

2 LINKER

Overview

You begin code generation by creating source code files in C/C++ language or assembly language. A module is a unit of assembly language comprising a main program, subroutine, or data variable declarations. C programmers write C/C++ language files and use C/C++ compiler to create assembly code modules from them. Assembly language programmers write assembly code modules directly. Each code module is assembled separately by the assembler.

The VisualDSP++ linker, `linker.exe`, *links* several modules (object and library files) to produce executable (`.EXE`) files, shared memory (`.SM`) files, and overlay (`.OVL`) files, which can be loaded onto the target. It maps code into memory, resolves references among the files and data, and also produces map files and other output.

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

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

It accomplishes these tasks according to settings that you can choose, under the direction of a Linker Description File, which is described below.

Chapter Outline

This chapter contains the following information on the linker:

- [“Mapping Files to Memory: the Linker Description File” 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 2-14](#) — describes how to use the linker to produce executable, shared memory and overlay files.
- [“Linker Command-Line Reference” on page 2-28](#) — lists linker command line switches and their syntax.
- [“Linker Description File Reference” on page 2-41](#) — describes Linker Description File (LDF) syntax and programming techniques.
- [“LDF Programming Examples” on page 2-88](#) — provides a series of programming examples for different types of systems
- [“Linker Glossary” on page 2-123](#) — provides definitions of linker-related terms

Mapping Files to Memory: the Linker Description File

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

All projects must locate their code and data in the DSP's memory (internal or external) to execute. Therefore, they must all specify the linking process with an LDF, even if you don't include one. See [“Default LDF and Object Code Placement” on page 2-3](#), for a discussion of what happens if there is no LDF in your project.

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

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

For an introduction to the LDF commands, and how to construct an LDF, refer to [“Linker Guide” on page 2-14](#).

Default LDF and Object Code Placement

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

Mapping Files to Memory: the Linker Description File

when you set your project's options, and locates the program (and data) portions of your object code according to the `-Dprocessor` command line switch, where *processor* is your target architecture.

The default compiler section names are `program` (for code), `data1` (for DM data), and `data2` (for PM data).

For more information on LDF syntax, see [“Linker Command-Line Reference” on page 2-28](#). For sample LDFs and related source files, see the samples included with your VisualDSP++ 2.0 software.

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++ 2.0 development software gives an object file a `.DOJ` extension.

Code in object files produced by the compiler and assembler is partitioned into various sections, called *Input Sections*. Each Input Section holds a particular type of compiled/assembled source code. For example, an Input Section could hold 24-bit wide program instructions or data such as variables (of various widths). Some of these Input Sections also can contain debug information.

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

Input Section Directives in Assembly Code

An assembly source file must describe how code and data are mapped into the memory on your target DSP. There are two types of memory: data memory, which typically contains data and memory-mapped ports, and program memory, which typically contains code (and can also store data). The way you structure your code and data into memory should follow from the memory architecture of the target DSP.

The mapping of code and data is accomplished using the `.SECTION` directive (formerly `.DMSEG` and `.PMSEG`). The `.SECTION` directive defines groupings of instructions and data that are set as contiguous memory addresses in the DSP. Each `.SECTION` name corresponds to an input section name in the Linker Description File (`.LDF`). Section directives are obligatory in assembly code. Each code and data object must be placed explicitly or assembly will fail.

In an assembly source, code that goes into a particular Input Section appears directly after a `.SECTION <name>` directive in the source file.

[Listing 2-1](#) shows an assembly code sample for ADSP-219x DSP found in `$ADI_DSP\219x\crt1\dsp1ib\abs.asm`:

Listing 2-1. Sample Assembly Code Section Directive: the ABS Function (for ADSP-219x DSP)

```
#include      "lib_glob.h"
.SECTION/PM   Library_Code_Space;      /* Section Directive */
.global _abs;
_abs:        rdarg(AX1,1);               /* Read arg from stack */
            RTS (DB);
            AR = ABS AX1;               /* Take absolute value of input */
            AX1=AR;
```

In this example, the VisualDSP++ assembler labels the global variable `_abs` as belonging to Input Section `Library_Code_Space`, as it processes this file into object code.

Mapping Files to Memory: the Linker Description File

Input Section Directives in C Source Files

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

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

As the VisualDSP++ compiler processes this source, creating a corresponding object file, it responds to these lines as follows:

- It defines an Input Section labeled `ext_data`, and puts the variable `temp` there.
- It defines another Input Section named `extern`, and stores the code generated from `func1` there.

In C source files, the `section("section_name")` directive is optional. Repeating and extending the previous code fragment, we add a function `func2`, without any Input Section directive.

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

If your code does not specify an Input Section name, the compiler uses a set of default names. In this case, the compiler puts the code from `func2` into an Input Section for program code, named `program`. The linker uses the Input Sections' contents to make executable files. The Input Section names are the labels the linker uses to find the Input Sections within object files. For information on the compiler's default Input Section names, see the *Compiler & Library Manual* for your target DSP.



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

Input Section and Output Section names occupy different namespaces. Therefore they are independent, and may be replicated in an LDF.



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

The Linking Process and the LDF

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

- Your source code specifies one or more Input Sections as destinations for its compiled/assembled object(s).
- The compiler and assembler produce object code, with labels directing which portion(s) are allocated to which Input Sections. More than one code item may be placed in the same Input Section, but not vice versa.

Mapping Files to Memory: the Linker Description File

- The linker maps each Input Section in the object code to an Output Section, as directed by the LDF. Again, more than one Input Section can be placed in any Output Section.
- The linker maps each Output Section into a *Memory Segment*, a contiguous range of memory on the target*, as specified by the LDF. More than one Output Section may map to a single Memory Segment.

[Figure 2-1 on page 2-9](#) shows how the LDF combines information, directing the linker to place program sections in an executable according to the memory available in the DSP system. The LDF describes the target system and maps your program's 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 files to create an executable program.

[“Sample LDF” on page 2-10](#) shows a part of an LDF and provides notes to describe major LDF commands, their functionality and use.

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

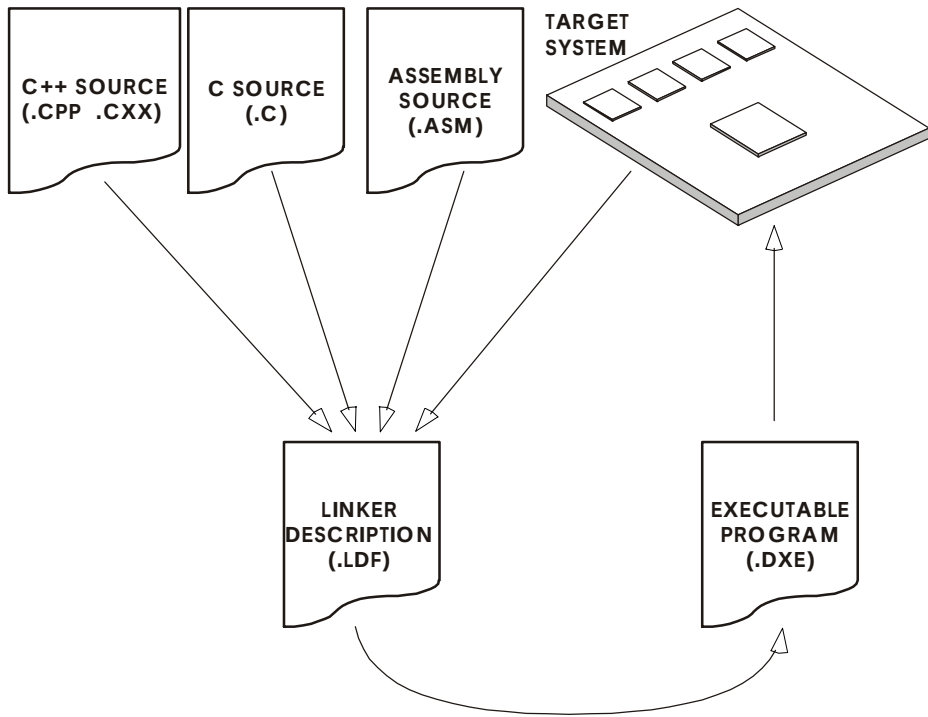


Figure 2-1. The LDF and the Linking Process

Mapping Files to Memory: the Linker Description File

Sample LDF

Listing 2-1 shows a sample LDF .

Linker Description File

```
ARCHITECTURE(ADSP-219x)          /* See Note 1 on page 2-11 */
SEARCH_DIR($ADI_DSP\219x\lib)    /* See Note 2 on page 2-11 */
$OBJECTS=$COMMAND_LINE_OBJECTS; /* See Note on page 2-11 */

MEMORY /* Define/label system memory See Note on page 2-12 */
{
    /* List of global Memory Segments */
    seg_rth {TYPE(PM RAM) START(0x00000) END(0x000241) WIDTH(24)}
    seg_code {TYPE(PM RAM) START(0x000242) END(0x00ffff) WIDTH(24)}
    seg_data1 {TYPE(DM RAM) START(0x10000) END(0x017fff) WIDTH(16)}
    seg_data {TYPE(DM RAM) START(0x018000) END(0x1ebff) WIDTH(24)}
    seg_heap {TYPE(DM RAM) START(0x0iec00) END(0x01efff) WIDTH(16)}
    seg_stack {TYPE(DM RAM) START(0x0if000) END(0x0iffff) WIDTH(16)}
}

PROCESSOR p0 /* The first (only?) processor in the system */
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT($COMMAND_LINE_OUTPUT_FILE) /* See Note 5 on page 2-12 */
    SECTIONS /* See Note 6 on page 2-13 */
    {
        /* List of sections for processor P0 */
        dxr_rth {INPUT_SECTIONS( $OBJECTS(seg_rth))} >seg_rth
        dxr_code {INPUT_SECTIONS( $OBJECTS(seg_code))} >seg_code
        dxr_code2 {INPUT_SECTIONS( $OBJECTS(y_input))} >seg_code
        dxr_data1 {INPUT_SECTIONS( $OBJECTS(seg_data1))} >seg_data1
        dxr_data {INPUT_SECTIONS( $OBJECTS(seg_data))} >seg_data
    }
}
```

As noted at the beginning of this chapter, you can run the linker from the VisualDSP++ IDDE or from the command line. Therefore, all VisualDSP++ IDDE linker options have equivalent command-line switches.

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

Notes on Sample LDF:

These notes describe the boldface lines 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 (x) must be installed in VisualDSP++.
2. **SEARCH_DIR** specifies path name(s) to search for libraries and object files. This example shows one search directory, the single argument to the **SEARCH_DIR** command.

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

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

3. **\$OBJECTS** expands to a comma-delimited list of object files to be linked together. The LDF supports string macros, making it easier to read them.

\$ADI_DSP is the home directory for VisualDSP++. You can also define macros in the LDF. The LDF macros are 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;
```

Mapping Files to Memory: the Linker Description File

`$COMMAND_LINE_OBJECTS` is an LDF command-line macro, which expands to a list of all the input object file names on the linker's command line. When working with VisualDSP++ IDDE, all object files listed under the `Link` option will be present on the linker's command line.

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

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

5. The `OUTPUT()` command directs the linker to produce an executable (`.DXE`) file, and specifies the filename. In this listing, the argument to the `OUTPUT` command is the `$COMMAND_LINE_OUTPUT_FILE` macro. Therefore the linker names the executable according to the text following its `-o` switch.

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

6. `SECTIONS` command defines the placement of code and data in physical memory. Based on the three parameters to each line of the `SECTIONS` 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 files that the linker uses to resolve the mapping.



In the example of [Listing 2-1](#) , two Output Sections (`dxcode` and `y_input`) are mapped into one Memory Segment (`seg_code`), as shown below:

