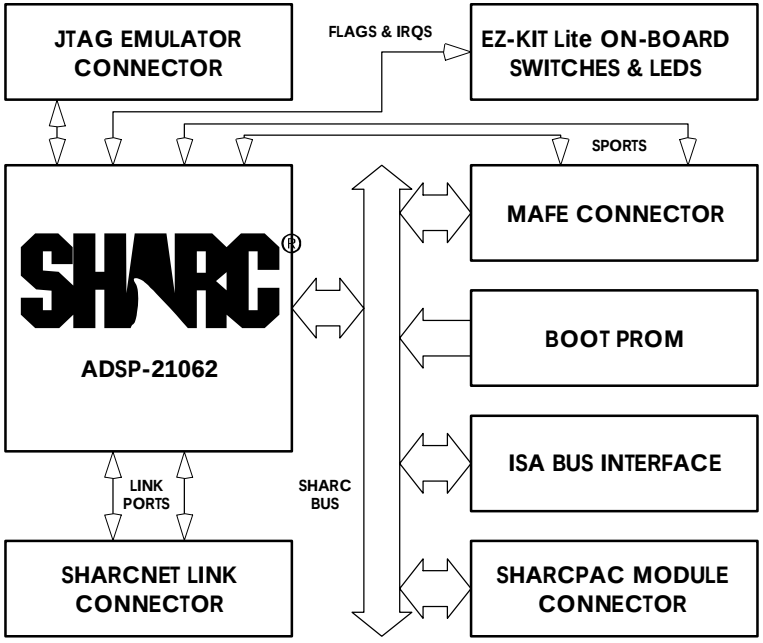


Figure 2-2. Example System for Linker Description

## Specifying the Memory Map

Refer to your processor documentation for a description of the DSP core that lists the widths of the address and data buses. Your program must conform to the constraints imposed by the processor's path widths and addressing capabilities. 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.

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

1. Memory usage — Input section names are generated by the compiler or are specified in the DSP source code. The LDF defines the memory section names and the output section names are defined in the LDF. The memory names are defined in the `MEMORY` command; the output section names are defined in the `SECTIONS` command.

   The default LDF handles all the sections that might be generated by the compiler (the column "Input Section" in Table 2-1). The produced `.dxe` file has appropriate "Output Section(s)" for which material (the corresponding "input section(s)") was found in the inputs. Although the memory labels are used primarily within the LDF, the output section labels can be important to downstream tools. For example, you can invoke `elfdump` to dump the contents of the "`seg_pmda`" section of an executable file.

   Table 2-1 shows correspondences used in the ADSP-21xxx default LDF.

Table 2-1. ADSP-21062 Memory vs. Sections Usage

| Input Section | Output Section | Memory |
|---|---|---|
| seg_pmco | dxe_pmco | mem_pmco (Program Memory code) |
| seg_dmda | dxe_dmda | mem_dmda (Data Memory data) |
| seg_pmda | dxe_pmda | mem_pmda (Program Memory data) |
| heap | dxe_heap | mem_heap (heap space) |
| stackseg | dxe_stak | mem_stak (stack space) |
| sec_rth | dxe_rth | mem_rth (interrupt table/run-time header) |
| seg_init | seg_init | mem_init (initialization data) |
| seg_pmco | dxe_pmco | mem_ovly (overlay) |

2. Memory characteristics — As a memory example, the ADSP-21062 DSP has internal memory addresses from `0x0` to `0x7ffff`, and the characteristics of this memory appear in Table 2-2.

Table 2-2. Example ADSP-21062 Memory

| Block | Memory Range | Word Size |
|---|---|---|
| | `0x00000-0x000ff` | IOP Registers |
| | `0x00100-0x1ffff` | Reserved |
| Block 0 | `0x20000-0x27fff` | (normal word) 32- or 48-bit |
| Block 1 | `0x28000-0x2ffff` | (normal word) 32- or 48-bit |
| Block 1 alias | `0x30000-0x37fff` | (normal word) 32- or 48-bit |
| Block 1 alias | `0x38000-0x3ffff` | (normal word) 32- or 48-bit |
| Block 0 | `0x40000-0x4ffff` | (short word) 16-bit |
| Block 1 | `0x50000-0x5ffff` | (short word) 16-bit |
| Block 1 alias | `0x60000-0x6ffff` | (short word) 16-bit |
| Block 1 alias | `0x70000-0x7ffff` | (short word) 16-bit |
| | `0x404000-0x404001` | (MAFE ports) 32-bit |

3. Linker `MEMORY{}` Command — referring to steps 1 and 2, you specify the SHARC EZ-KIT Lite's memory with the `MEMORY{}` command in Listing 2-2.

Listing 2-2. Linker MEMORY{} Command Example

```
    /* start 21062_memory.h file that is referred to in
    the "LDF Programming Examples" on page 2-84 */

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

    mem_rth {
      TYPE(PM RAM) START(0x20000) END(0x200ff) WIDTH(48)}
    mem_init {
      TYPE(PM RAM) START(0x20100) END(0x201ff) WIDTH(48)}
    mem_pmco {
      TYPE(PM RAM) START(0x20200) END(0x249ff) WIDTH(48)}
    mem_pmda {
      TYPE(PM RAM) START(0x24a00) END(0x24fff) WIDTH(40)}
    mem_dmda {
      TYPE(DM RAM) START(0x2a000) END(0x2bfff) WIDTH(32)}
    mem_heap {
      TYPE(DM RAM) START(0x2c000) END(0x2dfff) WIDTH(32)}
    mem_stak {
      TYPE(DM RAM) START(0x2e000) END(0x2ffff) WIDTH(32)}
    mafeadrs {
      TYPE(DM RAM) START(0x404000) END(0x404000) WIDTH(32)}
    mafeadrs {
      TYPE(DM RAM) START(0x404001) END(0x404001) WIDTH(32)}
// end 21062_memory.h file
```

## Placing Code on the Target

As defined by the MEMORY command, the SECTIONS command places your program's Input Sections in the memory of a DSP system. Each Input Section is declared as such in your assembly code.

You must embed SECTIONS commands within the linker's PROCESSOR{} or SHARED_MEMORY{} commands. These commands inform the linker to place the code in memory allocated to that processor, shared among multiple processors.

To write a linker SECTIONS{} command, per system architecture in Figure 2-2 on page 2-15 and Listing 2-2:

1. List each of the assembly code .SECTION directives in your DSP program, identifying their memory types (PM or DM) and noting when location is critical to their operation. These .SECTION portions include interrupt tables, data buffers, and on-chip code or data.
2. Compare this list with the segments you defined in the MEMORY command, identifying the Memory Segment in which each .SECTION must be placed.
3. Combine the information from these two lists to write one or more linker SECTIONS command(s). Combining the information from steps 1 and 2, you could specify how to place code for the system with the SECTIONS{} command in Listing 2-3.
4. Use Program sections in the C/C++ compiler to produces code that uses memory in predefined ways, with predefined section labels.

Listing 2-3. Linker SECTIONS{} Command Example

```
        /* start 21062_sections.h file that is referred to in
        the "LDF Programming Examples" on page 2-84 */

// SECTIONS {

// begin output sections
    dxe_rth { // run-time header & interrupt table
        INPUT_SECTIONS( $OBJS(seg_rth) $LIBS(seg_rth))
        } >mem_rth
    dxe_init { // initialization data
        INPUT_SECTIONS( $OBJS(seg_init) $LIBS(seg_init))
        } >mem_init
    dxe_pmco { // PM code
        INPUT_SECTIONS( $OBJS(seg_pmco) $LIBS(seg_pmco))
        } >mem_pmco
    dxe_pmda { // PM data
        INPUT_SECTIONS( $OBJS(seg_pmda) $LIBS(seg_pmda))
        } >mem_pmda
    dxe_dmda { // DM data
        INPUT_SECTIONS( $OBJS(seg_dmda) $LIBS(seg_dmda))
        } >mem_dmda
    stackseg {
    // Declare the stack space and length
    //
    // The linker will generate the following symbols
    //    "ldf_stack_space "
    //    "ldf_stack_length"
    // These symbols will be entered into the DXE's symbol
    // table.
    // These symbol definitions are required since there are
    // references to them in seg_init.asm (where the stack
    // and heap variables are initialized)

    ldf_stack_space = .;
    ldf_stack_length = 0x2000;
    } > seg_stak
    heap {
    // Declare the heap space and length
    //
    // The linker will generate the following symbols
    //    "ldf_heap_space"
```

```
        //    "ldf_heap_length"
        //    "ldf_heap_end"
        //
        // These symbols will also be entered into the DXE's
        // symbol table.
        // These symbol definitions are required since there are
        // references to then in seg_init.asm (where the stack
        // and heap variables are initialized)

        ldf_heap_space = .;
        ldf_heap_end = ldf_heap_space + 0x2000;
        ldf_heap_length = ldf_heap_end - ldf_heap_space;
        } > seg_heap

    } // end sections
```

## Using Linker Features

The two previous sections, "Describing the Link Target" on page 2-14 and "Placing Code on the 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.

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 2-46.

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 2-88.
- For information on shared memory, see the command syntax for "SHARED_MEMORY{}" on page 2-81 and the example in "Linking for Multi-Processor and Shared Memory" on page 2-88

- For information on overlay memory, see the command syntax for "SECTIONS{}" on page 2-75 and "PLIT{}" on page 2-69. Examples of overlay linking appear in "Linking for Overlay Memory" on page 2-95 and "Using a Procedure Linkage Table" on page 2-98.

## Specifying Linker Options

When developing within the VisualDSP++ environment, you can specify default tool settings for your project files. You may find it useful to modify the linker's default option settings in the VisualDSP++ IDDE. You can do this via the Link tab of the Project Options dialog box.

For more information, see the *VisualDSP++ User's Guide for ADSP-21xxx DSPs* or VisualDSP++ online help.

The linker also has command-line switches that correspond to the Link dialog box options. For more information on the linker command-line operation, see "Linker Command-Line Reference" on page 2-25.

## Linker Error and Warning Messages

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

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

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 containing the error,

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

```
[Error Li2005] my_file.ldf:177 Expected token was not found.:
Expected ‘Eof’ Before ‘3’
```

This error indicates that the linker expected end-of-file but encountered the character `3` on line `177` in `my_file.ldf`.

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 VisualDSP++ displays the line in the source file that contains the error. However, some build errors — such as a 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 ADSP-21xxx DSPs*.

Errors relating to missing cross-references often stem from omissions in the LDF. For example, if a Memory Segment from the objects is not placed by the LDF, there will be a cross-reference error in every object that refers to labels in the missing section. You can solve this problem by reviewing the LDF and correcting it to specify all sections that need placement.

# Linker Command-Line Reference

This section provides reference information on linker command-line switches. A description of each switch appears in Table 2-4 on page 2-30.

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

When you use the linker within the VisualDSP++ 2.0 IDDE, the settings in the Link tab correspond to linker command-line switches. The VisualDSP++ IDDE calls the linker with those settings when you link your code. For more information, see the *VisualDSP++ 2.0 User's Guide for ADSP-21xxx DSPs*.