```
dxcode {INPUT_SECTIONS($OBJECTS(seg_code))} >seg_code*
dxcode2 {INPUT_SECTIONS($OBJECTS(y_input))} >seg_code
```

The first line directs the linker to place the object code assembled from the `seg_code` Input Section, place it in the `dxcode` Output Section, and map it to the `seg_code` Memory Segment. The second line does the same for Input Section (`y_input`) and Output Section (`dxcode2`), mapping them to the same Memory Segment.

The two pieces of code follow each other in the `seg_code` Memory Segment. The linker does not know the order in which include assembled code. If there is more than one file in the `y_input` Section, the order likely follows the order in the `$OBJECTS` macro, which may in turn depends on the files listed on the command line.

For more information on this process, see [“Linker Guide” on page 2-14](#). For information on other LDF commands and syntax, see [“Linker Description File Reference” on page 2-41](#).

* This line perversely repeats the name (`seg_code`), but it demonstrates the independence of Input Section, Output Section and Memory Segment namespaces.

Linker Guide

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

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

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

The assembler section declarations in this document correspond to an ADSP-21xx Family assembler's `.SECTION` directive or legacy `.MODULE` and `.VAR/SEG=segName` directives.



The VisualDSP++ environment treats the LDF as a key part of the project that provides input to the Linker. You can access it as a source file in the project's file display.

Linker Operations

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

- [“Describing the Link Target” on page 2-15](#)
- [“Writing a MEMORY{} Command” on page 2-16](#)

- “Placing Code on the Target: the SECTIONS Command” on page 2-23
- “Using Link Features” on page 2-25
- “Setting Linker Options” on page 2-26
- “Interpreting Linker Error and Warning Messages” on page 2-26

Describing the Link Target

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

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



If you do not include an LDF, the linker uses a default linker description file that matches the `-Dprocessor` switch on the linker’s command line. The examples in this manual mostly correspond to `-DADSP-219x`, the ADSP-219x core.

Writing a MEMORY{} Command

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 steps for writing a `MEMORY{}` command are as follows:

1. List the ways that your program uses memory in your system. Typical uses for Memory Segments are interrupt table, initialization data, PM code, PM data, DM data, heap space, and stack space.
2. List the types of memory in your system and the address ranges of each type of memory. A memory type has several characteristics: PM, DM, RAM, ROM, or PORT; and word width.
3. Combine the information in these two lists to declare the Memory Segments in your system with a linker `MEMORY{}` command.

A Detailed Memory{} Command

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 steps that follow show a representative LDF associated with a hypothetical project. This LDF specifies several Memory Segments, used to support the `SECTIONS` command.

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 specified by the developer in the source code. Both the memory section names and the output section names are defined in the LDF. While the memory labels are used primarily within the

LDF, the outputs section labels can be important to downstream tools. (For example, one can invoke `elfdump` to dump the contents of the "sec_data1" section of an executable file).

The memory names are defined in the `MEMORY()` command; the output section names above are defined in the `SECTIONS()` command. The default LDF has been constructed to handle all the sections that might be generated by the compiler (the column "Input Section").

[Table 2-1](#) provides Memory vs. Sections usage for the ADSP-2192 DSP. The correspondences used in the ADSP-2192 default LDF are:

Table 2-1. ADSP-2192 Memory vs. Sections Usage

Input Section	Output Section	Memory
program	seg_code	seg_code (Program Memory code)
data1	seg_data1	sec_data1 (Data Memory data)
data2	seg_data2	sec_data2 (Program Memory data)
seg_heap	seg_heap	sec_heap (heap space)
seg_stack	seg_stack	sec_stack (stack space)
sec_ctor	seg_data1	sec_data1 (Data Memory data)
IVreset	seg_reset	sec_reset (reset interrupt vector)
IVpwrdsn	seg_itab	sec_itab (interrupt table: a vector to each service routine)
IVkernel	seg_itab	sec_itab

Table 2-1. ADSP-2192 Memory vs. Sections Usage

Input Section	Output Section	Memory
IVstackint	seg_itab	sec_itab
IVint4 - 15	seg_itab	sec_itab
lib_int_table	seg_itab	sec_itab



This LDF relies on run-time header execution to specify 64K page sizes, incorporated into the program's placement in memory.

2. Memory characteristics—The ADSP-219x has a 24-bit addressing range, so it can support memory addresses from 0x0 to 0xfffff, in 24- and 16-bit widths.
3. MEMORY{} Command—referring to steps 1 and 2, specify the target's memory with the MEMORY{} command in [Listing 2-2](#).

As an example of LDF structure for a ADSP-218x DSP, [Table 2-2](#) provides Memory vs. Sections usage for the ADSP-2189 DSP. The correspondences used in the ADSP-2189 default LDF are:

Table 2-2. ADSP-2189 Memory vs. Sections Usage

Input Section	Output Section	Memory
program	seg_code	seg_code (Program Memory code)
data1	seg_data1	sec_data1 (Data Memory data)
data2	seg_data2	sec_data2 (Program Memory data)
seg_heap	seg_heap	sec_heap (heap space)
seg_stack	seg_stack	sec_stack (stack space)

Table 2-2. ADSP-2189 Memory vs. Sections Usage

Input Section	Output Section	Memory
IVreset	seg_inttab	sec_inttab (reset interrupt vector)
IVpwrdown	seg_inttab	sec_inttab (interrupt table: a vector to each service routine)
IVstackint	seg_inttab	sec_inttab (interrupt table)
IVirq0	seg_inttab	sec_inttab
IVirq1	seg_inttab	sec_inttab
IVirq2	seg_inttab	sec_inttab
IVirq11	seg_inttab	sec_inttab
IVirq10	seg_inttab	sec_inttab
IVirqe	seg_inttab	sec_inttab
IVsport@xmit	seg_inttab	sec_inttab
IVsport@recv	seg_inttab	sec_inttab
IVbdma	seg_inttab	sec_inttab
IVtimer89	seg_inttab	sec_inttab (for ADSP-2189 DSP)
data2	seg_pmpage	sec_pmpage (PMOVLAY - program memory overlay page)
data1	seg_dmpage	sec_dmpage (DMOVLAY - data memory overlay page)

Listing 2-2. MEMORY, SECTIONS Commands

```
ARCHITECTURE(ADSP-219x)
SEARCH_DIR( $ADI_DSP\219x\lib )
// Libraries from the command line are included in
COMMAND_LINE_OBJECTS.
$OBJECTS = 219x_hdr.doj , 219x_int_tab.doj,
$COMMAND_LINE_OBJECTS , libc.dlb, libdsp.dlb, 219x_exit.doj ,
libsim.dlb;
// This memory map is set up to facilitate testing of the tool
// chain -- code and data area are as large as possible. Code
// is placed in page 0, starting with space reserved for the
// interrupt table. All data is placed in page 1. Note that
// the run time header must initialize the data page registers
// to 1 to match this placement of program data. All pages are
// 64K words.

MEMORY
{
    // The memory section where the reset vector resides
    seg_reset { TYPE(PM RAM) START(0x000000) END(0x00001f)
WIDTH(24) }

    // The memory section where the interrupt vector code
    // and an interrupt table used by library functions
    // resides. The library functions concerned include
    // signal(), interrupt(), raise(), and clear_interrupts()

    seg_itab { TYPE(PM RAM) START(0x000020) END(0x000241)
WIDTH(24) }

    // The default program memory used by the compiler.
    seg_code { TYPE(PM RAM) START(0x000242) END(0x00ffff)
WIDTH(24) }

    // The default DM data memory used by the compiler.
    seg_data1 { TYPE(DM RAM) START(0x010000) END(0x017fff)
WIDTH(16) }

    // The default PM data memory used by the compiler.
    seg_data2 { TYPE(PM RAM) START(0x018000) END(0x01ebff)
WIDTH(24) }
```

```

        // The memory section used for dynamic allocation
        routines.
        seg_heap { TYPE(DM RAM) START(0x01ec00) END(0x01efff)
        WIDTH(16) }

        // The memory section used for the software stack pointed
        to by
        // STACKPOINTER(I4) and FRAMEPOINTER(I5).
        seg_stack { TYPE(DM RAM) START(0x01f000) END(0x01ffff)
        WIDTH(16) }
    }

PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
seg_reset
    {
        INPUT_SECTIONS( $OBJECTS(IVreset))

    } > seg_reset

        sec_itab
        {
INPUT_SECTIONS( $OBJECTS(IVpwrdsn))      = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVstackint))    = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVkernel))      = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint4))        = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint5))        = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint6))        = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint7))        = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint8))        = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint9))        = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint10))       = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint11))       = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint12))       = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint13))       = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint14))       = .+0x1d;
INPUT_SECTIONS( $OBJECTS(IVint15))       = .+0x1d;

        INPUT_SECTIONS( $OBJECTS(lib_int_table))
    }
}

```

Linker Guide

```
} > seg_itab
    seg_code
    {
        INPUT_SECTIONS( $OBJECTS(program) )
    } >seg_code

    sec_data1
    {
        INPUT_SECTIONS( $OBJECTS(data1) )
    } >seg_data1

    sec_data2
    {
        INPUT_SECTIONS( $OBJECTS(data2) )
    } >seg_data2

    // support for initialization, including C++
    sec_ctor
    {
        INPUT_SECTIONS( $OBJECTS(ctor))
    } >seg_data1

    // provide linker variables describing the stack (grows
    // down)
    //   ldf_stack_limit is the lowest address in the stack
    //   ldf_stack_base is the highest address in the stack
    sec_stack
    {
        ldf_stack_limit = .;
        ldf_stack_base = . + MEMORY_SIZEOF(seg_stack) - 1;
    } >seg_stack
    sec_heap
    {
        .heap = .;
        .heap_size = MEMORY_SIZEOF(seg_heap);
        .heap_end = . + MEMORY_SIZEOF(seg_heap) - 1;
    } >seg_heap
}
```

Notes on Listing 2-2:

[Listing 2-2](#) applies to the preceding discussion of how to write a `MEMORY` command, and to the following discussion of the `SECTIONS` command.

1. It has been reformatted for easy reading when location is a word short. Normally, the location counter increments (`. = . + 1;`) between the section listings appear on separate lines: they are not part of the `INPUT_SECTIONS` command. They specify moving the current location one word.
2. As shown in [Listing 2-2](#), the `SECTIONS` command is not atomic: it can be interspersed with other directives, including location counter information.
3. You can define new symbols within the LDF; this example defines the starting stack address, the highest possible stack address, and the heap's starting location and size. These newly-created symbols will be entered in the executable's symbol table.

Placing Code on the Target: the `SECTIONS` Command

The `SECTIONS{}` command lets you place your program's Input Sections in the memory of a DSP system, as defined by the `MEMORY{}` command. Each Input Section is declared as such in your assembly code.

You must embed `SECTIONS{}` commands within the linker's `PROCESSOR{}` command. These commands tell the linker to place the code in memory allocated to that processor, or shift it between overlays. For information on assembler directives supporting these linker commands, see the *VisualDSP++ 2.0 Assembler and Preprocessor Manual for ADSP-218x DSPs* or *VisualDSP++ 2.0 Assembler and Preprocessor Manual for ADSP-219x DSPs*.

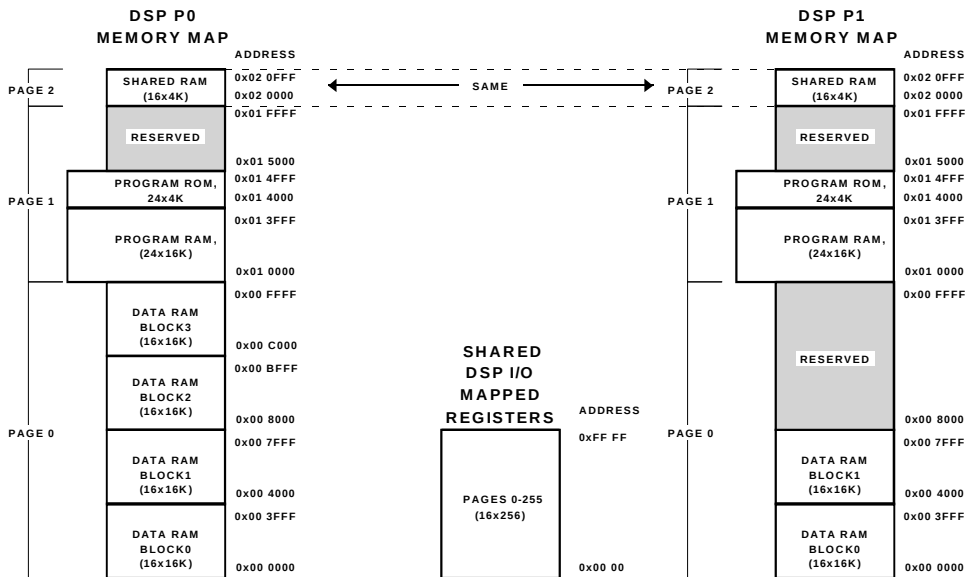
Linker Guide

Follow the steps below to write a linker `SECTIONS{}` command:

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

The system in [Figure on page 2-24](#) and any C program can provide a generic example for exercising this analysis technique.

Figure B-2. ADSP-2192 System Map



The following paragraphs show a sample system description.

1. Program sections—The VisualDSP++ C/C++ compiler produces code that can use memory in predefined ways, with predefined section labels.
2. Linker `MEMORY{}` Command—A `MEMORY{}` command for the system appears in [Listing 2-2](#) . This command declares Memory Segments containing the Output Sections produced by the compiler.
3. Linker `SECTIONS{}` Command—Combining the information from steps 1 and 2, you could specify how to place code for the system with the `SECTIONS{}` command in [Listing 2-2](#) .

Using Link Features

The three previous sections, “[Describing the Link Target](#)” on page 2-15, “[Writing a MEMORY{} Command](#)” on page 2-16, and “[Placing Code on the Target: the SECTIONS Command](#)” on page 2-23, provide an overview of how to link ordinary executables for DSP systems.

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 2-49.](#)

The flexible nature of the linker’s command language lets you make linking as simple or as complex as fits your needs. The linker supports the following actions.

- linking executables for systems with memory overlays
- calling additional LDFs from the command line
- calling files of linker commands or object filenames
- specifying search paths for object files and libraries

Linker Guide

- using (linking against) pre-existing executable files to resolve references

For more information on these topics, see the following sections:

- For information on overlay memory, see the command syntax for “[SECTIONS{}](#)” on page 2-79 and “[PLIT{}](#)” on page 2-72.
- Examples of overlay linking appear in “[Linking for Overlay Memory](#)” on page 2-96 and “[Using a Procedure Linkage Table](#)” on page 2-100.

Setting Linker Options

When you develop code within the VisualDSP++ 2.0 IDDE, you can set tool settings for all files in a project, individually or project-wide. You may find it useful to modify the linker’s default option settings in the VisualDSP++ environment. The linker also has command line switches that correspond to the `Link` property page in the `Project Options` settings.

For information on setting linker options from within VisualDSP++ 2.0 IDDE, see the *VisualDSP++ 2.0 User’s Guide for ADSP-21xx DSPs*.

Interpreting Linker Error and Warning Messages

Linker warning or error messages describe problems that the linker encountered when processing the LDF.

The linker reports link warnings and errors to `stderr`. If you build within VisualDSP++ 2.0 IDDE, these messages appear in the **Output Window**. The message contents are independent of the link-time environment.

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

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

```
[Error E2005] 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 of `my_file.ldf`.

When you develop code within the VisualDSP++ 2.0 IDDE, the Output window displays project build status and error messages. In most cases, you can double-click on a message or error number and the IDDE jumps to the source file and line number that contains the error. However, some build errors — such as incorrect or missing cross references to an object or executable file — do not correlate with any source file locations, so the IDDE cannot move to the erroneous file and line.



Note that errors relating to missing cross references often stem from omissions in the LDF. For example, if an Input Section 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 segment. Reviewing the LDF, and correcting it to specify all segments that need placement, typically solves this sort of error.

Linker Command-Line Reference

This section provides reference information on the linker command-line and linking. A description for each switch appears in [“Linker Command-Line Reference” on page 2-28](#).

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

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



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

Command-Line Syntax

The general, normalized format for a linker command line is shown below:

```
linker -switch [-switch ...] object [object ...]
```

Object Files in the Linker Command Line

The command line typically has one (typically more) object file(s) to be linked together. These files may also come from LDF as well. These files may be of several different types.

- The standard object file is produced by the assembler or compiler and has a .DOJ extension.

- It may also be an archive (library) of one or more files, with a `.DLB` extension. These may be obtained from the VisualDSP++ libraries, or from your own code.
- It may also be an executable (`.EXE`) file to be linked against*.



Object file names are not case-sensitive. But linker switches *are* case sensitive: `linker -t` is not the same as `linker -T`.

An object filename has the following characteristics:

- It can include the drive, directory path, filename, and file extension
- Its path may be absolute or relative to the directory where the linker is invoked
- It should enclose long file names within double-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 2-31](#) lists the allowed extensions and matching linker operations.

Switch Format in the Linker Command Line

The linker has many optional switches. These select the operations and modes for the compiler and other tools. The standard linker switch syntax is shown below.

- `-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.

* “Link Against” is described on [page -29](#), under `$COMMAND_LINE_LINK_AGAINST`

Linker Command-Line Reference

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

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

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

- Links object files `p0.doj`, `p1.doj` and `p2.doj` 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.

Filenames on the Linker Command Line

Many linker switches take a file name as an optional parameter. [Table 2-3 on page 2-31](#) lists the types of files, names, and extensions that the linker expects on filename arguments.

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

1. *Specified path*—If you include relative or absolute path information in a file name, the linker searches in that location for the file.
2. *User-selected directories*—If you do not include path information in the file name and the file is not in a default directory, the linker searches for the file in search directories that you select with the `-L`

(path) command line switch and `SEARCH_DIR` commands in the LDF. The linker searches these directories in the order that they appear on the command line or in the LDF.

3. *LDF directory*—If you do not include path information in the LDF named in the `-T` switch, the linker searches for the LDF named in the current working directory.

If you use a default LDF by omitting any LDF information in the command line, the linker searches in the processor-specific `ldf` directory, for example `\$ADI_DSP\219x\ldf`.

For more information on file searches, see [“LDF Macros” on page 2-49](#).

The linker follows the conventions for file name extensions listed in [Table 2-3](#). 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, containing spaces or special characters, within double quotation marks.
- Append the appropriate name extension to each file. The linker opens existing files and verifies their type before processing. When the linker creates a file, it uses the file extension to determine the type of file to create.

Table 2-3. File Name Extension Conventions

Extension	File Description
.dlb	Library (archive) file
.doj	Object file
.dxe	Executable file

Linker Command-Line Reference

Table 2-3. File Name Extension Conventions (Cont'd)

Extension	File Description
.ldf	Linker description file
.ovl	Overlay file

Linker Command-Line Options

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

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

Table 2-4 provides a summary of the linker's command-line options; a description of each option starts on [page 2-35](#).

Table 2-4. Linker Command-Line Switches

Switch	Description
<i>filename(s)</i> on page 2-35	Uses a file type when processing files that are not parameters to a switch.
<null> on page 2-35	Displays a summary of command line options and exit.
@ <i>file</i> on page 2-35	Uses <i>file</i> as input to the linker command line.
-D <i>architecture</i> on page 2-36	Specifies target architecture.
-L <i>path</i> on page 2-36	Adds path name to search libraries and objects.
-M on page 2-36	Produces dependencies.
-MM on page 2-36	Builds and produces dependencies.
-Map <i>file</i> on page 2-36	Outputs a map of link symbol information to <i>file</i> .
-MD <i>macro</i> [= <i>def</i>] on page 2-37	Defines and assign value <i>def</i> to preprocessor macro <i>macro</i> .
-S on page 2-37	Omits debugger symbol information.
-T <i>file</i> on page 2-37	Uses <i>file</i> as the LDF to specify executable program placement.
-e on page 2-37	Eliminates unused symbols from the executable.
-ev on page 2-37	Eliminates unused symbols verbosely.
-es <i>secName</i> on page 2-38	Names sections (<i>secName</i> list) to which elemiation algorithm is being applied.
-help on page 2-38	Displays a summary of command line options and exits.

Linker Command-Line Reference

Table 2-4. Linker Command-Line Switches (Cont'd)

Switch	Description
-i <i>directory</i> on page 2-38	Includes search directory.
-ip on page 2-38	Fills in fragmented memory with individual data objects that fit.
-keep <i>symName</i> on page 2-39	Keeps unused symbols; directs the linker.
-o <i>filename</i> on page 2-39	Outputs executable file <i>filename</i> .
-pp on page 2-39	Ends after preprocessing.
-s on page 2-39	Strips all symbols.
-sp on page 2-39	Skips preprocessing.
-t on page 2-39	Directs the linker to output the names of link objects.
-v on page 2-39	Verbose: directs the linker to output status information.
-version on page 2-40	Directs the linker to output its version and exit.
-warnonce on page 2-40	Warns only once for each undefined symbol.
-xref <i>filename</i> on page 2-40	Outputs a list of all cross referenced symbols.

filename

When processing files that are not parameters to a switch, the linker uses a file's type to determine how to handle it. The linker gets a file's type as follows:

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

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

The linker treats executable (.dxe) and shared-memory (.sm) files on the command line as executables to be linked against. For information on executables, see the \$COMMAND_LINE_LINK_AGAINST linker macro.

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

<null>

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

@ *file*

Uses *file* as input to the linker command line. This switch lets you circumvent environmental command line length restrictions.

file may not start with “linker” (it can't be a linker command line). Any whitespace in *file* serves to separate tokens, including newline.

Linker Command-Line Reference

-Darchitecture

Specifies 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 LDF is an error.

-L path

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

-M

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

-MM

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

-Map file

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

-MDmacro[=def]

Defines and assigns value *def* to preprocessor macro *macro*.

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

-S

Omits debugger symbol information (*not* all symbol information) from the output file. Compare with `-s` switch, below.

-T file

Uses *file* as the LDF to specify executable program placement.

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

file must exist and can be found (e.g. via the `-L` option); there must be whitespace before *file*.

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). Multiple `-T` options accumulate.

-e

Eliminates unused symbols from the executable.

-ev

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

Linker Command-Line Reference

-es *secName*

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

-help

Displays a summary of command-line options and exit. Same as `linker` with no argument.

-i *directory*

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

-ip

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

The `-ip` switch works only with objects assembled with the assembler's `-ip` switch. The `-ip` switch is a default for ADSP-218x DSPs.

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

The `-noip` option (in assembler) turns off the individual placement option. See the *VDSP++ 2.0 Assembler and Preprocessor Manual for 21xx DSPs*.

-keep *symName*

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

-o *filename*

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

-pp

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

-s

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

-sp

Skips preprocessing: link without preprocessing the LDF.

-t

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

-v

Verboses: outputs status information while linking.

Linker Command-Line Reference

-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

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

- [“LDF Structure” on page 2-42](#)
- [“LDF Expressions & Conventions” on page 2-43](#)
- [“Linker Expression Syntax” on page 2-44](#)
- [“Linker Keywords” on page 2-45](#)
- [“LDF Operators” on page 2-47](#)
- [“LDF Macros” on page 2-49](#)
- [“LDF Commands” on page 2-53](#)
- [“LDF Programming Examples” on page 2-88](#)

Because the linker runs the preprocessor on the LDF, you can use preprocessor commands (such as `#define`) within your LDF. For information on preprocessor commands, see the *VisualDSP++ 2.0 Assembler and Preprocessor Manual for ADSP-218x DSPs* or *VisualDSP++ 2.0 Assembler and Preprocessor Manual for ADSP-219x DSPs*.

LDF Structure

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

- Divide your 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. Memory that is unique to a processor will be declared within the scope of the corresponding `PROCESSOR{ }` command. Common memory definitions (shared or multiprocessor memory) are declared in the global LDF scope, before any `PROCESSOR{ }` commands.
- In ADSP_219x DSPs, place `MPMEMORY{ }` or `SHARED_MEMORY{ }` commands in the global LDF scope if they apply to your system. These commands represent system resources available to multiple processors.

For more information on LDF structure, see [“Describing the Link Target” on page 2-15](#), [“Placing Code on the Target: the SECTIONS Command” on page 2-23](#), and [“LDF Programming Examples” on page 2-88](#).

Command Scoping

The two LDF 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 (see [“LDF Macros” on page 2-49](#)) are available globally, regardless of the scope where the macro is defined.

Figure B-22-3 demonstrates some scoping issues. For example, the `MEMORY{}` command that appears in the global LDF scope is available in all the command scopes, but the `MEMORY{}` commands that appear in the command scopes are restricted to those scopes.

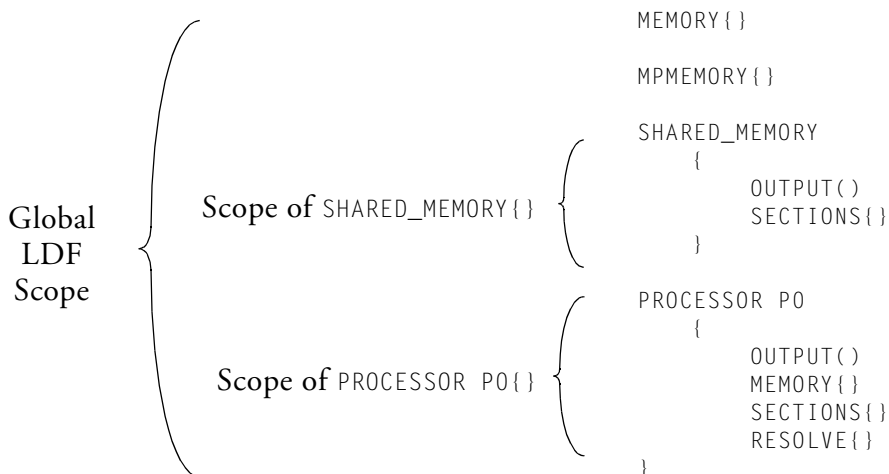


Figure 2-3. LDF Command Scoping Example

LDF Expressions & Conventions

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

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

Convention	Description
.	In an address expression, refers to the current location pointer.
0xnumber	A hexadecimal number
number	(No prefix): a decimal number

Linker Description File Reference

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

Convention	Description
<i>numberk</i> (or <i>K</i>)	A decimal number multiplied by 1024.
<i>/* comment */</i>	C style comments: can cross newline boundaries until <i>*/</i> is encountered.
<i>// comment</i>	A <i>“//”</i> string precedes single-line C++ style comments.
<i>[Bb]#number</i>	A binary number

Linker Expression Syntax

Linker commands may contain arithmetic expressions. These expressions follow the same syntax rules as 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 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

Linker Keywords

The linker recognizes labels conforming to the following constraints:

- Must start with a letter, underscore, or point
- May contain any letters, underscores, digits, and points
- Are whitespace-delimited
- Do not conflict with any keywords, and are unique.

Descriptions of linker keywords from [Table 2-6 on page 2-45](#) appear in the following sections:

- [“Miscellaneous LDF Keywords” on page 2-47](#)
- [“LDF Operators” on page 2-47](#)
- [“LDF Macros” on page 2-49](#)
- [“LDF Commands” on page 2-53](#)



Keywords are **case-sensitive**; the linker only recognizes a keyword when the **entire** word is in **uppercase** letters. Within the `PLIT{}` command, you may use any valid assembly instruction.

Table 2-6. Linker Keywords

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BOOT	COMAP
DEFINED	DM	DYNAMIC
ELIMINATE	ELIMINATE_SECTIONS	END
FALSE	FILL	FIRST_FIT

Linker Description File Reference

Table 2-6. Linker Keywords (Cont'd)

INCLUDE	INPUT_SECTION_ALIGN	INPUT_SECTIONS
KEEP	LENGTH	LINK_AGAINST
MAP	MEMORY	MEMORY_SIZEOF
MPMEMORY	NUMBER_OF_OVERLAYS	OUTPUT
OVERLAY_GROUP	OVERLAY_ID	OVERLAY_INPUT
OVERLAY_OUTPUT	PACKING	PAGE_INPUT
PAGE_OUTPUT	PLIT	PLIT_DATA_OVERLAY_IDS
PLIT_SYMBOL_ADDRESS	PLIT_SYMBOL_OVERLAYID	
PM	PORT	PROCESSOR
RAM	RESOLVE	RESOLVE_LOCALLY
ROM	SEARCH_DIR	SECTIONS
SHARED_MEMORY	SHT_NOBITS	SIZE
SIZEOF	START	TRUE
TYPE	VERBOSE	WIDTH
XREF		

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 2-47](#). For more information about linker keywords that are macros, see [“LDF Macros” on page 2-49](#). For more information about linker keywords that are commands, see [“LDF Commands” on page 2-53](#).

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

BOOT—Boot memory, memory from which the 219x can be booted.

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

FALSE—A constant with a 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 containing these operators terminate with a semicolon, except when you use the operator as a variable for an address.

The linker responds to the following LDF operators as follows:

- `ADDR(section_name)`

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

Linker Description File Reference

- `ALIGN(address_boundary_expression)`

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

- `DEFINED(symbol)`

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

- `ELIMINATE()`

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

- `ELIMINATE_SECTIONS(sectionList)`

Turns on section elimination, applying the elimination algorithm only to the named sections from the executable if they are not called. The *sectionList* is the whitespace-delimited list of sections.

- `KEEP(keepList)`

When section elimination is on, keeps the listed objects in the executable even if they are not called. The *keepList* is the comma-delimited list of objects.

- `MEMORY_SIZEOF(segment_name)`

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)`

Returns the size 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, `(.)`

Treats a whitespace-delimited period as the symbol for the current location counter. Because “.” only refers to a location in an Output Section, this operator may only appear within the `SECTIONS{}` command. Note that the `.` operator starts at the start address of the target memory segment and increments (in units of the target's word length) through the segment's addresses; it may not be decremented, or moved backwards. Observe the following rules when manipulating the `.` operator:

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

For an example of location counter usage, see the interrupt table construction on [page 2-23](#).