## Command-Line Syntax

You can use one of the following normalized formats:

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

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

Other switches are optional, and some commands are mutually exclusive. For example,

```
linker -DADSP-21062 p0.doj p1.doj -T target.ldf -t -o program.dxe
```

Analog Devices suggest that you use `-proc processorID` instead of `-Darchitecture` on the command line to make the target processor selection. See Table 2-4 on page 2-30 for more information.

## Object Files in the Linker Command Line

The command line must list at least one object file to be linked. These files may be of several different types.

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

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` file name 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 2-3 on page 2-29 lists valid extensions and matching linker operations.

* "Link Against" is described on page 2-49, under `$COMMAND_LINE_LINK_AGAINST`

## Switch Format in the Linker Command Line

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

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

As noted above, the linker command line (except for file names) 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` — Uses the LDF listed to specify executable program placement
- `-t` — Turns on trace information, echoing each link object's name to `stdout` as it is processed
- `-o program.dxe` — Names the linked, executable output file

## File Names on the Linker Command Line

Many linker switches take a file name as an optional parameter. Table 2-3 lists the 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 the default directory, the linker searches for the file in the search directories that you specify 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 by the `-T` switch, the linker searches for the LDF in the current working directory. If you use a default LDF by omitting any LDF information in the command line and instead specifying `-Darchitecture`, the linker searches in the processor-specific `ldf` directory; for example, ...`\$ADI_DSP\21062\ldf`.

For more information on file search, see "LDF Macros" on page 2-46.

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

- Use a space to delimit file names in a list of input files
- Enclose long file names within straight quotes; for example, `"long file name"`
- Include the appropriate file name extension with each file name

The linker opens existing files and verifies their type before processing. When the linker creates a file, it uses the file extension to determine the type of file to create. COFF format files from previous software tools releases are supported and the linker performs the appropriate file conversion before linking.

The linker follows the conventions for file name extensions that appear in Table 2-3.

Table 2-3. 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 Switches

This section describes the linker command-line switches. A list of all switches appears in Table 2-4, and a description of each switch appears starting on page 2-32.

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

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

Table 2-4. Linker Switch Summary

| Switch Name | Description |
|---|---|
| `objects` on page 2-32 | Specifies object files involved in the linking; process files that are not parameters to a switch. |
| `@` *`file`* on page 2-33 | Directs the linker to use the specified file as input on the command line. |
| `-Darchitecture` on page 2-33 | Specifies the target architecture (processor). |
| `-e` on page 2-35 | Directs the linker to eliminate unused symbols from the executable. |
| `-es secName` on page 2-35 | Names sections (*`secName`* list) to which elimination algorithm is being applied. |

Table 2-4. Linker Switch Summary (Cont'd)

| Switch Name | Description |
|---|---|
| `-ev` on page 2-35 | Eliminate unused symbols from the executable verbosely, displaying all objects that were eliminated. |
| `-h\|-help` on page 2-35 | Outputs the list of command-line switches and exits. |
| `-i` *`path`* on page 2-35 | Includes search directory for preprocessor include files. |
| `-ip` on page 2-35 | Fills in fragmented memory with individual data objects that fit. Also requires objects to have been assembled using the assembler's `-ip` switch.<br>NOT supported for SHARC DSPs in this release. |
| `-keep symName` on page 2-36 | Retains unused symbols. |
| `-L` *`path`* on page 2-33 | Adds the path name to search libraries for objects. |
| `-M` on page 2-34 | Produces dependencies. |
| `-MM` on page 2-34 | Builds and produces dependencies. |
| `-Map` *`filename`* on page 2-34 | Outputs a map of link symbol information to a file. |
| `-MD`*`macro[=def]`* on page 2-34 | Defines and assigns value `def` to preprocessor `macro`. |
| `-o` *`filename`* on page 2-36 | Outputs the named executable file. |
| `-pp` on page 2-36 | Stops after preprocessing. |
| `-proc` *`ProcessorID`* | Directs the linker to select a target processor. |
| `-S` on page 2-34 | Omits debugging symbol information from the output file. |
| `-s` on page 2-36 | Strips symbol information from the output file. |
| `-sp` on page 2-36 | Skips preprocessing. |

Table 2-4. Linker Switch Summary (Cont'd)

| Switch Name | Description |
|---|---|
| -T *filename* on page 2-34 | Names the LDF. |
| -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 *filename* on page 2-37 | Outputs a list of all cross-referenced symbols. |

### objects

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

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

The linker treats object (`.doj`) and library (`.dlb`) files that appear on the command line as object files to be linked. For more information on objects, see the `$COMMAND_LINE_OBJECTS` linker macro on page 2-47.

The linker treats executable (`.dxe`) and shared-memory (`.sm`) files 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 2-47. If you do not specify link objects on the command line or in the linker description file, the linker generates an appropriate information/error message.

**<null>**

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

**@ *file***

Uses *`file`* as input to the linker command line. This switch allows you to 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`.

**-D*architecture***

Specifies the target architecture.
No whitespace is permitted between `-D` and *`architecture`*.
*`architecture`* 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 causes an error.

**-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

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

### -MM

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

### -Map *file*

Outputs a map of link symbol information to a *`file`*, which can have any name. The *`file`* parameter is obligatory. The linker names the file with a `.MAP` extension. The whitespace is obligatory before *`file`*; otherwise, the link fails.

### -MD*macro*[=*def*]

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

### -S

Omits debugging symbol information (*not* all symbol information) from the output file. Compare with `-s` switch (on page 2-36).

### -T *file*

Uses *`file`* to name an LDF.
The LDF specified following the `-T` switch must contain an `ARCHITECTURE()` command if the command line does not have `-Darchitecture`.
The linker "requires" the `-T` switch when linking for a processor for which

no IDDE support has been installed (e.g., the processor ID does not show up in the `Target processor` field of the Project Options dialog box.)

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

**-e**

Eliminates unused symbols from the executable.

**-es *secName***

Names sections (*`secName`* list) to which the 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.

**-h|-help**

Displays a summary of command-line options and exits.

**-i *path***

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

**-ip**

Fills in fragmented memory with individual data objects that fit.

NOT supported for SHARC DSPs in this release of VisualDSP++.

### -keep *symName*

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

### -o *filename*

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

### -pp

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

### -proc *ProcessorID*

Directs the linker to select a target processor.

If you do not specify your target, the default is `ADSP-21062` (also supports ADSP-21060 DSPs and ADSP-21061 DSPs.
To select ADSP-21065L DSP, enter `ADSP-21065L`.
To select ADSP-21160 DSP, enter `ADSP-21160`.
To select ADSP-21161 DSP, enter `ADSP-21161`.

### -s

Strips all symbols. Directs the linker to omit all symbol information from the output file.

### -sp

Skips preprocessing. Links without preprocessing the LDF.

**-t**

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

**-v**

(Verbose) — Outputs status information while linking.

**-version**

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

**-warnonce**

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

**-xref *filename***

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

# Linker Description File Reference

An LDF allows you to develop code for any system that contains a DSP. The syntax of the LDF defines your system to the linker and specifies how the linker processes executable code for your system. This reference describes LDF syntax and provides LDF examples for typical systems.

This section includes the following topics:


- "LDF Structure" on page 2-39
- "LDF Expressions and Conventions" on page 2-40
- "Linker Keywords" on page 2-42
- "LDF Operators" on page 2-44
- "LDF Macros" on page 2-46
- "LDF Command Summary" on page 2-49
- "LDF Programming Examples" on page 2-84


Because the linker runs the preprocessor on the LDF, you can use any preprocessor commands (such as `#define`) within your LDF. For more information on preprocessor commands, see the *VisualDSP++ 2.0 Assembler & Preprocessor Manual for ADSP-21xxx DSPs*.

## LDF Structure

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

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

For more information on the LDF structure, see "Describing the Link Target" on page 2-14, "Placing Code on the Target" on page 2-20, and "LDF Programming Examples" on page 2-84.

### Command Scoping

The two LDF file scopes are *global* and *command*. A command scope defines the content for an `OUTPUT()` command. You can use an `OUTPUT()` command within 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 command scopes are limited to those scopes. Note that LDF macros are available globally regardless of the scope where the macro is defined (see "LDF Macros" on page 2-46).

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

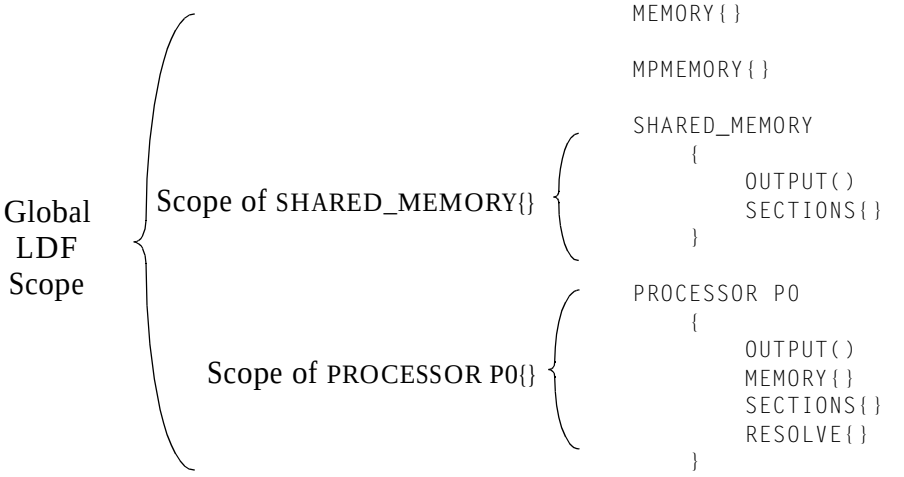


Figure 2-3. LDF Command Scoping Example

## LDF Expressions and Conventions

Table 2-5 lists the linker's non-keyword operators and conventions.

Table 2-5. Linker Non-Keyword Operators and Conventions

| Convention | Description |
|---|---|
| . | A dot "." in an `address expression` refers to the current location pointer. |
| `0x`*`number`* | A "`0x`" prefix indicates a hexadecimal number. |
| *`number`* | A number without a prefix is a decimal number. |

Table 2-5. Linker Non-Keyword Operators and Conventions (Cont'd)

| Convention | Description |
|---|---|
| *number*k (or K) | A decimal number multiplied by 1024. |
| /* comment */ | C-style comments: These can cross newline boundaries until */ is encountered. |
| // comment | A "//" string precedes single-line C++ style comments |

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

- 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
- 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 linker keywords from Table 2-6 appear in the following sections:

- "Miscellaneous LDF Keywords" on page 2-44
- "LDF Operators" on page 2-44
- "LDF Macros" on page 2-46
- "LDF Command Summary" on page 2-49

The keywords in Table 2-6 are case-sensitive; the linker only recognizes a keyword when the entire word is UPPERCASE.