LDF Macros

Macros are names of text strings. They may be assigned values, textual or procedural, or simply declared to exist. Programs encountering these identifiers in their input files can implement macros in several ways:

- Substitute the string value for the name. Normally the string value is longer than the name, so the macro “expands” to its textual length.
- Perform actions conditional on the existence or value of the macro.

Linker Description File Reference

- 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.

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

Macros in the LDF funnel input from the linker command line into pre-defined macros and provide support for user defined macro substitutions. Linker macros are available globally in the LDF regardless of where they are defined. For more information on these topics, see [“LDF Macros and Command-Line Interaction” on page 2-52](#).



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

LDF Macro List

The linker supports the following LDF macros:

- `$COMMAND_LINE_OBJECTS`

Expands into the list of object (`.DOJ`) and library (`.DLB`) files that are input on the linker's command line. It is useful within the `INPUT_SECTIONS{}` syntax of the linker's `SECTIONS{}` command. This macro provides a comprehensive list of object file input that the linker searches for Input Sections.

- `$COMMAND_LINE_LINK_AGAINST`

Expands to the list of executable (.DXX 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`

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

- `$ADI_DSP`

Expands to the path to the VisualDSP++ installation directory. This macro is useful when controlling how the linker searches for files.

- `$macro = list_of_files ;`

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

LDF Macros and Command-Line Interaction

Whether you run the linker from the VisualDSP++ IDDE 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 2-49.](#)

Whether you should you 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 only specify a single output from the command line, an LDF that only relies on command-line input should be written to produce one output file for a single processor system.
The command-line interface for the linker does not let you name the multiple outputs that you need for multiprocessor, shared memory, or overlay memory.
2. Writing an LDF that *does not use* command-line inputs can produce a more specific LDF that you can use with more complex linker features. The environment interface for the linker lets you name the multiple outputs that you need for multiprocessor, shared memory, or overlay memory.

LDF Commands

Commands in the LDF define the target system and order the processing of linker 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 2-42.](#) Filenames containing white space or colons must be enclosed in straight quotes.

This section describes the following LDF commands:

- [“ARCHITECTURE\(\)” on page 2-54](#)
- [“DYNAMIC\(\)” on page 2-54](#)
- [“INCLUDE\(\)” on page 2-55](#)
- [“INPUT_SECTION_ALIGN\(\)” on page 2-55](#)
- [“LINK_AGAINST\(\)” on page 2-57](#)
- [“MAP\(\)” on page 2-57](#)
- [“MEMORY{}” on page 2-58](#)
- [“MPMEMORY{}” on page 2-61](#)
- [“OVERLAY_GROUP{}” on page 2-62](#)
- [“PACKING\(\)” on page 2-67](#)
- [“PAGE_INPUT\(\)” on page 2-71](#)
- [“PAGE_OUTPUT\(\)” on page 2-72](#)
- [“PLIT{}” on page 2-72](#)
- [“PROCESSOR{}” on page 2-77](#)
- [“SEARCH_DIR\(\)” on page 2-79](#)

Linker Description File Reference

- “[SECTIONS{}](#)” on page 2-79
- “[SHARED_MEMORY{}](#)” on page 2-85 (used with ADSP-219x DSPs)

ARCHITECTURE()

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

 The `ARCHITECTURE()` command is case-sensitive. Hence, `ADSP-2181` is a legal value but `adsp-2181` is not.

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

Syntax: `ARCHITECTURE(processor)`

 If you specify a processor type that your installation does not support, the system displays an appropriate diagnostic message.

DYNAMIC()

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

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

The arguments are defined as follows:

- *resolve_locally*—A boolean variable indicating whether the linker should generate PLIT entries for function calls that can be resolved within the DLO. A value of `TRUE` (the default) instructs the linker **not** to generate PLIT entries for function calls that can be resolved within the DLO; a value of `FALSE` instructs the linker to generate PLIT entries for each and every function call.
- *outputFileName*—The name of the linker output file for this processor. The specified name **must** be a DLO file.
- *sections*—The output section for this processor. See “[SECTIONS{}](#)” on page 2-79 for more information about the sections command.
- *plit*—A processor-specific PLIT description. See “[PLIT{}](#)” on page 2-72 for more information about PLITs.

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. All the 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()

Aligns the input section to the next boundary defined by the input argument. The input argument must be an expression whose value is a power of two. Specifying the command `INPUT_SECTION_ALIGN(1)` ends the alignment directive of the input sections.

The `INPUT_SECTION_ALIGN` command should be used only within the *output_section_command* in the LDF.

Linker Description File Reference

In the following example, the input sections from `a.doj`, `b.doj`, and `c.doj` **will** be aligned. However, the input sections from `d.doj` and `e.doj` may be **not** be aligned, because the `INPUT_SECTION_ALIGN(1)` command has indicated the end of the alignment directive. However, they may be aligned if the alignment criteria is set back to 1, i.e., the object begins at the next available boundary.

```
SECTIONS
{
    code
    {
        INPUT_SECTION_ALIGN(4)

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

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

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

    } >M2Code
}
```

LINK_AGAINST()

Directs the linker to check specified executables (.DXEs and .SMs) to resolve variables and labels that have not been resolved locally. To link multiprocessor programs, you must use the LINK_AGAINST() command in the LDF.

LINK_AGAINST() is an optional part of the PROCESSOR{}, SHARE_MEMORY{}, and OVERLAY_INPUT{} commands. Its syntax (as part of a PROCESSOR{} command) is shown below:

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

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

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

MAP()

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

This command corresponds to and is overridden by the -Map <filename> command line switch. If your VisualDSP++ project's options include generating a symbol map (in the link options property page), that means the linker runs with -Map <projectname>.map asserted, and your LDF's MAP() command will merely draw a warning.

MEMORY{}

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

Your 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 2-42.](#)

In each scope scenario, the `MEMORY{ }` command should be followed by the `SECTIONS{ }` command. Use the memory segment names for placing program `SECTIONS`. Only memory segment declarations may appear within the `MEMORY{ }` command.

Segment names are limited to eight characters in length. There is no limit on section name lengths. If you do not specify the target processor's memory map with the `MEMORY{ }` command, the linker cannot link your program. If the combined sections directed to a segment require more space than exists in the segment, the linker issues an error message and halts the link.

The syntax for the `MEMORY{ }` command appears in [Figure 2-4 on page 2-59](#), followed by definitions for the command's components:

- *segment_commands*

Declare your target processor's memory segments. You may subdivide memory into an arbitrary number of segments. Each segment declaration must contain all of the following parameters:

- a `LENGTH( )` or `END( )` command
- a `WIDTH( )` command

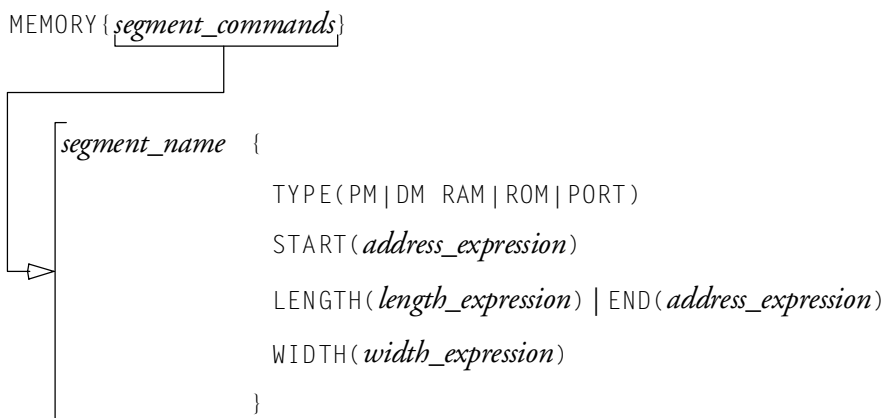


Figure 2-4. Syntax Tree of the MEMORY{} Command

- a *segment_name* command

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

- a segment TYPE command: `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.

The TYPE command provides two items of information:

- the type of memory usage: PM—program memory, DM—data memory
- the memory's functional or hardware locus: RAM, ROM, PORT

Linker Description File Reference

- a `START(address_expression)` command

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

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

Identifies the segment length in bytes or sets the segment's end address. If stating the length, *length_expression* must be the number of addressable words within the region or an expression that evaluates to the number of words. If stating the end address, *address_expression* must be an absolute address or an expression that evaluates to an absolute address, such as `START + LENGTH = END`.

- a `WIDTH(width_expression)` command

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

MPMEMORY{}



The `MPMEMORY{}` command is used with DSPs that implement physical shared memory, such as ADSP-2192-12 DSP.

It 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. The `MPMEMORY{}` command should be followed by `PROCESSOR processor_name{}` commands, containing each processor's `MEMORY{}` and `SECTIONS{}` commands.

The syntax for the `MPMEMORY{}` command appears in [Figure 2-5](#), followed by definitions for the command's components.

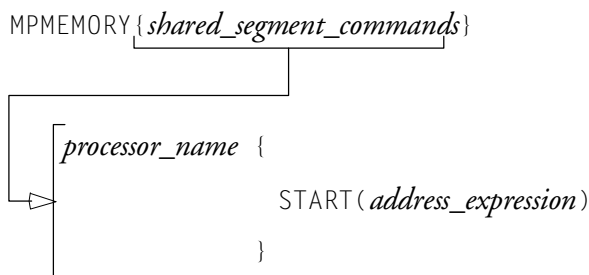


Figure 2-5. Syntax Tree of the `MPMEMORY{}` Command

- *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 “Linker Expression Syntax” on page 2-44.](#)

Linker Description File Reference

- `PROCESSOR processor_name{placement_commands}`

Applies the *processor_name* offset for multiprocessor linking. [For more information, see “PROCESSOR{}” on page 2-77.](#)

OVERLAY_GROUP{}

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

Overlaying lets you improve performance by placing currently executing code in your fastest memory, though the entire program may be too large to fit in that memory. Performance is most enhanced in code which 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 sequentially from a single starting address in “live” memory. The `OVERLAY_GROUP` command lets you group overlays. The command creates a set of overlays; each set has its own run memory. Therefore, one overlay from each set may be active at the same time. [Figure 2-6 on page 2-65](#) and [Figure 2-7 on page 2-66](#) demonstrate ungrouped versus grouped overlays.

In the examples below, 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 only active when they are executed in runtime memory, all of which is located in segment `pmco`.

The LDF fragments below, [Listing 2-4 on page 2-64](#) and [Listing 2-5 on page 2-65](#), represent parts of a single processor’s `SECTIONS{}` command. They are pseudo-code for a generic target, not examples of a particular implementation. [Listing 2-3](#) shows where they fit into an LDF.

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

Listing 2-3. Prologue for LDF Overlay Examples

```

ARCHITECTURE(ADSP-XXXX)                                // Generic.
SEARCH_DIR ( $ADI_DSP\XXXX\LIB )                       // Generic.
$LIBRARIES = libc.dlb, libio.dlb;
$OBJECTS = $COMMAND_LINE_OBJECTS;

MEMORY
{
    // Generic - do not use!
    seg_rth { TYPE(PM RAM) START(0x...) END(0x...) WIDTH(..) }
    seg_init { TYPE(PM RAM) START(0x...) END(0x...) WIDTH(..) }
    seg_code { TYPE(PM RAM) START(0x...) END(0x...) WIDTH(..) }
    // Overlays "live" in DM RAM
    seg_ovly { TYPE(DM RAM) START(0x...) END(0x...) WIDTH(..) }
    ...
} // End of MEMORY command

PROCESSOR P0 {
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)

    SECTIONS {
        dxr_rth { INPUT_SECTIONS( $OBJECTS(seg_rth) ) >seg_rth
        dxr_init { INPUT_SECTIONS( $OBJECTS(seg_init) ) >seg_init
        ...
        dxr_code { INPUT_SECTIONS( $OBJECTS(program)
            $LIBRARIES(program)
            // THE OVERLAY MANAGER ROUTINE OverlayManager
            // GOES HERE. IT STAYS HERE

            // *** THE OVERLAY DEFINITIONS,
            // *** Listing 2-4 AND Listing 2-5 GO HERE
        } > seg_code

        // .plit is the output section used by the linker
        .plit {} > seg_pmco // Overlay tracking: see page 2-72

        // More SECTION definitions
    } // End of SECTIONS command.

```

Linker Description File Reference

Ungrouped Overlay Execution

In this example, as the FFT progresses and the overlay functions are called in turn, they are brought into runtime memory in sequence; four function transfers.



Note that the “live” locations reside in several different memory segments. Note also that the linker outputs the overlay archive (.OVL) files while allocating destinations for them in `seg_pmco`.

Listing 2-4. LDF Overlays, Not Grouped

```
// This listing is part of the SECTIONS command for
// Processor P0. See Listing 2-3 on page 2-63.

// Declare which functions reside in which overlay. The over-
// lays have been split up either into different segments in
// one file, or into different files. The overlays declared
// in this section, seg_pmco, will run in seg_pmco.

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

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

OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS(  Fft_3rd.doj(pm1_co) ) } >ovl_code

OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_last.ovl)
    INPUT_SECTIONS(  Fft_4th.doj(pm1_co) ) } >ovl_code

    INPUT_SECTIONS(  Fft_last.doj(pm2_code) ) } >ovl_code
```

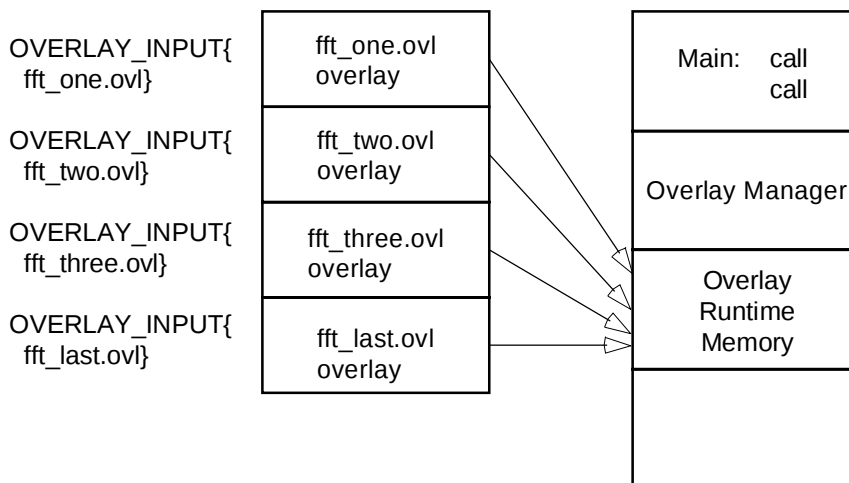


Figure 2-6. Overlays, Not Grouped

Grouped Overlay Execution

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

[Listing 2-5](#) also fits into the same place in the common prologue, [Listing 2-3 on page 2-63](#): it is part of the `SECTIONS{}` command for Processor P0.

Listing 2-5. LDF Overlays, Grouped

```
OVERLAY_GROUP {          // Declare first overlay group
  OVERLAY_INPUT {        // Overlays to live in section ovl_code
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_1st.doj(pm_code) ) } >ovl_code
  OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
```

Linker Description File Reference

```

OVERLAY_OUTPUT(fft_two.ovl)
INPUT_SECTIONS(  Fft_mid.doj(pm_code) ) } >ovl_code
}

OVERLAY_GROUP {      // Declare second overlay group
  OVERLAY_INPUT {    // Overlays to live in section ovl_code
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS(  Fft_last.doj(pm1_code) ) } >ovl_code
  OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_last.ovl)
    INPUT_SECTIONS(  Fft_last.doj(pm2_code) ) } >ovl_code
}

```

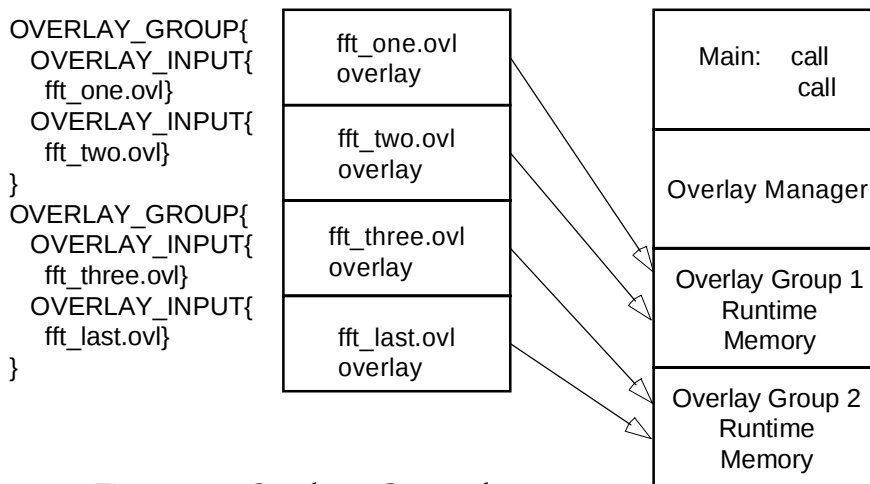


Figure 2-7. Overlays, Grouped

PACKING()

The DSP exchange data with their environments (on-chip or off-chip) through several buses. In the 21xx family, some are 24 bits wide (PMA, DMA, PMD, ...), some are 16 bits wide (DMD). The configuration, placement, and amounts of memory for each device are implementation-specific. Please refer to data sheets and reference manuals for particular DSPs and DSP families.

The linker must be able to place data in memory according to the constraints imposed by your system's architecture. For this purpose, 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 lets the linker structure its executable output to comport with your target'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 non-trivial use of the `PACKING()` command *reorders* bytes in the executable (`.DXX`, `.SM` or `.OVL`) files, 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 command syntax is:

```
PACKING(number_of_bytes byte_order_list)
```

- The `number_of_bytes` is an integer specifying the number of bytes to pack (reorder) before repeating the pattern

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

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

Linker Description File Reference

- The `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`.

If the processor unpacks three 16-bit memory words to build two 24-bit instruction words, the following unpacked and storage orders could apply to such a data transfer mode:

Table 2-7. Unpack vs. Storage Order

Unpacked Order: Two 24-Bit Internal Words	Native storage order; 16-bit external memory
<i>b23 b00, b23 b00 [Bit labels]</i> B3, B2, B1, B6, B5, B4 word1 word2	B2, B1, B4, B3, B6, B5 addr[n] addr[n+1] addr[n+2]

Because the order of unpacked bytes does not match the requisite transfer order, the linker must reorder the bytes into their unpacked order.

Depending how the memory interface to the PMD (program memory bus) is programmed, a command for doing so could be:

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

Each set of 6 bytes would then be reordered as shown above.

The `PACKING()` order must match how the target is programmed to transfer data. In the example above, the target must handle each 16-bit memory read (in a set of three) differently, which is computationally expensive.

Efficient Packing

The packing shown in [Table 2-7](#) is *efficient*. Each 16-bit memory location is used in its entirety to build 2/3 of an instruction word on the target. Such a scheme carries the benefit of limiting program size at the target, and perhaps decreasing download time. Conversely, it implies programming overhead and transfer latency constraints: each address in a set of three memory reads must be handled differently. For that reason, the ordering shown above is possible rather than obligatory: you could program the port to handle any reordering.

Inefficient Packing: Null Bytes

Given the target byte order requirements shown in [Table 2-7](#), you can program memory access more far more simply by directing the linker to add unused (“null”) bytes to your executable, so alignment and sequencing are easier.

Consider an executable that places bytes in the following order, into three 16-bit memory addresses (MSByte on the left):

B2, B1, B4, B3, B6, B5

which you want to load as two 24-bit instructions:

B3, B2, B1, B6, B5, B4

Reordering them with `PACKING(6 B3 B2 B1 B6 B5 B4)` leaves alignment and programming overhead, as shown above. However, adding null bytes after the third and sixth bytes simplifies things considerably.

`PACKING(6 B3 B2 B1 B0 B6 B5 B4 B0)`

Linker Description File Reference

This new order defines four 16-bit memory reads to generate two 24-bit addresses

- Reads 1 and 3 are copied to the two MSBytes on PMD
- Reads 2 and 4 are copied to the LSByte on PMD. The lower byte is ignored

Note that the same number of bytes are reordered as in the efficient packing example. Hence the byte count parameter to the `PACKING()` command is the same (6).



A byte designated `B0` in the `PACKING()` syntax acts as a place holder. The linker sets that byte to zero in the external memory.

Overlay Packing Formats

The `PACKING()` command also packs code and data for overlays, which by definition “live” external memory. You need an explicit `PACKING()` command whenever the width or byte order of stored data (executable or overlay in “live” location) differs from its run-time organization.

Trivial Packing—no reordering

If your memory organization matches its run-time environment, and will load and run without reordering, the linker uses (implicit) “trivial” packing. Only nontrivial 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.

PAGE_INPUT()



The `PAGE_INPUT()` command is supported for paged memory architectures, such as in ADSP-218x DSPs.

The `PAGE_INPUT()` command is used to specify overlays for memory pages. The command `PAGE_INPUT()` can be used instead of the command `OVERLAY_INPUT`, in any location in the LDF file.

The linker creates an additional symbol in each page overlay object. The symbol name identifies the overlay as one created from a `PAGE_INPUT()` command, identifies the register name used to activate the page, and specifies the page value. The linker determines the register name based on the type of memory selected for “run” and “live” space for the overlay. The linker determines the page value based on the description of the “live” space specified in the `PAGE_INPUT()` command.

Page overlays that appear in the `PAGE_INPUT()` commands as “live” memory must first be described in the `MEMORY{}` command. The address specified should contain a leading digit to indicate the page number.

For example, the memory corresponding to `PMOVLAY 4` on the ADSP-2187 DSP could appear in the `MEMORY{}` command as:

```
seg_page4{ TYPE(PM RAM) START (0x42000) END(0x43FFF) WIDTH(24) }
```

According to this definition, the memory section to operate as both the “runtime” memory for all of the paged overlays and the “live” memory for `PMOVLAY 0` has page value 0, and the start address for this section is 0x2000. In implementations where there is no internal memory at that address (for example, the ADSP-2186 DSP), the linker generates an error for the page specified to “live” at that address.

PAGE_OUTPUT()



The `PAGE_OUTPUT()` command is supported for paged memory architectures, such as in ADSP-218x DSPs.

The `PAGE_OUTPUT()` command is used with the `PAGE_INPUT()` command to specify overlays for memory pages. The command `PAGE_OUTPUT()` can be used instead of the command `OVERLAY_OUTPUT`, in any location in the LDF file.

PLIT{}

Whenever a symbol in its input file(s) resolves to a function in overlay memory, the DSP program needs a procedure to read the overlay into run memory, then execute it. The linker supports the `PLIT{}` command to generate *Procedure Linkage Tables*, that can be used to implement an overlay manager in the DSP program.

What is a PLIT?

A PLIT is a template of instructions from which the linker generates code to set up the information necessary to support the DSP program's overlay manager. Every brach instruction that references a global label defined in an overlay is replaced by a call to this generated code.

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

For each overlay routine in the program, the linker builds and stores a list of PLIT instances according to that template, as it builds its executable.

PLIT Functions

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

- Return the absolute address of the resolved symbol in “live” memory: `PLIT_SYMBOL_ADDRESS`.
- Identify the overlay it must bring in and run when it encounters the symbol resolving an address in it: `PLIT_SYMBOL_OVERLAYID`.
- If necessary, return a null-terminated array of overlay IDs containing any data for the overlay function: `PLIT_DATA_OVERLAY_IDS`. Data can be overlaid, just like code.
- Save the target’s state on the stack or in memory before loading and executing an overlay function, so it continues correctly on return. Your program may not need this information saved.
- Initiate (jump to) the routine that transfers the overlay code to internal memory, given the previous information about its identity, size and location: `_OverlayManager`. “Smart” overlay managers first check whether the overlay function is already in internal memory, and avoid reloading it.

Allocating Space for PLITs

Your LDF must allocate space in memory to hold any PLITs your linker builds. Typically, that memory resides in the program code (`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 pm_code segment
.plit {} > seg_code
```

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

* Whatever you name your program code Memory Segment.

Linker Description File Reference

- There is no Input Section associated with the `.plit` Output Section: the LDF is allocating space for linker-generated routines, not containing any of your (input) data objects.
- This segment allocation does not take any parameters. You write the structure of this command (see “Syntax” below). The linker creates instances of the command for each symbol that resolves to an overlay, and stores each instance in the `.plit` Output Section (which becomes part of the program code Memory Segment).

PLIT Syntax

As noted above, you write the structure of 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 (*symbol*) local to an overlay function.

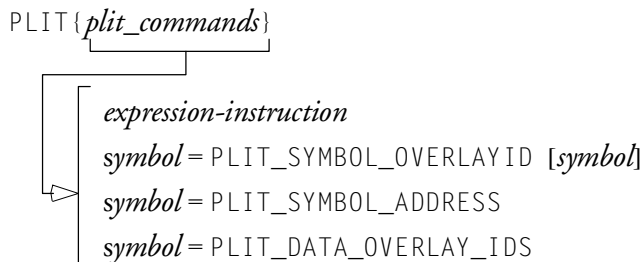


Figure 2-8. Syntax of the PLIT{} Command

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

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

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 runtime 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 runtime 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.

Linker Description File Reference

Simple PLIT—no state saved

A simple PLIT merely copies the symbol's address and overlay ID into registers, and jumps to the overlay manager, as shown in [Listing 2-6](#). This fragment was extracted from the global scope (just after the `MEMORY{}` command) of sample `fft_group.ldf`:

```
/* The global PLIT to be used whenever a PROCESSOR or OVERLAY
specific PLIT description is not provided. The plit initial-
izes a register to the overlay id and the overlay runtime
address of the symbol called. Be sure the registers used in
the plit do not contain values which cannot be overwritten.
*/

PLIT
{
    AX0 = PLIT_SYMBOL_OVERLAYID;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

Listing 2-6. A Simple PLIT{} Command

In this case, you are responsible for verifying the contents of `AX0` and `AX1` are either safe or irrelevant.

A PLIT that saves register contents

The `PLIT{}` command in the code fragment below saves the contents of `AX0` and `AX1` in data memory before being overwritten.

```
PLIT {
    DM(save_AX0) = AX0;
    DM(save_AX1) = AX1;
    AX0 = PLIT_SYMBOL_OVERLAYID;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP _OverlayManager;
}
```

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

For an example of using a PLIT, see [“Using a Procedure Linkage Table” on page 2-100](#).

PROCESSOR{}

The `PROCESSOR{}` command declares a processor and its related link information. A `PROCESSOR{}` command contains the `MEMORY{}`, `SECTIONS{}`, and other linker commands 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 of the `PROCESSOR{}` Command

- *processor_name*

Identifies the processor. Processor names follow the same rules as any linker label. [For more information, see “Linker Expression Syntax” on page 2-44](#).

Linker Description File Reference

- `OUTPUT(file_name.DXE)`

Names the executable (`.DXE`). Note that an `OUTPUT()` command in any LDF scope must appear before a `SECTIONS{}` command in that scope.

- `MEMORY{segment_commands}`

Declares memory segments local to this processor. Using LDF command scoping, you can define these segments outside the `PROCESSOR{}` command. For more information, see [“Command Scoping” on page 2-42](#), and [“MEMORY{” on page 2-58](#).

- `PLIT{plit_commands}`

Defines Procedure Linkage Table (PLIT) commands that apply only to this processor. [For more information, see “PLIT{” on page 2-72](#).

- `SECTIONS{section_commands}`

Defines sections for placement within the executable (`.DXE`). [For more information, see “SECTIONS{” on page 2-79](#).

- `RESOLVE(symbol, resolver)`

In a *section_command*, directs the linker to resolve a particular *symbol* to an address using the *resolver*. The *resolver* is an absolute address or a file (`.DXE` or `.SM`) containing the definition of the symbol.