Table 2-6. Linker Keywords and Operators

| | | |
|---|---|---|
| ABSOLUTE on page 2-44 | ADDR on page 2-45 | ALGORITHM on page 2-80 |
| ALIGN on page 2-50 | ALL_FIT on page 2-80 | ARCHITECTURE on page 2-50 |
| BEST_FIT on page 2-80 | BOOT on page 2-44 | DEFINED on page 2-45 |
| DM on page 2-55 | ELIMINATE on page 2-51 | ELIMINATE_SECTIONS on page 2-51 |
| END on page 2-56 | FALSE on page 2-44 | FILL on page 2-78 |
| FIRST_FIT on page 2-80 | INCLUDE on page 2-49 | INPUT_SECTION_ALIGN on page 2-52 |
| INPUT_SECTIONS on page 2-77 | KEEP on page 2-52 | LENGTH on page 2-56 |
| LINK_AGAINST on page 2-49 | MAP on page 2-49 | MEMORY on page 2-54 |

Table 2-6. Linker Keywords and Operators

| | | |
|---|---|---|
| MEMORY_SIZEOF on page 2-45 | MPMEMORY on page 2-56 | NUMBER_OF_OVERLAYS on page 2-80 |
| OUTPUT on page 2-73 | OVERLAY_ID on page 2-80 | OVERLAY_INPUT on page 2-80 |
| OVERLAY_GROUP on page 2-57 | OVERLAY_OUTPUT on page 2-80 | PACKING[1] |
| PLIT on page 2-69 | PLIT_DATA_OVERLAY_IDS on page 2-72 | PLIT_SYMBOL_ADDRESS on page 2-72 |
| PLIT_SYMBOL_OVERLAY ID on page 2-72 | PM on page 2-55 | PROCESSOR on page 2-72 |
| RAM on page 2-55 | RESOLVE on page 2-74 | RESOLVE_LOCALLY on page 2-81 |
| ROM on page 2-55 | SEARCH_DIR on page 2-74 | SECTIONS on page 2-75 |
| SHARED_MEMORY on page 2-81 | SHT_NOBITS on page 2-77 | SIZE on page 2-81 |
| SIZEOF on page 2-45 | START on page 2-56 | TYPE on page 2-55 |
| VERBOSE on page 2-51 | WIDTH on page 2-56 | XREF on page 2-44 |

## Miscellaneous LDF Keywords

This section describes the linker keywords that are not operators, macros, or commands. For more information about linker keywords that are operators, see "LDF Operators" on page 2-44. For more information about linker keywords that are macros, see "LDF Macros" on page 2-46. For more information about linker keywords that are commands, see "LDF Command Summary" on page 2-49.

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

BOOT—Boot memory, memory from which a ADSP-21xxx DSP can be booted.

DM—Data Memory, the default memory space for all variables.

FALSE—A constant with the value of `0`.

PM—Program Memory, the memory space for functions.

TRUE—A constant with a value of `1`.

XREF—A cross-reference option setting.

## 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()`. Use this operator to assign 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(). Use this operator to assign a section's absolute address to a symbol.

- DEFINED(*symbol*)

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

- 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 appear only within the SECTIONS{} command. 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.
- 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:

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

A macros is identified with leading dollar sign ($).

LDF macros funnel input from the linker command line into predefined macros and provide support for user-defined macro substitutions. Linker macros are available globally in the LDF regardless of where they are defined. For more information on these topics, see "Command Scoping" on page 2-39 and "LDF Macros and Command-Line Interaction" on page 2-48.

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

### LDF Macro List

The linker provides 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. Use this macro within the INPUT_SECTIONS{} syntax of the linker's SECTIONS{} command. This macro provides a comprehensive list of object file input that the linker searches for input sections.

- $COMMAND_LINE_LINK_AGAINST

  The linker expands 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 PROCESSOR{} 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. Use this macro only once in your LDF for file name substitution within an OUTPUT() command.

- `$ADI_DSP`

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

- *`$macro = list_of_files ;`*

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

## LDF Macros and Command-Line Interaction

Whether you run the linker from the VisualDSP++ environment or from a command line, the linker gets its commands through a command-line interface. Many linker operations, such as input, output, and link against files can be controlled through the command line. Using LDF macros, you can apply these command-line inputs throughout your LDF. For more information, see “LDF Macros” on page 2-46.

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

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 specify only a single output from the command line, an LDF that relies on command-line input should be written to produce one output file for a single-processor system.

   The linker command-line interface does not allow you to name the multiple outputs 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. From VisualDSP++, you can name multiple outputs that you need for multiprocessor, shared memory, or overlay memory.

## LDF Command Summary

Commands in the LDF define the target system and specify the order in which the linker processes output for that system. Linker commands operate within a scope, influencing the operation of other commands that appear within the range of that scope. For more information, see "Command Scoping" on page 2-39.

The linker supports the following LDF commands:


- "ALIGN()" on page 2-50
- "ARCHITECTURE()" on page 2-50
- "ELIMINATE()" on page 2-51
- "ELIMINATE_SECTIONS()" on page 2-51
- "INCLUDE()" on page 2-51
- "INPUT_SECTION_ALIGN()" on page 2-52
- "KEEP()" on page 2-52
- "LINK_AGAINST()" on page 2-52
- "MAP()" on page 2-53
- "MEMORY{}" on page 2-54
- "MPMEMORY{}" on page 2-56
- "OVERLAY_GROUP{}" on page 2-57

- "PACKING()" on page 2-61
- "PLIT{}" on page 2-69
- "PROCESSOR{}" on page 2-72
- "RESOLVE()" on page 2-74
- "SEARCH_DIR()" on page 2-74
- "SECTIONS{}" on page 2-75
- "SHARED_MEMORY{}" on page 2-81


## ALIGN()

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

## ARCHITECTURE()

The linker's `ARCHITECTURE()` command specifies the processor in your target system. Your LDF may contain only one `ARCHITECTURE()` command. The command must appear in a global LDF scope, applying to the entire linker description file.

SYNTAX: `ARCHITECTURE(`*processor*`)`

The `ARCHITECTURE()` command is case-sensitive. Hence, `ADSP-21160` is a legal value but `adsp-21160` is not.

If you do not specify the target processor with the `ARCHITECTURE()` command in the LDF, 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.

## ELIMINATE()

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

## ELIMINATE_SECTIONS()

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

## INCLUDE()

Specifies an additional LDF that the linker processes before processing the remainder of the current LDF. You may specify any number of additional LDFs. Supply one filename per `INCLUDE()` command. Each LDFs must specify the same `Architecture()`, though only one is obligated to do so. Normally, that is the top-level LDF, which calls the others.

## INPUT_SECTION_ALIGN()

The `INPUT_SECTION_ALIGN(address_boundary_expression)` command is valid only within the scope of an output section. For more information, see "Command Scoping" on page 2-39. For more information on output sections, see the syntax description for "SECTIONS{}" on page 2-75.)

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 2-78.

The linker aligns each input section (instruction or data) placed in an output section with the address specified by the *address_boundry_expression*. The address boundary expression (a power of 2) is a word boundary (address). Legal values for this expression depend on the word size of the segment that receive the output section being aligned.

## KEEP()

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

## LINK_AGAINST()

To link multiprocessor programs for ADSP-21xxx 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 n
{
    ...
    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. Multiple file names must be 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.

You can override the search order for a specific variable or label by using the RESOLVE() command, which directs the linker to ignore LINK_AGAINST() for a specific symbol. LINK_AGAINST() for other symbols still applies. Example LDFs containing the LINK_AGAINST() and RESOLVE() commands are useful for seeing how this process works. For more information, see Listing 2-9 on page 2-89.

## MAP()

The MAP() command outputs a map file with the specified name. You must supply the *file name*. 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 (Link tab of the Project Options dialog box), the linker runs with -Map <*projectname*>.map asserted, and your LDF's MAP() command generates a warning.

## MEMORY{}

The linker's MEMORY{} command specifies the memory map of your target system. After you declare memory segment names with this command, the linker uses the memory segment names to place program SECTIONs through the SECTIONS{} command.

Your LDF may contain a MEMORY{} command that applies to each processor's scope and must contain a MEMORY{} command for any global memory on your target system. There is no limit to the number of segments you can declare within each MEMORY{} command. For more information, see "Command Scoping" on page 2-39.

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

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

The syntax for the MEMORY{} command appears in Figure 2-4 on page 2-55, followed by definitions for the command's components.

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

- *segment_commands*

  Declares your target processor's memory segments. Although 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.

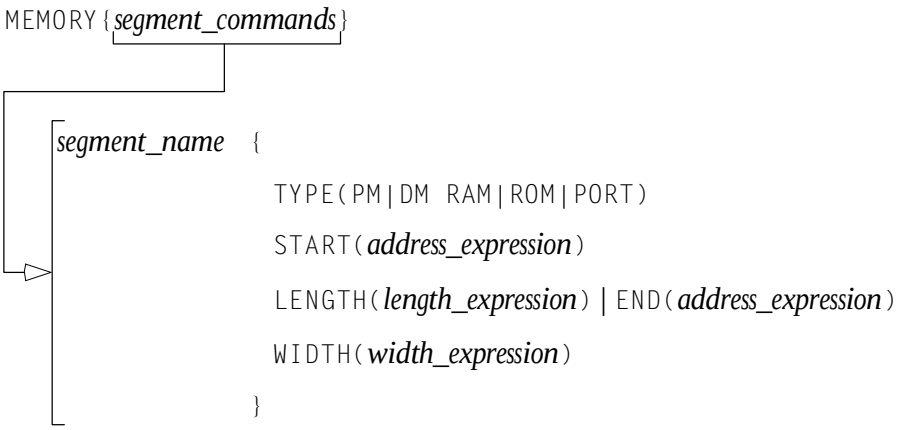


Figure 2-4. Syntax Tree of the MEMORY{} Command

- *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(PM|DM RAM|ROM|PORT)`

  Identifies the architecture-specific type of memory within the segment. (Note: not all target processors support all types of memory.) The linker stores this information in the executable for use by other development tools, such as the PROM splitter. A valid `TYPE()` command specifies the type of memory usage (`PM` for program memory or `DM` for data memory) and the memory's functional or hardware locus (`RAM`, `ROM`, or `PORT`).

- START(*address_expression*)

  Identifies the segment's start address. 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 segment length in words or sets the segment's end address. When 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. When 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 the number of bits per word within the region or an expression that evaluates to the number of bits per word.

## MPMEMORY{}

The 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 LDF, and any other LDFs it includes, may contain only one MPMEMORY{} command. The maximum number of processors that you can declare is architecture-specific. Follow the MPMEMORY{} command with PROCESSOR *processor_name*{} commands containing each processor's MEMORY{} and SECTIONS{} commands.

The syntax for the MPMEMORY{} command appears in Figure 2-5, followed by definitions for the command's components.

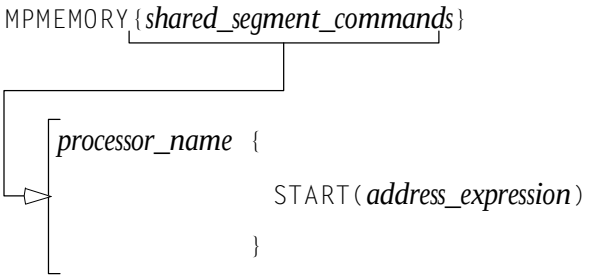


Figure 2-5. Syntax Tree of the MPMEMORY{} Command

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's 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 2-72.