SEARCH_DIR()

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

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

SECTIONS{}

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

Syntax

The syntax for the `SECTIONS{}` command appears in [Figure 2-10 on page 2-80](#). Definitions for the parts of the `SECTIONS{}` command's syntax are as follows:

- *section_commands* or *expressions*

Defines *expressions* or Output Sections (*section_name*). Use *expressions* to manipulate symbols or position the current location counter. Use Output Section commands to declare your program's sections. While your LDF may contain only one `SECTIONS{}` command within each `PROCESSOR's` or `SHARED_MEMORY's` scope, there is no limit to the number of sections that you can declare within each

Linker Description File Reference

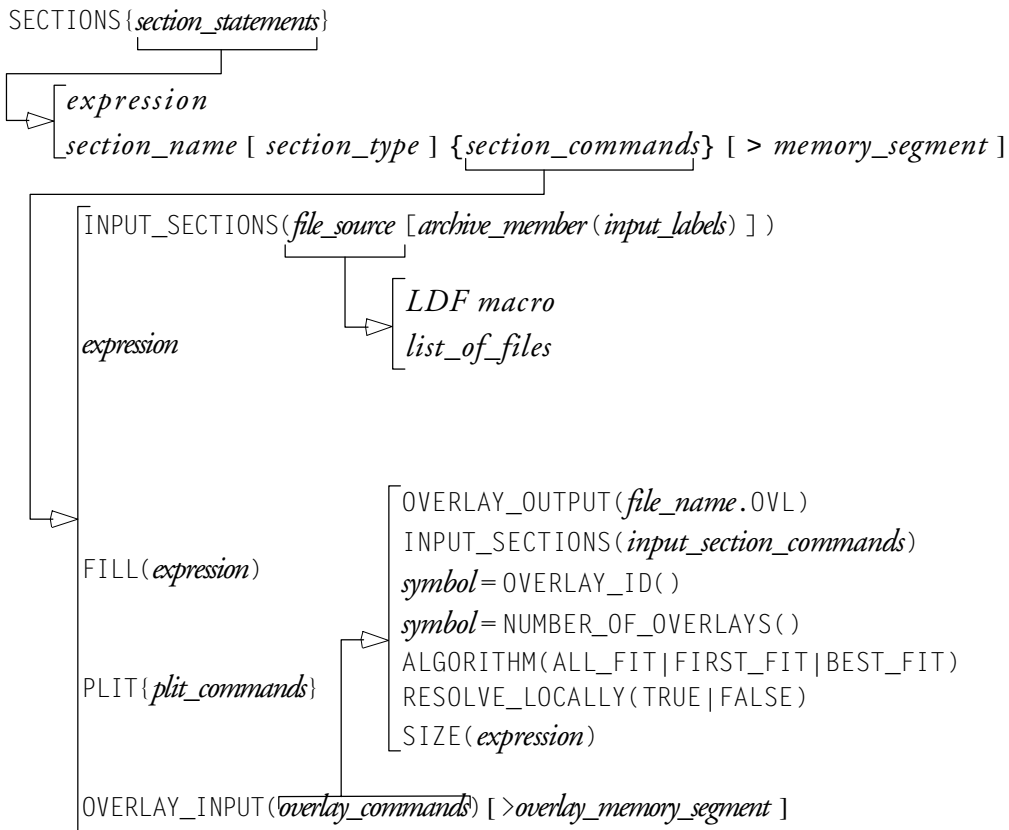


Figure 2-10. Syntax Tree of the SECTIONS{} Command

SECTIONS{} command. [For more information, see “Command Scoping” on page 2-42.](#)

An Output Section declaration has the following syntax rules:

- Its name, *section_name*, starts with a letter, underscore, or periods and may include any letters, underscores, digits, and points.
- A *section_name* must not conflict with any linker keywords.

- The special *section_name* `.plt`, indicates the Procedure Linkage Table (PLT) 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 assigns an ELF section type. The only legal section type keyword is `SHT_NOBITS`. This section type contains uninitialized data, so even if it is large, it can download quickly: space is allocated but not written. For an example of how to use `SHT_NOBITS`, see [Listing 2-8 on page 2-91](#).
- *section_commands* may contain any combination of the following commands: an `INPUT_SECTIONS()` command, an *expression*, a `FILL()` command, a `RESOLVE()` command, a `PLIT{}` command, or an `OVERLAY_INPUT()` command.
- *> memory_segment* at the end of a section definition declares that 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.

Linker Description File Reference

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

- *file_source* may be a list of files or any LDF macro that expands into a file list, such as `$COMMAND_LINE_OBJECTS`. The list may contain object or archive files. Use commas to delimit files within the list.
- *archive_member* names the source-object file within an archive. The *archive_member* parameter and the left/right brackets, `[ ]`, are only required if the *file_source* of the *input_label* is an archive.
- *input_labels* come from the run-time `.SECTION` names in your assembly program. Use commas to delimit `.SECTION` names within the list. The linker places `.SECTIONS` in the order that you list them.

- *expression*

Manipulates symbols or positions the current location counter, specified by a period.

- `FILL(expression)`

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

- `PLIT{plit_commands}`

In a *section_command* declares a locally^{*} scoped Procedure Linkage Table (PLIT). It contain its own labels and expressions. [For more information, see “PLIT{}” on page 2-72.](#)

^{*} In that section only.

- `OVERLAY_INPUT(overlay_commands)`

In a *section_command*, identifies parts of your program to place in an overlay executable with *overlay_commands*. [For more information, see “Linking for Overlay Memory” on page 2-96.](#)

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

 Within the `OVERLAY_INPUT()` command, you can only increment the current location counter.

overlay_memory_segment is optional. It determines whether the section is placed in an overlay segment. Some overlay sections, such as those loaded from a host, do not need to be included in the overlay memory image of the executable, 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 a overlay file (`.OVL`) for the overlay with the specified name. Note that a `OVERLAY_OUTPUT()` command in an `OVERLAY_INPUT()` command must appear before any `INPUT_SECTIONS()` for that overlay.
- The `INPUT_SECTIONS()` command has the same syntax within an `OVERLAY_INPUT()` command as when it appears within a *output_section_command* except that you may not place the `.plt` section in overlay memory. [For more information, see “INPUT_SECTIONS\(\)” on page 2-81.](#)

Linker Description File Reference

- 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 runtime 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 runtime memory segment, according to First-In-First-Out (FIFO) order.

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

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

$$\text{Next ID} = \text{OVERLAY_ID}() + \text{NUMBER_OF_OVERLAYS}()$$

- 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 may be occupied by an overlay.

SHARED_MEMORY{}



The `SHARED_MEMORY{ }` command is used with ADSP-219x DSPs ONLY.

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 multiprocessor 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 in your LDF, the linker cannot link your program.

Your LDF may contain multiple `SHARED_MEMORY{ }` commands, but the maximum number of processors that can access a shared memory is processor-specific.

The `SHARED_MEMORY{ }` command must appear within the LDF scope of a `MEMORY{ }` command that describes the shared memory. The `PROCESSOR` commands declaring the processors that share this memory must also appear within this LDF scope.

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

Linker Description File Reference

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

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

- 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 2-79.](#)

An example of this command scoping appears in [Figure 2-12 on page 2-87](#). [For more information, see “Command Scoping” on page 2-42.](#)

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



```

MEMORY
{
    my_shared_ram
    {
        TYPE(PM RAM) START(5120k) LENGTH(8k) WIDTH(32)
    }
}

SHARED_MEMORY
{
    OUTPUT(shared.sm)

    SECTIONS
    {
        my_shared_sections{section_commands}
        > my_shared_ram
    }
}

PROCESSOR p0{
    processor_commands with link against shared memory}
PROCESSOR p1{
    processor_commands with link against shared memory}

```

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

LDF Programming Examples

This section shows LDFs for several typical system models. As you modify these examples, refer to the syntax descriptions in “LDF Commands” on page 2-53. This section provides the following examples:

- “Linking for Single Processor Memory” on page 2-89
- “Linking Large Uninitialized Variables” on page 2-90
- “Linking for Multi-Processor and Shared Memory” on page 2-91
- “Linking for Overlay Memory” on page 2-96
- “Using a Procedure Linkage Table” on page 2-100
- “Using An Overlay Memory Manager” on page 2-104
- “Managing Overlays” on page 2-106
- “Managing Two Overlays (Example)” on page 2-112
- “Reducing Overlay Manager Overhead (Example)” on page 2-118



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

```
$ADI_DSP\218x\examples  
$ADI_DSP\219x\examples
```



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

```
$ADI_DSP\218x\ldf  
$ADI_DSP\219x\ldf
```

Linking for Single Processor Memory

When linking an executable for a single processor system, the LDF describes the processor's memory and places code for that processor. The LDF in [Listing 2-7](#) shows a single processor LDF. Note the following commands in this LDF:

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

Listing 2-7. Single-Processor System LDF Example

```
ARCHITECTURE(ADSP-219x) // Link for the ADSP-219x

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

// $ADI_DSP is a predefined linker macro that expands
// to the VDSP install directory. Search for objects in
// directory 219x/lib relative to the install directory

SEARCH_DIR( $ADI_DSP\219x\lib )

// 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
```

LDF Programming Examples

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

// 219x_hdr.doj: standard initialization file for 219x DSPs
$OBS = $COMMAND_LINE_OBJECTS, 219x_hdr.doj;

//      A linker project to generate a DXE file

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

    MEMORY                // Processor specific memory command
    { INCLUDE( "219x_memory.ldf" ) }*

    SECTIONS {            // Specify the Output Sections
    {
        INCLUDE( "219x_sections.ldf" )
    }                    // end P0 sections
    }                    // end P0 processor
```

Linking Large Uninitialized Variables

When linking an executable file that contains large uninitialized variables, you can reduce the size of the file by using the `SHT_NOBITS` section qualifier.

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 be omitted from the output file. The Output Section type `SHT_NOBITS` directs the linker to omit data for that section from the output file.

* This included LDF consists of Memory Segment declarations, as shown in [Listing 2-2 on page 2-20](#).

[Listing 2-9](#) shows an example using the SHT_NOBITS section to avoid initialization of a segment.

Listing 2-8. Large Uninitialized Variables: Assembly Source

```
SECTION/DATA      extram_area;    /* 1Mx16 EXTRAM */
.VAR huge_bufffer[0x006000];
```

Listing 2-9. Large Uninitialized Variables: LDF Source

```
ARCHITECTURE(ADSP-219x)

$OBJECTS = $COMMAND_LINE_OBJECTS; // Libraries & objects from
                                   // the command line

MEMORY {
    mem_extram {
        TYPE(DM RAM) START(0x10000) END(0x15fff) WIDTH(16)
    } // end segment
} // end memory

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

Linking for Multi-Processor and Shared Memory

When linking executable files for multiprocessor memory or a shared memory system, the LDF describes the shared memory and each proces-

LDF Programming Examples

processor's separate memory (with offsets), and places code for each processor.

[Listing 2-10](#) shows a multiprocessor memory and shared memory LDF.

- `ARCHITECTURE()` defines the processor type (only one processor type can be defined within an LDF)
- `SEARCH_DIR()` commands add the `lib` and current working directory to the search path
- `$OBJ`s and `$LIBS` macros get object (`.DOJ`) and library (`.DLB`) file input
- `MPMEMORY{}` defines each processor's offset within multiprocessor memory
- `SHARED_MEMORY{}` identifies the output for the shared memory items
- `MAP()` commands output 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 2-10. Multi Processor System LDF Example

```

ARCHITECTURE(ADSP-219x)
SEARCH_DIR( $ADI_DSP\219x\lib )

// Multiprocessor memory space is allocated with the MPMEMORY
// command. The values represent an “addend” that the linker
// uses when it resolves undefined symbols in one DXE to symbols
// defined in another DXE. The addend is added to each defined
// symbol’s value.
// E.g. PROCESSOR project PSH0 contains the undefined symbol
// “buffer”, PROCESSOR project PSH1 defines “buffer” at
// address 0x22000. The linker will “fix up” the reference to
// “buffer” in PSH0’s code to address:
//      0x22000 + MPMEMORY(PSH1) = 0x22000 + 0x280000= 0x2a2000
MPMEMORY
{
    PSH0 { START (0x200000) }
    PSH1 { START (0x280000) }
}

MEMORY {
    // Used for all processors. Alternatively, a
    seg_reset // PROCESSOR could describe its own MEMORY
    { TYPE(PM RAM) START(0x00000) END(0x00004) WIDTH(24) }
    seg_itab
    { TYPE(PM RAM) START(0x000004) END(0x0000ff) WIDTH(24) }
    seg_code
    { TYPE(PM RAM) START(0x000100) END(0x00ffff) WIDTH(24) }
    seg_dmda
    { TYPE(DM RAM) START(0x010000) END(0x017fff) WIDTH(16) }
    seg_data2
    { TYPE(PM RAM) START(0x018000) END(0x01ebff) WIDTH(24) }
    seg_heap
    { TYPE(DM RAM) START(0x01ec00) END(0x01efff) WIDTH(16) }
    seg_stack
    { TYPE(DM RAM) START(0x01f000) END(0x01ffff) WIDTH(16) }
}

$LIBRARIES = lib219x.dlb, libc.dlb;

// This LDF specifies three link projects. The first is a
// shared memory project against which the PROCESSOR projects
// will be linked. The file containing the shared data