## OVERLAY_GROUP{}

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

*Overlaying* improves 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 may be *grouped* or *ungrouped*. Ungrouped overlays execute from a single starting address in "run" memory. The `OVERLAY_GROUP` command allows you to group overlays, so that each group has its own run-time memory.

Grouping through `OVERLAY_GROUP` allows you to define a series of overlays sequentially in the LDF, but segment the "run" memory so that multiple overlays may reside in "run" memory simultaneously.

Figure 2-6 and Figure 2-7 demonstrate ungrouped versus grouped overlays.

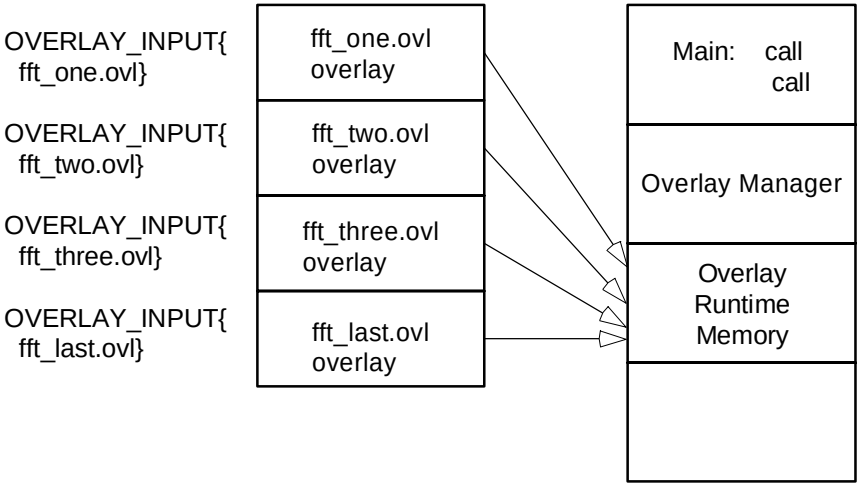


Figure 2-6. Overlays, Not Grouped

Ungrouped overlays are executed from *a* single starting address in "run" memory. The `OVERLAY_GROUP` command lets you group overlays, so that one of each group of overlays resides in run-time memory, running the

overlay for each group from a *different* starting address in run-time memory.

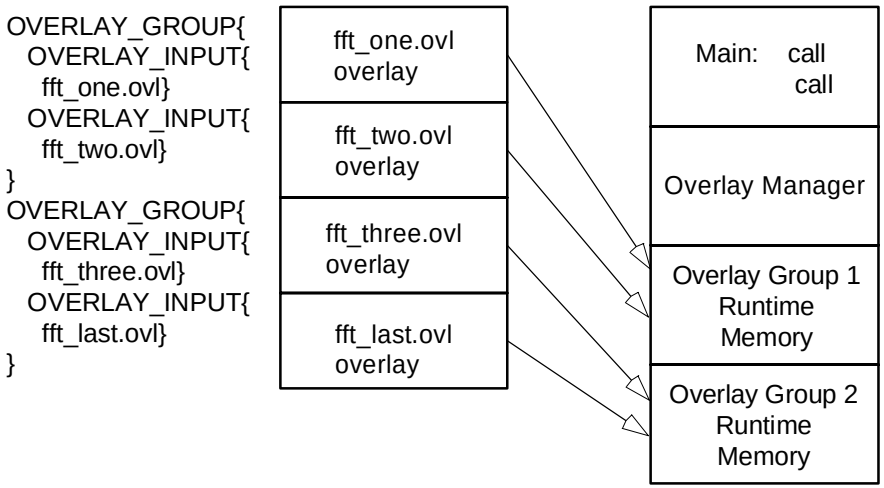


Figure 2-7. Overlays, Grouped

In the examples in Listing 2-4 and Listing 2-5, the functions are written to *overlay files* (`*.OVL`). It does not matter (except to a DMA that brings them in) whether these are disk files or Memory Segments. The overlays are active only when executed in run-time memory, all of which is located in segment `pmco`.

Overlay declarations syntactically resemble `SECTIONS{}` commands; they are portions of `SECTIONS{}` commands.

Listing 2-4. LDF Overlays, Not Grouped

```
// Declare which functions reside in which overlay
// The overlays have been split into different segments
// if in the same file or different files.
// The overlays declared in this section, pm_code, will run
// in pm_code.

OVERLAY_INPUT {
   ALGORITHM(ALL_FIT)
   OVERLAY_OUTPUT(fft_one.ovl)
   INPUT_SECTIONS(  Fft_1st.doj(pm_code) )
   } >ovl_code  // Overlay to live in section ovl_code
OVERLAY_INPUT {
   ALGORITHM(ALL_FIT)
   OVERLAY_OUTPUT(fft_two.ovl)
   INPUT_SECTIONS(  Fft_mid.doj(pm_code) )
   } >ovl_code // Overlay to live in section ovl_code
OVERLAY_INPUT {
   ALGORITHM(ALL_FIT)
   OVERLAY_OUTPUT(fft_three.ovl)
   INPUT_SECTIONS(  Fft_last.doj(pm1_code) )
   } >ovl_code // Overlay to live in section ovl_code
OVERLAY_INPUT {
   ALGORITHM(ALL_FIT)
   OVERLAY_OUTPUT(fft_last.ovl)
   INPUT_SECTIONS(  Fft_last.doj(pm2_code) )
   } >ovl_code // Overlay to live in section ovl_code
```

Listing 2-5. LDF Overlays, Grouped

```
// Declare which functions reside in which overlay
// The overlays have been split into different
// segments if in the same file or different files.
// The overlays declared in this section, pm_code, will run
// in pm_code.

OVERLAY_GROUP {
   OVERLAY_INPUT {
      ALGORITHM(ALL_FIT)
      OVERLAY_OUTPUT(fft_one.ovl)
      INPUT_SECTIONS(  Fft_1st.doj(pm_code) )
```

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

OVERLAY_GROUP {
    OVERLAY_INPUT {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_three.ovl)
        INPUT_SECTIONS(  Fft_last.doj(pm1_code) )
        } >ovl_code // Overlay to live in section ovl_code
    OVERLAY_INPUT {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(fft_last.ovl)
        INPUT_SECTIONS(  Fft_last.doj(pm2_code) )
        } >ovl_code // Overlay to live in section ovl_code
}
```

## PACKING()

The DSP exchanges data with its environment (on-chip or off-chip) through several buses. The configuration, placement, and amounts of memory are end-product-specific. You can specify memory of width(s) and data transfer byte order(s) that suit your needs.

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

The `PACKING()` command allows the linker to structure its executable output to be consistent with your installation's memory organization. It can be applied (scoped) on a segment-by-segment basis within the LDF, with adequate granularity* to handle heterogeneous memory configurations.

* Any segment needing more than one packing command can be divided into homogeneous segments.

The PACKING() command syntax is:

```
PACKING (number_of_bytes, byte_order_list)
```

where:

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

Note that first byte is B1 (not B0), second byte is B2, etc.

For example,

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

Non-default use of the PACKING() command *reorders* bytes in executable files (.DXE .SM, or .OVL), so they arrive at the target in the correct number, alignment, and sequence. To accomplish this task, the command must know the size of the reordered group, the byte order within the group, and whether and where null* bytes must be inserted to preserve alignment on the target.

The order used to place bytes in a memory 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. On an ADSP-21xxx DSP,

* In this case, "null" refers to usage: the target ignores a null byte. Coincidentally, the linker sets these bytes to all 0s.

PACKING() applies to the External Port. Each external port buffer contains data packing logic that allows 8-, 16- or 32-bit external bus words to be packed into 32- or 48-bit internal words: This logic is fully reversible.

## Packing in ADSP-21xxx LDF

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

Your VisualDSP++ software comes with the packing.h file in the ...21k\include folder. This file provides macros defining packing commands to be used in a linker description file (LDF). The file provides macros to support the various types of packing for DMA (used in overlays) and for direct external execution. To use these macros, place them in the SECTIONS portion of the LDF whereever a PACKING() command is needed.

In some DMA modes, ADSP-2106x processors unpack three 32-bit words to build two 48-bit instruction words when the processor receives data from 32-bit memory. For example, the following unpacked order and storage order could apply to a DMA mode of ADSP-21xxx DSP (B = byte):

Table 2-7. Packing Order for DMA

| Unpacked Order:<br>Two 48-Bit Internal Words (after the third transfer) | Transfer order from storage in a 32-Bit External |
|---|---|
| Word1, bits 47-0<br>(B1,B2,B3,B4,B5,B6)<br><br>Word2, bits 47-0<br>(B7,B8,B9,B10,B11,B12) | 1. Word1, bits 47-32 (B1 & B2)<br>Word1, bits 31-16 (B3 & B4)<br>2. Word2, bits 15-0 (B11 & B12)<br>Word1, bits 15-0 (B5 & B6)<br>3. Word2, bits 47-32 (B7 & B8)<br>Word2, bits 31-16 (B9 & B10) |

Note that the order of unpacked bytes does NOT match the transfer (stored) order. Because there are two bytes per short word used in the ADSP-21xxx DSP, the above transfer translates into the following format (where `B` = Byte), as shown in Table 2-8:

Table 2-8. Unpack vs. Storage Order

| Unpacked Order:<br>Two 48-Bit Internal Words | Storage Order:<br>In 32-Bit External |
|---|---|
| B1, B2, B3, B4, B5, B6,<br>B7, B8, B9, B10, B11, B12 | B1, B2, B3, B4, B11, B12,<br>B5, B6, B7, B8, B9, B10 |

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

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

The SHARC DSP implementation requires packing (`B0`) using the `PACKING()` command depending upon whether the 32-bit storage is set as PM or DM. A DM transfer is virtually 40 bits, so a single `B0` is required after `B4`, `B6` and `B10`, creating three DM words. For storage in memory of the type "PM" and width 32 bits, the packing command would have two null bytes per transfer.

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

To see an example of the `PACKING()` command in a linker description file, see the `fft_ovly.ldf` example file that comes with your VisualDSP++ software.

For ADSP-21xxx processors, the following types of packing transfers and corresponding `PACKING()` syntax apply (as in Table 2-8, null placeholders are omitted):

1. 48-bit to 32-bit Instruction Word Packing

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

2. 48-bit to 16-bit Instruction Word Packing (optional, not required because the byte ordering is sequential)

   ```
   PACKING(6 B1 B2 B3 B4 B5 B6)
   ```

3. 32-bit to 16-bit Data Word Packing (optional, not required because the byte ordering is sequential)

   ```
   PACKING(4 B1 B2 B3 B4)
   ```

4. ADSP-21161N DSP supports 8-bit packing

   a. 48-bit to 8-bit Word Packing

      ```
      PACKING(6 B1 B2 B3 B4 B5 B6)
      ```

   b. 32-bit to 8-bit Word Packing

      ```
      PACKING(4 B1 B2 B3 B4)
      ```

   c. 16-bit to 8-bit Word Packing

      ```
      PACKING(2 B1 B2)
      ```

Note that a byte indicated with `B0` in the `PACKING()` syntax acts as a place holder, and the linker sets those bytes to zero in the executable.

## Overlay Packing Formats

The PACKING() command has two uses. One is to pack data and instructions for overlays executed from external memory (by definition those overlays "live" in external memory) and the other is an explicit PACKING() command to be used whenever the width or byte order of stored data differs from its run-time organization. The linker word aligns the packing instruction as needed.

Table 2-9 indicates the packing format combinations for ADSP-21xxx DSP's DMA overlays possibly used under each of the two operations.

Table 2-9. Packing Formats for ADSP-21xxx DMA Overlays

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

Table 2-10 indicates the packing format combinations for ADSP-21161N DSP's DMA overlays for storage in 8-bit wide memory.

Table 2-10. Additional Packing Formats for ADSP-21161N DMA Overlays

| Execution Memory Type | Storage Memory Type | Packing Instruction |
|---|---|---|
| 48 Bit PM | 8 Bit[1] DM | PACKING(6 B0 B0 B0 B1 B0 B0 B0 B0 B2 B0 B0 B0 B0 B3 B0 B0 B0 B0 B4 B0 B0 B0 B0 B5 B0 B0 B0 B0 B6 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0) |
| 32 Bit DM | 8 Bit DM | PACKING(4 B0 B0 B0 B1 B0 B0 B0 B0 B2 B0 B0 B0 B0 B3 B0 B0 B0 B0 B4 B0) |
| 16 Bit DM | 8 Bit DM | PACKING(2 B0 B0 B0 B1 B0 B0 B0 B0 B2 B0) |

1 * 8-bit packing is available in ADSP-2106x DSPs and ADSP-21160 DSPS *only* during EPROM booting.

## External Execution Packing

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

Table 2-11 and Table 2-12 indicate the packing formats for packed external execution.

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

Table 2-11. External Execution Packing Formats for ADSP-21161N DSPs

| Memory Type | Packing Instruction |
|---|---|
| 48 bit in 32 Bit | PACKING(6 B1 B2 B3 B4 B0 B0 B0 B0 B5 B6 B0 B0) |
| 48 Bit in 16 Bit | PACKING(6 B0 B0 B1 B2 B0 B0 B0 B0 B3 B4 B0 B0<br>B0 B0 B5 B6 B0 B0 B0 B0 B0 B0 B0 B0) |
| 48 Bit in 8 Bit | PACKING(6 B0 B0 B0 B1 B0 B0 B0 B0 B0 B2 B0 B0<br>B0 B0 B0 B3 B0 B0 B0 B0 B0 B4 B0 B0 B0 B0 B0 B5 B0 B0<br>B0 B0 B0 B6 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0 B0) |

Table 2-12. External Execution Packing Formats for ADSP-21065L DSPs

| Memory Type | Packing Instruction |
|---|---|
| 48 Bit PM[1] | PACKING(6 B0 B0 B5 B6 B0 B0 B1 B2 B3 B4 B0 B0) |

[1] LDF memory command defines a “logical” segment, rather than a physical, which is 32 bits.

## Default Packing - No Reordering

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

## PLIT{}

The linker's PLIT{} command lets you add Procedure Linkage Table (PLIT) commands to your LDF. These commands provide a template from which the linker generates assembly code whenever a symbol resolves to a function in overlay memory. These instructions typically handle a call to a function in overlay memory by calling an overlay memory manager.

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

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

- 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.

- 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

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

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

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

- There is no Input Section associated with the `.plit` Output Section: the LDF 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 per PLIT syntax. 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 previously, 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.

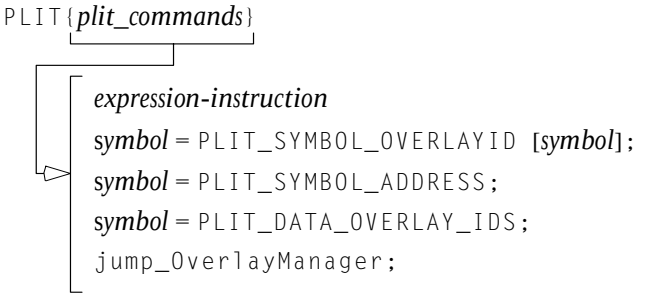


Figure 2-8. Syntax Tree of the PLIT{} Command

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.

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

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.

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

After the setup and variable identification are completed, the overlay itself must be brought (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{}, RESOLVE{}, 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 processors 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

The PROCESSOR{} command syntax is defined as follows:

- *processor_name*

  Assigns a *processor_name* to the processor. Processor names follow the same rules as any linker label. For more information, see "LDF Expressions and Conventions" on page 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. Use LDF command scoping to define these segments outside the PROCESSOR{} command. For more information, see "Command Scoping" on page 2-39, and "MEMORY{}" on page 2-54.

- PLIT{*plit_commands*}

  Defines Procedure Linkage Table (PLIT) commands that apply only to this processor. For more information, see "PLIT{}" on page 2-69.

- SECTIONS{*section_commands*}

  Defines sections for placement within the executable (.DXE). For more information, see “SECTIONS{}” on page 2-75.

## RESOLVE()

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

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

Use the RESOLVE() command, which directs the linker to ignore a LINK_AGAINST() for a specific symbol, to override the search order for a specific variable or label. For more information, see Listing 2-9 on page 2-89.

## SEARCH_DIR()

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

For example,

```
ARCHITECTURE(ADSP-21062)
// 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 21k/lib relative to the install directory

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

## SECTIONS{}

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

The LDF 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.

The SECTIONS{} command syntax is defined 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. Although your LDF may contain only one SECTIONS{} command within each LDF scope, there is no limit to the number of output sections that you can declare within each SECTIONS{} command. For more information, see "Command Scoping" on page 2-39.

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


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

- *section_name* starts with a letter, underscore, or period and may include any letters, underscores, digits, and points. A *section_name* must not conflict with any linker keywords. The special *section_name*, `.plit`, indicates the Procedure Linkage Table (PLIT) section that the linker generates when resolving

symbols in overlay memory. You must place this section in non-overlay memory to manage references to items in overlay memory.

- *section_type* is optional and is used to assign 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.
- *input_labels* come from the run-time `.SECTION` labels in your assembly program. Use commas to delimit `.SECTION` names within the list.

- *`expression`*

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

- `FILL(`*`hex number`*`)`

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

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

- `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 2-69.

[*] In that section only.

- `OVERLAY_INPUT(`*`overlay_commands`*`)`

  In a *section_command*, identifies parts of your program to be placed in an overlay executable with *overlay_commands*. For more information, see “Linking for Overlay Memory” on page 2-95.

  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 `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 .plit section in the overlay memory. For more information, refer to page 2-77.
- 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.
  - 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.

  When you use the FIRST_FIT or BEST_FIT algorithm, the overlay ID of the next overlay is:

  ```
  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

The SHARED_MEMORY{} command syntax is defined 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 2-75.

An example of this command scoping appears in Figure 2-12 on page 2-83. For more information, see "Command Scoping" on page 2-39.

* For SHARC 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.

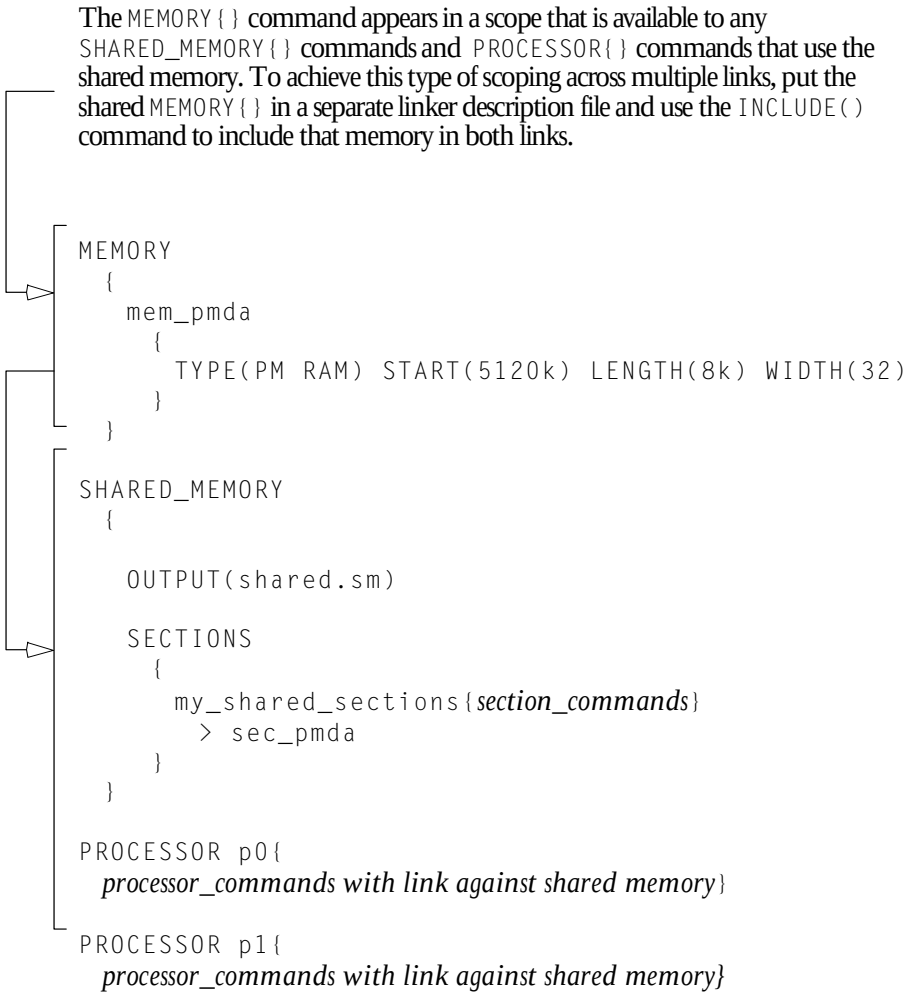


Figure 2-12. LDF Scoping for the SHARED_MEMORY{} Command

# 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 Command Summary" on page 2-49. The examples in this section include the following:


- "Linking for Single-Processor Memory" on page 2-85
- "Linking Large Uninitialized Variables" on page 2-87
- "Linking for Multi-Processor and Shared Memory" on page 2-88
- "Linking for Overlay Memory" on page 2-95
- "Using a Procedure Linkage Table" on page 2-98
- "Managing Overlays" on page 2-105
- "Managing Two Overlays" on page 2-111
- "Reducing Overlay Manager Overhead" on page 2-118


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 ADSP-21xxx DSPs, these examples are in the directory:

```
<VisualDSP++ InstallPath>\21xxx\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 ADSP-21xxx DSPs, these default LDFs are in the directory:

```
<VisualDSP++ InstallPath>\21xxx\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
- `$OBJS` and `$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-21062

ARCHITECTURE(ADSP-21062)

// 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 21k/lib relative to the install directory

SEARCH_DIR( $ADI_DSP\21k\lib )

// Lib060.dlb is a 2106x specific archive library and it
```

```
// must precede libc.dlb, the "C" archive library in order
// to link in 2106X specific routines

$LIBS = lib060.dlb, libc.dlb;

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

//    060_hdr.doj is the standard initialization file
//    for 2106X
$OBJS = $COMMAND_LINE_OBJECTS, 060_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( "21062_memory.h" )
      // For more information, see Listing 2-2 on page 2-19.
      }
      // Specify the output sections
   SECTIONS {
      INCLUDE( "21062_sections.h" )
      // For more information, 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 executable by using the `SHT_NOBITS` section qualifier (Section Header Type No Bits). This is a useful technique, especially for ADSP-21065L users who work 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 loader output 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