```

LDF Programming Examples

```
// buffers is defined in shared.c
SHARED_MEMORY {
    $SHARED_OBJECTS = shared.doj;

    // The output name of this shared object is subsequently
    // used in the PROCESSOR projects' LINK_AGAINST command
    OUTPUT(shared.sm)

    //shared.c has only data declarations. No need to
    //specify any Output Section other than "seg_dmda"
    SECTIONS {
        seg_dmda {INPUT_SECTIONS( $SHARED_OBJECTS(seg_dmda))
                } > seg_dmda
    }
}

// The second link project is a DXE project. It will be linked
// against the SHARED link project defined above.
PROCESSOR PSH0 {
    $PSH0_OBJECTS = psh0.doj, 219x_hdr.doj;
    LINK_AGAINST(shared.sm)
    OUTPUT( psh0.dxe )

    SECTIONS {
        dxe_pmco
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_pmco)
                $LIBRARIES(seg_pmco)) } > seg_pmco
        dxe_pmda
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_pmda)
                $LIBRARIES(seg_pmda)) } > seg_pmda
        dxe_dmda
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_dmda)
                $LIBRARIES(seg_dmda)) } > seg_dmda
        dxe_init
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_init)
                $LIBRARIES(seg_init)) } > seg_init
        dxe_rth
            {INPUT_SECTIONS($PSH0_OBJECTS(seg_rth)
                $LIBRARIES(seg_rth)) } > seg_rth
        stackseg { // allocate a stack for the application
            ldf_stack_space = .;
            ldf_stack_length = 0x2000; } > seg_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;
    } > seg_heap
}

// The last project defined in this LDF is another DXE
// project. This PROCESSOR project will be linked against both
// the SHARED and the PSH0 DXE projects defined above.
PROCESSOR PSH1 {
    $PSH1_OBJECTS = psh1.doj, 219x_hdr.doj;
    LINK_AGAINST(shared.sm, psh0.dxe)
    OUTPUT(psh1.dxe)
    SECTIONS {
        dxe_pmco
            { INPUT_SECTIONS( $PSH1_OBJECTS(seg_pmco)
                $LIBRARIES(seg_pmco)) } > seg_pmco
        dxe_pmda
            { INPUT_SECTIONS( $PSH1_OBJECTS(seg_pmda)
                $LIBRARIES(seg_pmda)) } > seg_pmda
        dxe_dmda
            { INPUT_SECTIONS( $PSH1_OBJECTS(seg_dmda)
                $LIBRARIES(seg_dmda)) } > seg_dmda
        dxe_init
            { INPUT_SECTIONS( $PSH1_OBJECTS(seg_init)
                $LIBRARIES(seg_init)) } > seg_init
        dxe_rth
            { INPUT_SECTIONS( $PSH1_OBJECTS(seg_rth)
                $LIBRARIES(seg_rth)) } > seg_rth
        stringstab
            { INPUT_SECTIONS( $PSH1_OBJECTS(.stringstab)
                $LIBRARIES(.stringstab)) }
        SDB
            { INPUT_SECTIONS( $PSH1_OBJECTS(.SDB)
                $LIBRARIES(.SDB)) }
        lno_seg_pmco
            { INPUT_SECTIONS($PSH1_OBJECTS(.lno_seg_pmco)
                $LIBRARIES(.lno_seg_pmco)) }
        stackseg { // stack for the application
            ldf_stack_space = .;
            ldf_stack_length = 0x2000; } > seg_stak
        heap { // heap for the application
            ldf_heap_space = .;
            ldf_heap_end = ldf_heap_space + 0x2000;

```

LDF Programming Examples

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

Linking for Overlay Memory

When linking executable files for an overlay memory system, the LDF describes the overlay memory, the processor(s) that use the overlay memory, and each processor's unique memory. The LDF places code for each processor and the special PLIT{} section. [Listing 2-11](#) shows an overlay memory LDF with explanatory comments.

Listing 2-11. Overlay-Memory System LDF Example

```
ARCHITECTURE(ADSP-219x)
SEARCH_DIR( $ADI_DSP\219x\lib )

MAP(overlay.map)
// This simple example uses internal memory for overlays
// (Real overlays would never "live" in internal memory)
MEMORY {
    seg_reset { TYPE(PM RAM) START(0x000000) END(0x000004) WIDTH(24) }
    seg_itab { TYPE(PM RAM) START(0x000004) END(0x0000ff) WIDTH(24) }
    seg_code { TYPE(PM RAM) START(0x000100) END(0x00ffff) WIDTH(24) }
    seg_data1 { TYPE(DM RAM) START(0x010000) END(0x017fff) WIDTH(16) }
    seg_data2 { TYPE(PM RAM) START(0x018000) END(0x01ebff) WIDTH(24) }
    seg_heap { TYPE(DM RAM) START(0x01ec00) END(0x01efff) WIDTH(16) }
    seg_stack { TYPE(DM RAM) START(0x01f000) END(0x01ffff) WIDTH(16) }
}

PROCESSOR p0 {
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS
    { dx_e_reset { INPUT_SECTIONS($OBJECTS(IVreset)) } >seg_reset
      dx_e_itab { INPUT_SECTIONS($OBJECTS(IVpwrdsn))
                  INPUT_SECTIONS($OBJECTS(IVsinglestep))
                  INPUT_SECTIONS($OBJECTS(IVstackint))
                  INPUT_SECTIONS($OBJECTS(IVint4))
                  INPUT_SECTIONS($OBJECTS(IVint5))
                  INPUT_SECTIONS($OBJECTS(IVint6))
                  INPUT_SECTIONS($OBJECTS(IVint7))
```

```

        INPUT_SECTIONS($OBJECTS(Ivint8))
        INPUT_SECTIONS($OBJECTS(Ivint9))
        INPUT_SECTIONS($OBJECTS(Ivint10))
        INPUT_SECTIONS($OBJECTS(Ivint11))
        INPUT_SECTIONS($OBJECTS(Ivint12))
        INPUT_SECTIONS($OBJECTS(Ivint13))
        INPUT_SECTIONS($OBJECTS(Ivint14))
        INPUT_SECTIONS($OBJECTS(Ivint15))
    }
    dxec_code { INPUT_SECTIONS($OBJECTS(program)) } >seg_itab
    dxec_data1 { INPUT_SECTIONS($OBJECTS(data1)) } >seg_code
    dxec_data2 { INPUT_SECTIONS($OBJECTS(data2)) } >seg_data1
    dxec_data3 { INPUT_SECTIONS($OBJECTS(data3)) } >seg_data2

// support for initialization
dxec_ctor { INPUT_SECTIONS($OBJECTS(ctor)) } >seg_data1

// provide linker variables describing the stack (grows down)
// ldf_stack_limit is the lowest address in the stack
// ldf_stack_base is the highest address in the stack
dxec_stack { ldf_stack_limit = .;
    ldf_stack_base = . + MEMORY_SIZEOF(seg_stack) - 1; >seg_stack

    dxec_heap { .heap = .;
        .heap_size = MEMORY_SIZEOF(seg_heap);
        .heap_end = . + MEMORY_SIZEOF(seg_heap) - 1; } >seg_heap
    }
} //end of SECTIONS command
//end of PROCESSOR p0 (END OF LDF)

//
// Processor and application specific assembly language instructions,
// generated for each symbol that is resolved in overlay memory.
PLIT {
    AX0 = PLIT_SYMBOL_OVERLAYID; // overlay ID of the resolved symbol
    AX1 = PLIT_SYMBOL_ADDRESS; //run address of the resolved symbol
    dm(_overlayID) = AX0;
    dm(_pf) = AX1;
    JUMP _OverlayManager;
}

$LIBRARIES = lib219x.dlb, libc.dlb;
$OBJECTS = 219x_hdr.doj;

PROCESSOR P0 {
    $P0_OBJECTS = main.doj , manager.doj;
    OUTPUT(mgrovly.dxe)
    SECTIONS {
        // .text Output Section
        seg_code {

```

LDF Programming Examples

```
        INPUT_SECTIONS(  
            $PO_OBJECTS(seg_code) $LIBRARIES(seg_code))  
  
    //Specify the first overlay. This overlay will “live” in the memory  
    // defined by “seg_ovly”. It will run in the memory space defined  
    // by “seg_code”  
  
    OVERLAY_INPUT {  
        // The output archive file “overlay1.ovl” will  
        // contain the code and symbol table for this  
        // overlay.  
  
    OVERLAY_OUTPUT( overlay1.ovl )  
  
        // Only take the code from the file overlay1.doj.  
        // If this code needs data, it must be either the INPUT of a data  
        // overlay or the INPUT to non-overlay data memory.  
        INPUT_SECTIONS(overlay1.doj( seg_code))  
  
        // Tell the linker that all of the code must fit into  
        // the “run” memory all at once. ALGORITHM(FIRST_FIT)  
        // would allow the linker to break the code into  
        // several overlays as necessary (in the event that  
        // not all of the code fits)  
  
        ALGORITHM( ALL_FIT )  
        SIZE(0x100)  
    } > seg_ovly  
  
    // This is the second overlay. Note that these  
    // OVERLAY_INPUT commands must be contiguous in the LDF  
    // in order for them to occupy the same “runtime” memory.  
    OVERLAY_INPUT {  
        OVERLAY_OUTPUT( overlay2.ovl )  
        INPUT_SECTIONS(overlay2.doj( seg_pmco))  
        ALGORITHM( ALL_FIT )  
        SIZE( 0x100)  
    } > seg_ovly  
  
} > dxm_pmco  
  
    // The instructions generated by the linker in the .plit section  
    // must be placed in non-overlay memory. Here is the sole spec-  
    // ification telling the linker where to place these instructions  
    .plit {} > seg_pmco  
  
dxm_pmda {
```

```

        INPUT_SECTIONS(
            $PO_OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))) > seg_pmda
dxe_gmda {
    INPUT_SECTIONS( $PO_OBJECTS(seg_dmda)
        $LIBRARIES(seg_dmda))
    } > seg_dmda

    .seg_init {
        INPUT_SECTIONS(
            $PO_OBJECTS(seg_init) $LIBRARIES(seg_init))
    } > seg_init

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

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

```

Using a Procedure Linkage Table

The linker resolves function calls, both direct and indirect, across overlays. This support implies that the linker generates extra code in order to transfer control to a user defined routine (an overlay manager) that handles the loading of overlays. The linker-generated overlay manager code goes in a special section of the executable, which has the section name `.plit`. An LDF for an application that uses PLITs must contain a linker command to place the generated section `.plit`.

For more details, see the description of “[PLIT{}](#)” on page 2-72.

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 2-101](#) shows the interaction between a PLIT and an overlay manager. To make this kind of interaction possible, the linker generates some special symbols for overlays. These overlay symbols include: `_ov_startaddress_#`, `_ov_endaddress_#`, `_ov_size_#`, and `_ov_runtimestartaddress_#`. The `#` in these symbols indicates the overlay number.

In [Figure 2-13 on page 2-101](#), the two functions are on different overlays. By default, the linker generates PLIT code only when an unresolved function reference is resolved to a function definition in overlay memory.

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

Non-Overlay Memory

```

main()
{
    int (*pf)() = X;
    Y();
}

/* PLIT & software-manager handle calls,
using the PLIT for resolving calls and
loading overlays as needed */

.plt_X: call OM

.plt_Y: call OM

```

```

Overlay 1 Storage  X() {...}           // function X defined
Overlay 2 Storage  Y() { ... }        // function Y defined

```

```

Runtime Overlay Memory           // currently loaded overlay

```

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

Using PLITs lets you resolve inter-overlay calls, as shown in [Figure 2-14 on page 2-102](#). You should structure your LDF so the PLIT code that the linker generates for inter-overlay function references is part of the `.plt` section for `main()`, which is stored in non-overlay memory.

i The `.plt` 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 2-103](#), for architectures in which a processor (or processors) have access to the memory of another processor(s). When one processor makes a call into another processor's overlay, the call increases the size of the `.plt` section in the executable that manages the overlay.

LDF Programming Examples

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 referenced.