```
.SEGMENT/DM     sdram_area;   /* 1Mx32 SDRAM */
.VAR huge_bufffer[0x100000];
.ENDSEG;
```

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

```
ARCHITECTURE(ADSP-21061)

// Libraries & objects from the command line

$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY {
   mem_sdram {
      TYPE(DM RAM) START(0x3000000) END(0x30fffff) WIDTH(32)
      } // end segment
   } // end memory
PROCESSOR P0 {
   LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
   // SHT_NOBITS section isn't written to the output file
   SECTION {
      sdram_ouput SHT_NOBITS {
         INPUT_SECTIONS( $OBJECTS ( sdram_area ) )
         } >mem_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, and each processor's memory; it also places code for each processor. The sample 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
- `$OBJS` and `$LIBS` macros get object (`.DOJ`) and library (`.DLB`) file input

- MPMEMORY{} command defines each processor's offset within multiprocessor 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 place program sections for each processor's output file, using the memory definitions
- LINK_AGAINST() commands resolve symbols within multiprocessor memory

Listing 2-9. Multiprocessor System LDF Example

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )
//    Multiprocessor memory space is represented in the
//     LDF via the MPMEMORY command. The values represent
//    an “offset” that the linker will use when
//    it resolves undefined symbols in one DXE to symbols
//    defined in another DXE. That is, the offset is added
//    to the defined symbols value.
//
// For example,
//    PROCESSOR project PSH0 references 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 + 0x280000       =
//    0x2a2000
```

```
MPMEMORY {
    PSH0 { START (0x200000) }
    PSH1 { START (0x280000) }
}
MEMORY {
    //This memory description is to be used for all
    //processors
    //Alternatively, a PROCESSOR could describe its
    //own MEMORY
    mem_rth {
      TYPE(PM RAM) START(0x20000) END(0x200ff) WIDTH(48) }
    mem_init {
      TYPE(PM RAM) START(0x20100) END(0x201ff) WIDTH(48) }
    mem_pmco {
      TYPE(PM RAM) START(0x20200) END(0x249ff) WIDTH(48) }
    mem_pmda {
      TYPE(DM RAM) START(0x24a00) END(0x24fff) WIDTH(40) }
    mem_dmda {
      TYPE(DM RAM) START(0x28000) END(0x2bfff) WIDTH(40) }
    mem_heap {
      TYPE(DM RAM) START(0x2e000) END(0x2efff) WIDTH(32) }
    mem_stak {
      TYPE(DM RAM) START(0x2f000) END(0x2ffff) WIDTH(32) }
}

$LIBRARIES = lib060.dlb, libc.dlb;

//
//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 “seg_dmda”

SECTIONS {
   .sec_dmda {INPUT_SECTIONS( $SHARED_OBJECTS(seg_dmda))
      } > mem_dmda
    }
}

//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, 060_hdr.doj;
LINK_AGAINST(shared.sm)

OUTPUT( psh0.dxe )

SECTIONS {
   dxe_pmco {INPUT_SECTIONS(
      $PSH0_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
      } >mem_pmco
    dxe_pmda {INPUT_SECTIONS(
      $PSH0_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
      } >mem_pmda
    dxe_dmda {INPUT_SECTIONS(
      $PSH0_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
      } >mem_dmda
    dxe_init {INPUT_SECTIONS(
      $PSH0_OBJECTS(seg_init) $LIBRARIES(seg_init))
      } >mem_init
   dxe_rth {INPUT_SECTIONS(
      $PSH0_OBJECTS(seg_rth) $LIBRARIES(seg_rth))
      } >mem_rth
   stackseg {
      // allocate a stack for the application
      ldf_stack_space = .;
      ldf_stack_length = 0x2000;
      } > mem_stak
   heap {
```

```
      // allocate a heap for the application
      ldf_heap_space = .;
      ldf_heap_end = ldf_heap_space + 0x2000;
      ldf_heap_length = ldf_heap_end - ldf_heap_space;
      } > mem_heap
   }
}
//
//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, 060_hdr.doj;
   LINK_AGAINST(shared.sm, psh0.dxe)
   OUTPUT(psh1.dxe)
   SECTIONS {
       dxe_pmco {INPUT_SECTIONS(
         $PSH1_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
         } >mem_pmco
       dxe_pmda {INPUT_SECTIONS(
         $PSH1_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
         } >mem_pmda
       dxe_dmda {INPUT_SECTIONS(
         $PSH1_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
         } >mem_dmda
       dxe_init {INPUT_SECTIONS(
         $PSH1_OBJECTS(seg_init) $LIBRARIES(seg_init))
         } >mem_init
       dxe_rth {INPUT_SECTIONS(
         $PSH1_OBJECTS(seg_rth) $LIBRARIES(seg_rth))
         } >mem_rth
      stackseg {
         // allocate a stack for the application
         ldf_stack_space = .;
         ldf_stack_length = 0x2000;
         } > mem_stak
      heap {
         // allocate a heap for the application
         ldf_heap_space = .;
         ldf_heap_end = ldf_heap_space + 0x2000;
         ldf_heap_length = ldf_heap_end - ldf_heap_space;
```

```
        } > mem_heap

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

## Using Reflective Semaphores

Semaphores can be used in multiprocessor (MP) systems to allow the processors to share resources such as memory or I/O. A semaphore is a flag that can be read and written by any of the processors sharing the resource. The value of the semaphore tells the processor when it can access the resource. *Reflective semaphore* is a way to communicate among processors which share a multiprocessor memory space.

Broadcast writes can be used to implement reflective semaphores in a multiprocessing system. Broadcast writes allow simultaneous transmission of data to all of the ADSP-21xxx DSPs in a multiprocessing system. The master processor can perform broadcast writes to the same memory location or IOP register on all of the slaves. During broadcast writes, the master also writes to itself unless the broadcast is a DMA write. Broadcast writes can be used to implement reflective semaphores in a multiprocessing system. Broadcast writes can also be used to simultaneously download code or data to multiple processors.

Bus lock can be used in combination with broadcast writes to implement reflective semaphores in a multiprocessing system. The reflective sema-

phore should be located at the same address in internal memory (or IOP register) of each ADSP-21xxx DSP.

The ADSP-21xxx SHARC DSPs have a “broadcast” space. You may use LDF (or HH file) to define a memory segment in this space, just as in internal or any processor MP space. The broadcast space aliases internal space, so if there is a memory segment defined in the broadcast space, LDF cannot have a segment at the corresponding address in the internal space, or in the MP space of any processor. Otherwise, the linker will generate an error indicating that the memory definition is not valid.

To check the semaphore, each ADSP-21xxx DSP simply reads from its own internal memory. Any object in the project could be mapped to an appropriate segment defined in the broadcast space for use as a “reflective semaphore”. If an object defining symbol `SemA` is mapped to a broadcast segment, then when the program writes to `SemA`, the written value appears at the aliased internal address of each processor in the cluster. Each processor may read the value using `SemA`, or from internal memory by selecting (`SemA-0x380000`), thus avoiding bus traffic.

To modify the semaphore, an ADSP-21xxx DSP requests bus lock and then performs a broadcast write to the semaphore address (i.e., `SemA`) on every DSP, including itself. Before modifying the semaphore, though, the DSP should read it to verify that another processor has not changed it.

For more information on semaphores, refer to the *Hardware Reference* manual for appropriate DSP model.

## 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-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )

MAP(overlay.map)

//
//This simple overlay example will use internal memory as
//overlay
//memory (overlays would never "live" in internal memory)
//

MEMORY {
    mem_rth {
        TYPE(PM RAM) START(0x20000) END(0x200ff) WIDTH(48) }
    mem_init {
        TYPE(PM RAM) START(0x20100) END(0x201ff) WIDTH(48) }
    mem_pmco {
        TYPE(PM RAM) START(0x20200) END(0x20723) WIDTH(48) }
    mem_ovly {
        TYPE(PM RAM) START(0x20724) END(0x23fff) WIDTH(48) }
    mem_pmda {
        TYPE(PM RAM) START(0x26000) END(0x27fff) WIDTH(32) }
    mem_dmda {
        TYPE(DM RAM) START(0x2a000) END(0x2bfff) WIDTH(32) }
    mem_heap {
        TYPE(DM RAM) START(0x2e000) END(0x2efff) WIDTH(32) }
    mem_stak {
        TYPE(DM RAM) START(0x2f000) END(0x2ffff) WIDTH(32) }
}
// mem_pmco is the "run" memory segment
```

```
// mem_ovly is the “live” memory segment
//
//Processor and application specific assembly language
//instructions. One instance of these instructions is
//generated for each symbol that is resolved in overlay memory.

PLIT {   //Each of five instructions are duplicated for each
          //symbol in an overlay
    R0 = PLIT_SYMBOL_OVERLAYID;
    // Assign the overlay ID of the resolved symbol to R0

    R1 = PLIT_SYMBOL_ADDRESS;
    // Assign the “run” address of the resolved symbol to R1

    dm(_overlayID) = R0;
    dm(_pf) = R1;
    JUMP _OverlayManager;
}

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

PROCESSOR P0 {
    $P0_OBJECTS = main.doj , manager.doj;
    OUTPUT(mgrovly.dxe)
    SECTIONS {
        // .text output section
seg_pmco {
    INPUT_SECTIONS(
        $P0_OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))

//Specify the first overlay. This overlay will
//“live” in the memory defined by “mem_ovly”. It
//will run in the memory space defined by “mem_pmco”

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

    OVERLAY_OUTPUT( overlay1.ovl )

    // Only take the code from the file overlay1.doj.
    // If there is data that this code needs it must be
    // the INPUT of a data overlay or the INPUT to
```

```
        // non-overlay data memory.

        INPUT_SECTIONS(overlay1.doj( seg_pmco))

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

        ALGORITHM( ALL_FIT )
        SIZE(0x100) // max ovlay size

    } > mem_ovly

        // This is the second overlay. Not that these
        // OVERLAY_INPUT command must be contiguous in the LDF
        // file in order for them to occupy the same
        // "run-time" memory.

        OVERLAY_INPUT {
            OVERLAY_OUTPUT( overlay2.ovl )
            INPUT_SECTIONS(overlay2.doj( seg_pmco))
            ALGORITHM( ALL_FIT )
            SIZE( 0x100)
        } > mem_ovly

    } > dxe_pmco

        // 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 {
            } > mem_pmco

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

        dxe_dmda {
            INPUT_SECTIONS(
                $PO_OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
            } > mem_dmda
```

```
        seg_init {
        INPUT_SECTIONS(
            $PO_OBJECTS(seg_init) $LIBRARIES(seg_init))
     } > mem_init

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

    stackseg {
        // allocate a stack for the application
        ldf_stack_space = .;
        ldf_stack_length = 0x2000;
    } > mem_stak

    heap {
        // allocate a heap for the application
        ldf_heap_space = .;
        ldf_heap_end = ldf_heap_space + 0x2000;
        ldf_heap_length = ldf_heap_end - ldf_heap_space;
      } > mem_heap
    }
}
```

## Using a Procedure Linkage Table

The linker resolves function calls and variable accesses, 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 generated 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 2-69.

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 2-99 shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker

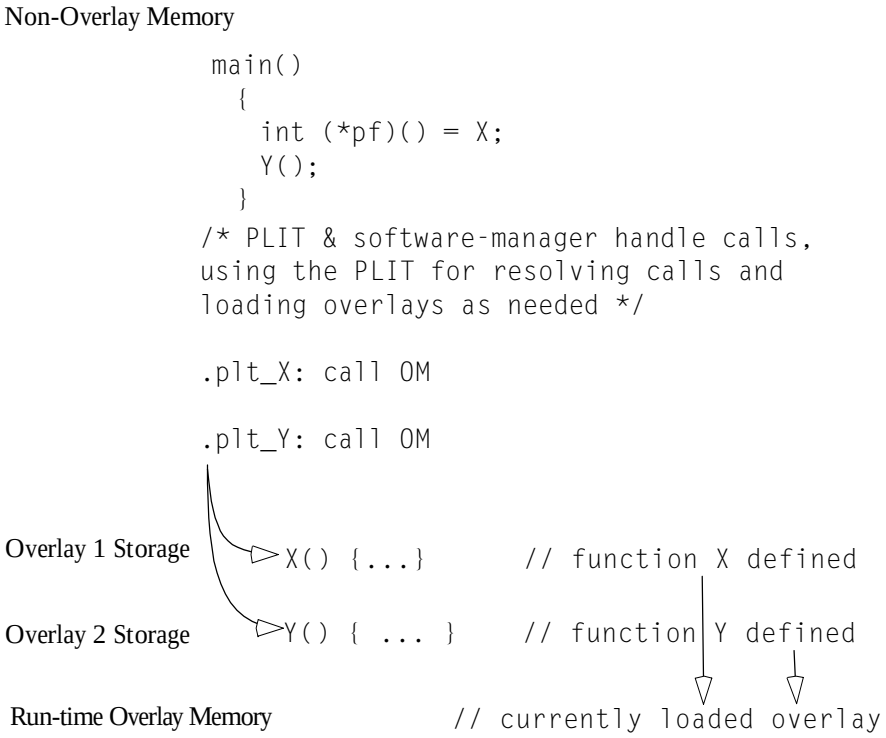


Figure 2-13. PLITs & Overlay Memory; main() Calls to Overlays

generates some special symbols for overlays. These overlay symbols include:

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

The # in these symbols indicates the overlay number.

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. However, you can force the linker to output a PLIT, even when all references can be resolved locally.

PLITs let you resolve inter-overlay calls, as shown in Figure 2-14 on page 2-101. 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 2-102. 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 overlays. One way to optimize overlays is to not put global variables in overlays. This avoids the difficulty of making sure the proper overlay is loaded before a global gets called.

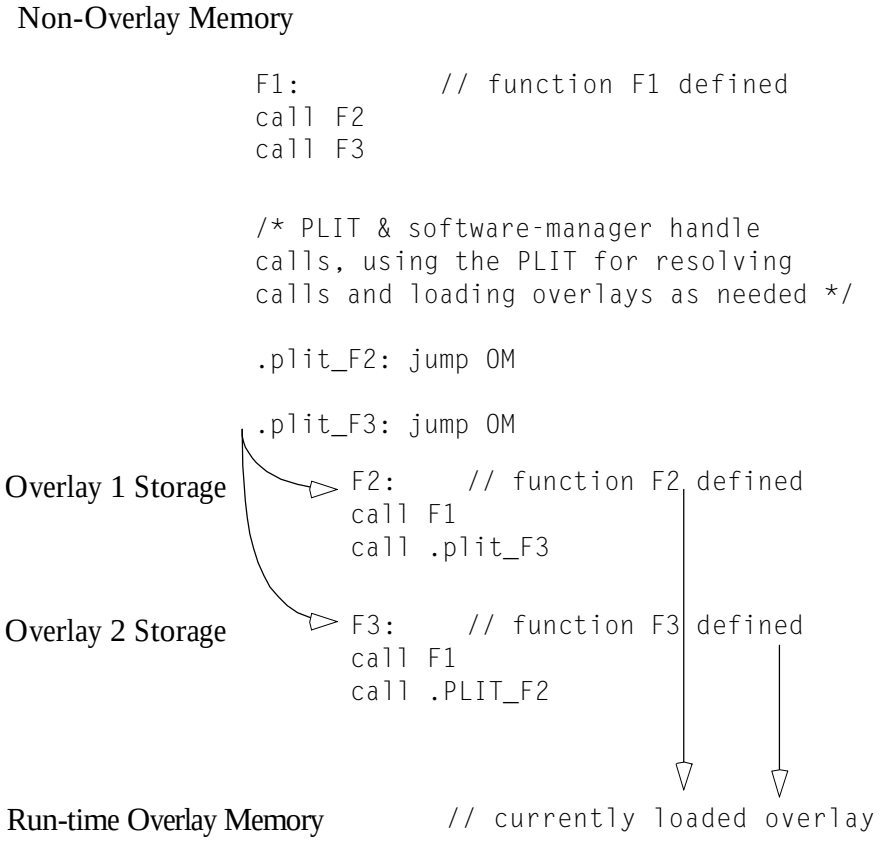


Figure 2-14. PLITs & Overlay Memory; Inter-Overlay Calls

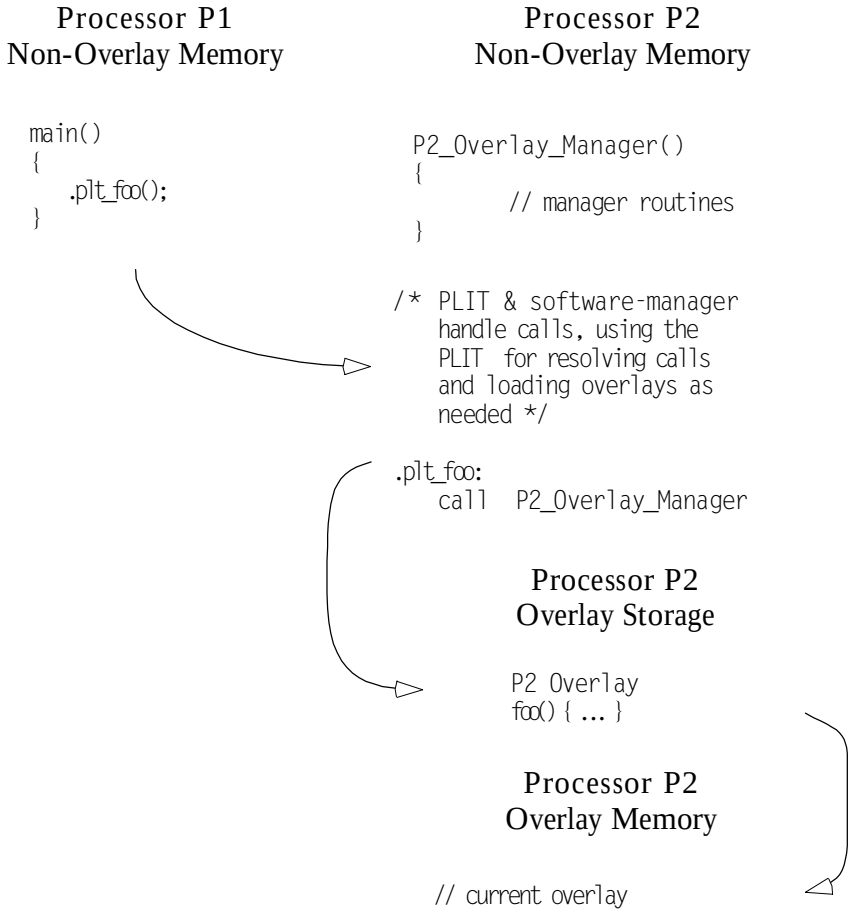


Figure 2-15. PLITs & Overlay Memory; Inter-Processor Calls

## 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 section discusses the concept of memory overlays and how they are used with Analog Devices 32-bit SHARC DSPs. The following topics and examples are discussed:


- "Managing Overlays" on page 2-105
- "Managing Two Overlays" on page 2-111
- "Reducing Overlay Manager Overhead" on page 2-118


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

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" (run-time) 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

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

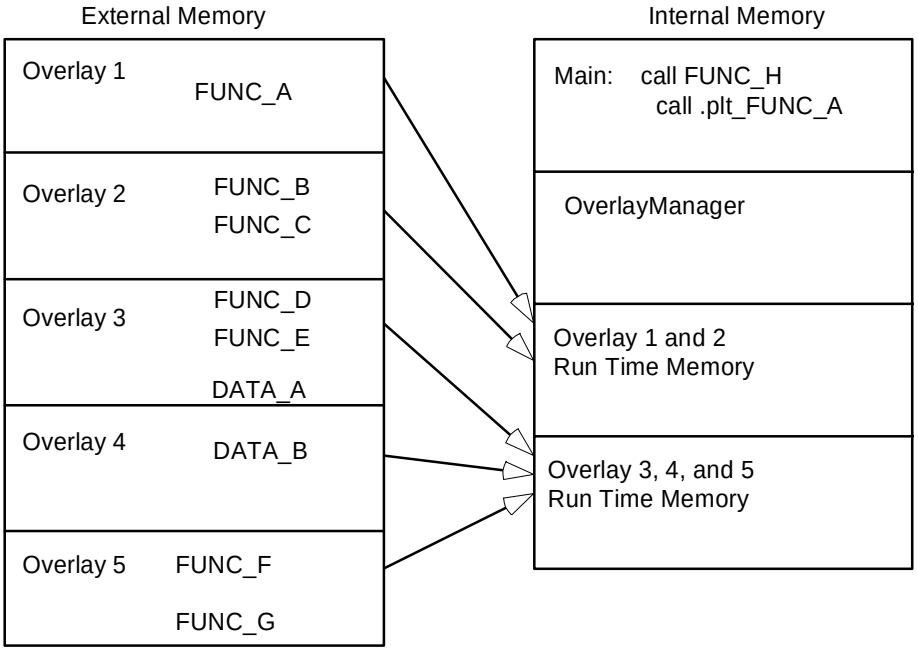


Figure 2-16. Memory Overlays

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. If `FUNC_B` is required, the overlay manager loads Overlay 2 in the location within internal memory where Overlay 2 is designated to run. If `FUNC_D` is required, the overlay manager loads Overlay 3 into its designated run-time memory.

The overlay manager is a user-defined function responsible for insuring that a required symbol (function or data) within an overlay is in the run-

time memory when it is needed. The transfer occurs using the direct memory access (DMA) capability of the SHARC processor.

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

## 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 in the LDF. 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

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

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

The 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 `ovl_pmco`.
- `OVLY_two` contains functions `FUNC_B` and `FUNC_C`. It also lives in memory segment `ovl_pmco`.

The common run-time location shared by overlays `OVL_one` and `OVL_two` is within the memory segment `sec_pmco`.

The LDF tells the linker 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_size_run_N
_ov_word_size_live_N
_ov_runtimestartaddress_N
```

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

The overlay live and run word sizes are different if the internal memory and external memory widths are different. For example, the instruction word width of the SHARC DSP is 48-bits. A system containing 32-bit wide external memory requires data packing to store an overlay containing instructions. The overlay live word size (number of words in the overlay) is based on the number of 32-bit words required to pack all of the 48-bit instructions.

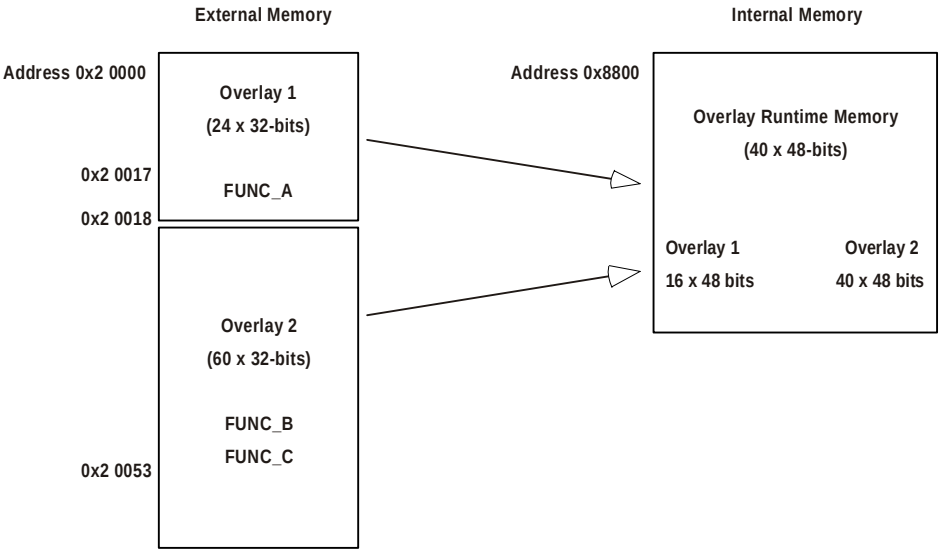


Figure 2-17. Example Overlay Run and Live Sizes

Figure 2-17 shows the difference between overlay live and run size.

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

The following code shows the value of all constants generated by the linker for the example in Figure 2-17:

Listing 2-13. Linker Generated Constants

```
_ov_startaddress_1 = 0x20000      _ov_startaddress_1 = 0x20000
_ov_endaddress_1 = 0x20017         _ov_endaddress_1 = 0x20017
_ov_word_size_run_1 = 0x118       _ov_word_size_run_1 = 0x118
_ov_word_live_run_1 = 0x10        _ov_word_size_live_1 = 0x10
_ov_run-timestartaddress_1 = 0x8800
_ov_run-timestartaddress_1 = 0x8800
```

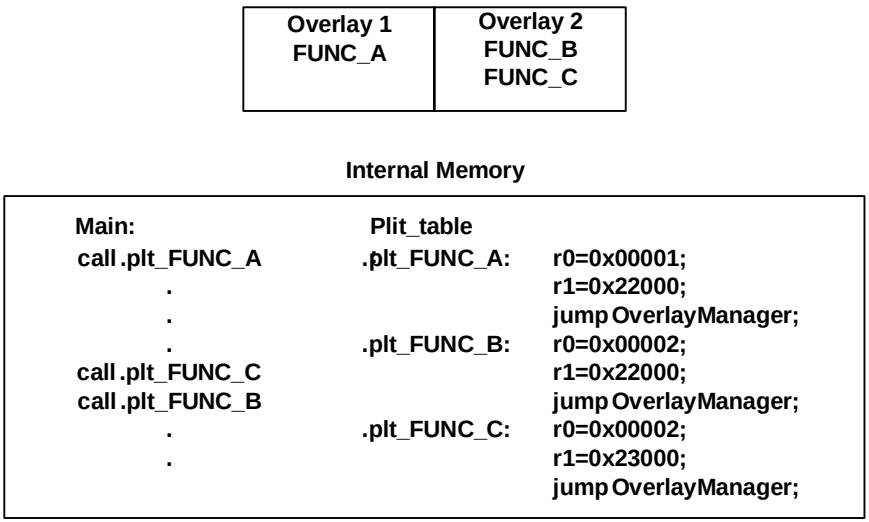


Figure 2-18. Expanded PLIT Table.

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 and then jumps to the overlay manager.The linker replaces an overlay symbol reference (function call) with a jump to a location in the PLIT.

PLIT code is defined within the linker description file (LDF) by the programmer. This code prepares the overlay manager to handle the overlay containing the referenced symbol. The code generally initializes registers to contain the overlay ID and the referenced symbols run time address.

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

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

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

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

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 2-14 is an example PLIT definition from an LDF where the register R0 is set to the value of the overlay ID that contains the referenced symbol, and register R1 is set to the run time address of the referenced symbol. (`PLIT_SYMBOL_OVERLAY_ID` and `PLIT_SYMBOL_ADDRESS` are linker key words). The last instruction branches to the overlay manager that uses the initialized registers to determine which overlay to load, and where to jump to execute the overlay function called.

Listing 2-14. PLIT Definition in LDF

```
PLIT
{
   R0 = PLIT_SYMBOL_OVERLAY_ID;
   R1 = 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 2-18. 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 2-18 is for the overlay symbol `FUNC_A`. The linker replaces the constant name `PLIT_SYMBOL_OVERLAYID` with the ID of the overlay containing `FUNC_A`. The linker also replaces the constant name `PLIT_SYMBOL_ADDRESS` with the run time address of `FUNC_A`.

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

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 run-time space as a result of a previous reference.

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

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

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

### 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 the sample overlay manager source code, see the examples that come with the development software. In the following 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

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

Listing 2-15. FFT Overlay Example

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

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

The first and the last functions called are in overlay `fft_one`. The two middle functions called are in overlay `fft_two`. When the first function, `fft_one`, is referenced during code execution, `overlay id=1` is transferred to internal memory. When the second function, `fft_two`, is referenced, `overlay id=2` is transferred to internal memory. Since the third function is in the 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, overlay

`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 (db);
    call fft_middle_stages (db);
    call fft_next_to_last (db);
    call fft_last_stage (db);

wait:       idle;
             jump wait;
```

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

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

Register `R0` contains the overlay ID that occupies the referenced symbol and register `R1` 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 `R1`.

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:

- The size of the overlays, using both run time word sizes and live word sizes
- The starting address of the live space
- The starting address of the run space

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

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

Before preparing the 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.

The following segment of the overlay manager function creates the run time stack, stores the overlay ID in a modify register, and checks the overlay ID stored in `ov_id_loaded`:

```
/*  _overlayID has been defined as R0. R0 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.   */

dm(ov_stack)=i8;
dm(ov_stack+1)=m8;
dm(ov_stack+2)=l8;
dm(ov_stack+3)=r2;

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

r2=dm(ov_id_loaded);
r0=r0-r2;
if EQ jump continue;
dm(ov_id_loaded)=m8;

r0=i0;       dm(ov_stack+4)=r0;
r0=m0;       dm(ov_stack+5)=r0;
r0=l0;       dm(ov_stack+6)=r0;
```

The overlay manager uses the value of the linker generated constants to set up the DMA transfer as shown in the following code segment of the overlay manager function. The constants are in arrays as previously described.

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

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

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

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

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

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

/* Set DMA external modifier */
r0=1;
dm(EMEP0)=r0;

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

/* Set DMA internal modify to 1 */
dm(IMEP0)=r0;

/* Set DMA internal count to Overlay run size.   */
r0=dm(m0,i0);
dm(CEP0)=r0;
```

```
/* Set DMA external count to Overlay live size.   */
r0=pm(m8,i8);
dm(ECEP0)=r0;

/* DMA enabled, instruction word, Master, 48-32 packing */
r0=0x2e1;
dm(DMAC0)=r0;
```

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:

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

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

## Reducing Overlay Manager Overhead

The 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 overlay manager source code, see the examples that come with the development software.

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

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

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

Since the linker handles all overlay symbol references in the same way (jump to PLIT table then overlay manager), it is up to the overlay manager to distinguish between a symbol reference requesting the load of an 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 immediately following 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 `R0` is set to the value that is placed in the flag buffer, and the value in `R0` is stored in memory; 1 indicates a load and 0 indicates an execution call. At each overlay function call the load buffer must be updated.

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 (db);
          r0=0;
          dm(prefetch) = r0;
call fft_middle_stages (db);
          r0=0;
          dm(prefetch) = r0;
call fft_next_to_last (db);
          r0=0;
          dm(prefetch) = r0;
call fft_last_stage (db);
          r0=0;
          dm(prefetch) = r0;
```

The next set of instructions represents a load function call.

```
call fft_middle_stages (db);
          /* This function call pre loads */
          /* the function into the overlay run memory. */
r0=1;
     dm(prefetch) = r0;
          /* Set prefetch flag to 1 to indicate a load. */
```

The code executes the first function and transfers the second function and so on. Therefore, 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(seg_pmco) )
     PACKING( 12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
} >ovl_pmco  // Overlay to live in section ovl_pmco

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

INPUT_SECTIONS(ovly_mgr.doj(pm_code))

OVERLAY_INPUT
{
       ALGORITHM(ALL_FIT)
       OVERLAY_OUTPUT(fft_two.ovl)
       INPUT_SECTIONS(  Fft_ovl.doj(seg_pmco3) )
     PACKING( 12 B1 B2 B3 B4 B0 B11 B12 B5 B6 B0 B7 B8 B9 B10 B0)
```

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

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.

The overlay manager requires modification from that of Example 1. 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 for the elimination of some instructions. For instance, if your application allows for the free use of registers, you may not need a run-time stack.

# Linker Glossary

**LDF commands** — Commands in the Linker Description File that define the target system and order the processing of linker output for that system.

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

**LDF macros** — Built-in macros (text strings to be executed), with pre-defined procedures or values, which may be system-specific. LDF macros are available globally regardless of the scope where the macro is defined.

**LDF Memory Segment** — parts of the DSP/Chip 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 scope** — The two types of LDF file scopes are *global* and *command*. A command scope defines the content for an `OUTPUT()` command that can be used within a `PROCESSOR{}` or `SHARED_MEMORY{}` command; the effects of commands and expressions that appear in the command scope are limited to those scopes. The global scope occurs outside of commands and defines commands and expressions to be available in the global scope and visible in all subsequent scopes.

**LDF Sections Versus Segments** — the linker takes Input Sections as inputs, places them in a Output Section, and maps Output Sections into selected Memory Segments. The line:

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

directs the linker to take the `isr_tbl` *Input Section*, place it in the `dxe_isr` *Output Section*, and map it to the `mem_isr` *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 your program in memory.

**Live Memory** — The off-chip area of memory where an overlay lives (is stored) when it is not running.

**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.

**Run-time Memory** — The on-chip area where an overlay (or a group of overlays) runs.

**Target** — The type of processor chip on which a project is intended to be run.