Non-Overlay Memory

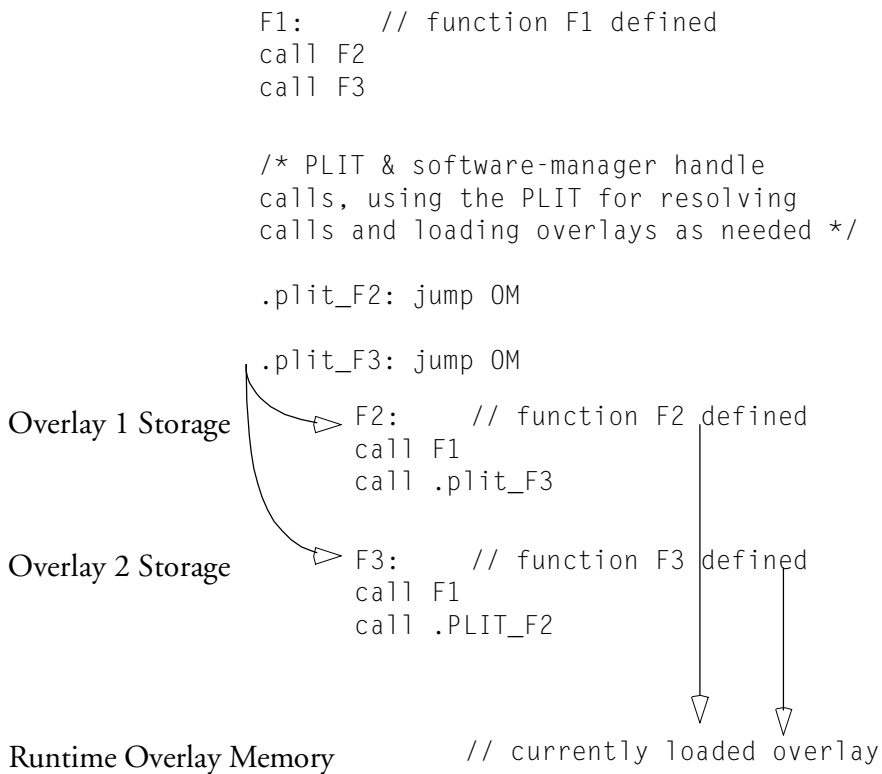


Figure 2-14. PLITs & Overlay Memory; Inter-Overlay Calls

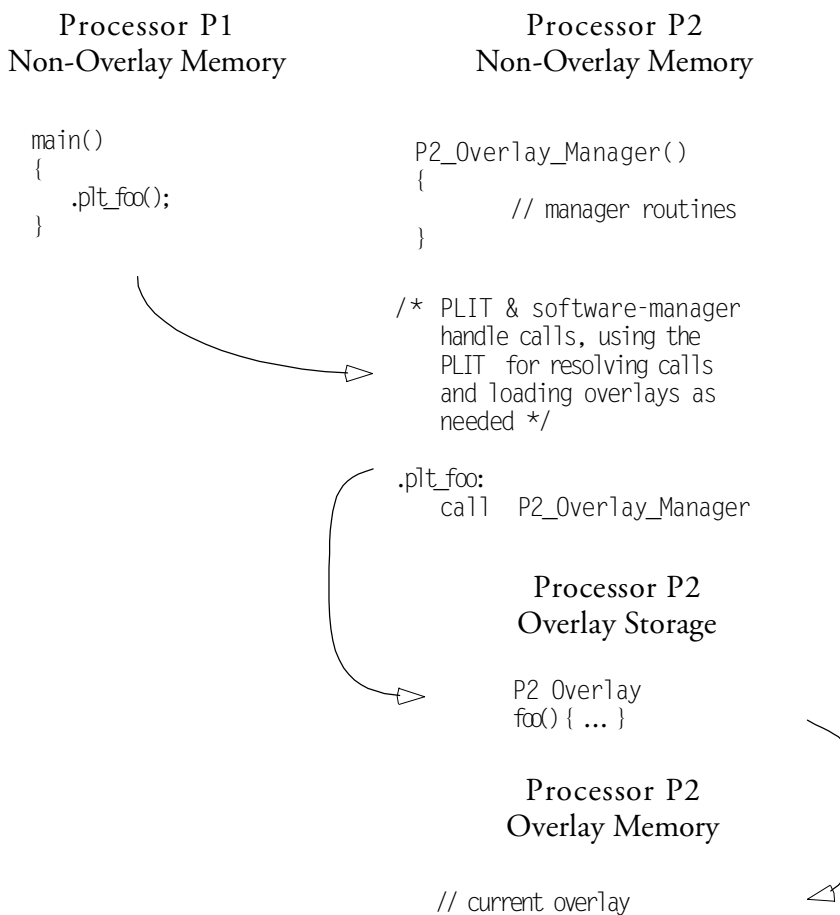


Figure 2-15. PLITs & Overlay Memory; Inter-Processor Calls

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 DSPs, including the following topics and examples:

- [“Managing Overlays” on page 2-106](#)
- [“Managing Two Overlays \(Example\)” on page 2-112](#)
- [“Reducing Overlay Manager Overhead \(Example\)” on page 2-118](#)

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

[Figure 2-16 on page 2-105](#) demonstrates the concept of memory overlays. In it, 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.

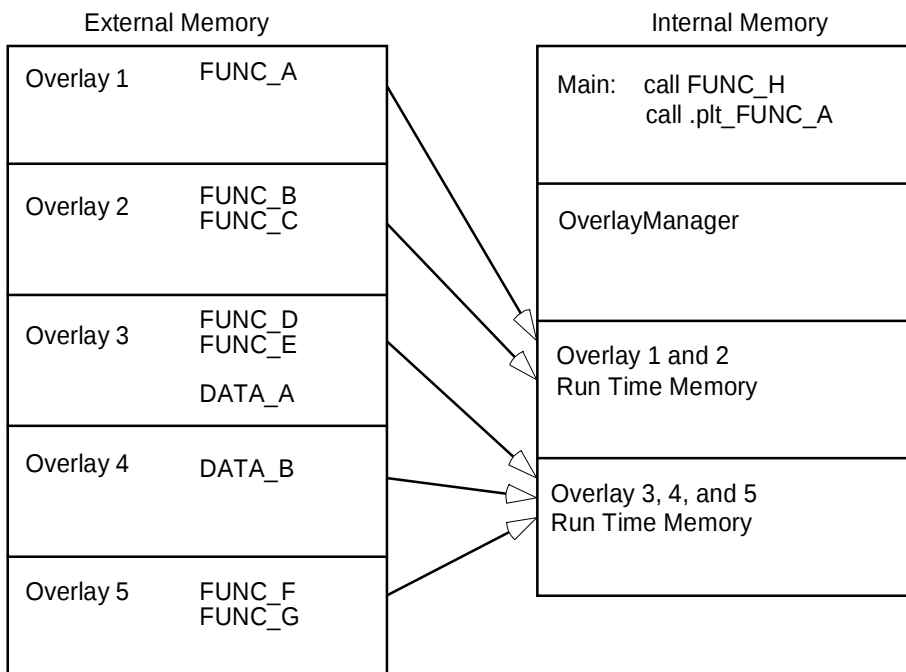


Figure 2-16. Memory Overlays

In this example, Overlay 1 and 2 share the same run-time location within the 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 via DMA.

LDF Programming Examples

The overlay manager may 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 tools provide the following overlay support:

- Specification of the live and run location of each overlay
- Generation of constants
- Redirection of overlay function calls to a jump table
- The overlay manager.

Overlay support is partially designed by the user. The user specifies which overlays share run time memory and which memory segments establish the live and run space. [Listing 2-12](#) shows the section of an LDF defining two overlays

Listing 2-12. Overlay Declaration in LDF

```
.pm_code
{
    OVERLAY_INPUT
        OVERLAY_OUTPUT(OVLY_one.ovl)
        INPUT_SECTIONS(FUNC_A.doj(pm_code))
}>ovl_code

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

The overlay declaration in [Listing 2-12](#) configures two overlays to share a common run time memory space.

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

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

The LDF tells the linker how to configure the overlays, and provides the information necessary for the overlay manager routine to load the overlays. The linker uses this information to generate the following constants:

Listing 2-13. Linker Generated Overlay Constants

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

Each overlay has a word size and an address, which the overlay manager uses to determine where the overlay resides and where it is executed.

The overlay live and run word sizes are different if the internal and external memory widths are different. For example, the instruction word width of the ADSP-21xx DSP is 24 bits. A system containing 16-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 16-bit words required to pack all of the 24-bit instructions.

LDF Programming Examples

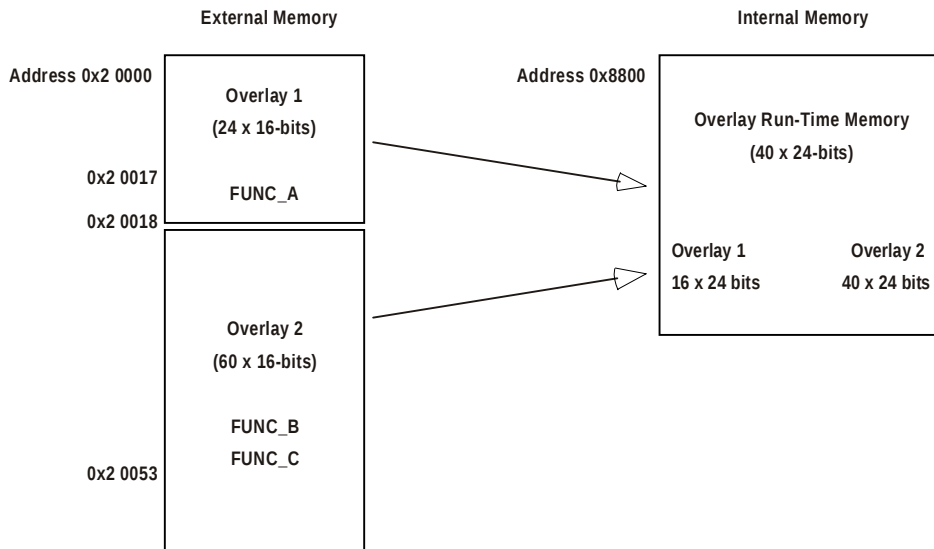


Figure 2-17. Example Overlay Live and Run Memory Sizes

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 24 bits. Because external memory is 16 bits, their **live** word width is 16 bits.
- Overlay 1 contains one function with 16 instructions. Overlay 2 contains two functions with a total of 40 instructions.
- The **live** word sizes for overlays 1 and 2 are 24 and 60 words.
- The **run** word sizes for overlays 1 and 2 are 16 and 40 words.

Listing 2-14. Linker-Generated Constants

```

_ov_startaddress_1 = 0x20000    _ov_startaddress_2 = 0x20018
_ov_endaddress_1 = 0x20017    _ov_endaddress_2 = 0x20J3
_ov_word_run_size_1 = 0x10    _ov_word_run_size_2 = 0x28
_ov_word_live_size_1 = 0x18    _ov_word_live_size_2 = 0x3C
_ov_runtimestartaddress_1 = 0x8800
_ov_runtimestartaddress_2 = 0x8800

```

[Listing 2-15](#) is a sample PLIT definition from an LDF.

Listing 2-15. PLIT Definition in LDF

```

PLIT
{
    AX0 = PLIT_SYMBOL_ID;
    AX1 = PLIT_SYMBOL_ADDRESS;
    JUMP_OverlayManager
}

```

In the example, the register `AX0` is set to the value of the overlay ID that contains the referenced symbol, and register `AX1` is set to the run time address of the referenced symbol. (`PLIT_SYMBOL_OVERLAY_ID` and `PLIT_SYMBOL_ADDRESS` are linker keywords). The last instruction branches to the overlay manager. The overlay manager uses the initialized registers to determine which overlay to load, and where to jump to execute the overlay function called.

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

LDF Programming Examples

The following code shows the value of all constants generated by the linker for the example in Figure 2-17 and based on definition in Listing 2-15:

Overlay 1 FUNC_A	Overlay 2 FUNC_B FUNC_C
---------------------	-------------------------------

Internal Memory

Main:	Plit_table	
call .plt_FUNC_A	.plt_FUNC_A:	ax0=0x0000; ax1=0x8800; jump OverlayManager;
.		
.		
.	.plt_FUNC_B:	ax0=0x0002; ax1=0x8800; jump OverlayManager;
call .plt_FUNC_C		
call .plt_FUNC_B		
.	.plt_FUNC_C:	ax0=0x0002; ax1=0x8814; jump OverlayManager;
.		

Figure 2-18. Expanded PLIT Table.

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

Given an instruction sequence including a call to a function in an overlay:

```
AX0 = DM(I0,M3);  
AX1 = AX0 * MR2;  
CALL FUNC_A;          /* Call to function in overlay */  
DM(I3,M3) = AX1;
```

The linker replaces the function call with the following instruction:

```
AX0 = DM(I0,M3);  
AX1 = AX0 * MR2;  
CALL .plt_FUNC_A; / * Call to PLIT entry */  
DM(I3,M3) = AX1;
```

`.plt_FUNC_A` is the entry in the PLIT containing the overlay-handling instructions for `FUNC_A`. These instructions direct the overlay manager to load the overlay containing `FUNC_A`. The instructions executed in the PLIT are specified within the LDF.

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 on page 2-110](#). 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 on page 2-110](#) 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, `AX0` contains the referenced function's overlay ID, and `AX1` 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.

LDF Programming Examples

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

- Set up a stack to store register values.

In some cases stacks may be corrupted by the overlay.

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

If the overlay is already in internal memory, the overlay transfer is bypassed and 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 (Example)

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`.

The overlay manager performs the following functions:

1. creates and maintains a stack for the registers it uses
2. determines if the referenced function is in internal memory
3. sets up a DMA transfer
4. flushes the cache and executes the referenced function

The following code segment from the LDF represents the overlay definitions of Example 1. Several code segments for the LDF and the Overlay Manager are displayed and explained in the text.

Listing 2-16. FFT Overlay Example 1

```
OVERLAY_INPUT
{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_1st_last.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_c
```

Two overlays are defined: `fft_one.ovl` and `fft_two.ovl`. Both overlays live in segment `ovl_code` (defined in the `MEMORY{}` command), and run in section `pm_code`. 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 `pm_code` 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 is referenced during code execution, `fft_one`, overlay `id=1` is transferred to internal memory. When the second function is referenced, `fft_two`, overlay `id=2` is transferred to internal memory. Since the third function is in overlay `fft_two`, when it is referenced the overlay manager recognizes that it is already in internal memory and an overlay transfer does not occur.



To verify whether an overlay is already in internal memory, place the overlay ID of each overlay already loaded and the overlay to be loaded in registers, for example AY0 and AY1, and compare:

```
AY0 = AY0 - AY1; /* Is overlay already in internal memory? */  
if EQ jump skipped_DMA_setup;  
/* If so, bypass transferring it in.*/
```

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

```
fftrad2:  
    call fft_first_2_stages (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  
{  
    AX0 = PLIT_SYMBOL_OVERLAYID;  
    AX1 = PLIT_SYMBOL_ADDRESS;  
    JUMP _OverlayManager;  
}
```

Register `AX0` contains the overlay ID that contains the referenced symbol and register `AX1` 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 `AX1`.

Linker-Generated Constants

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

```
.EXTERN _ov_word_run_size_1;
.EXTERN _ov_word_run_size_2;
.EXTERN _ov_word_live_size_1;
.EXTERN _ov_word_live_size_2;
.EXTERN _ov_startaddress_1;
.EXTERN _ov_startaddress_2;
.EXTERN _ov_runtimestartaddress_1;
.EXTERN _ov_runtimestartaddress_2;
```

These constants provide the following information to the overlay manager:

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

Overlay Manager Operations

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

LDF Programming Examples

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 is defined as AX0, set in the PLIT of LDF.
   Set up DMA transfer to internal memory. Store values of
   registers used by the overlay manager on the stack. */

dm(ov_stack)=i1;
dm(ov_stack+1)=m3;
dm(ov_stack+2)=AF;
dm(ov_stack+3)=MR2;

/* Use the overlay id as an index (must subtract one) */
AX0=AX0-1;          /* Overlay ID -1 */
m3=AX0;             /* Offset into the arrays containing linker
                    defined overlay constants. */

MR2=dm(ov_id_loaded);
AX0=AX0-MR2;
if EQ jump continue;
dm(ov_id_loaded)=m3;

AX0=i0;             dm(ov_stack+4)=AX0;
AX0=m0;             dm(ov_stack+5)=AX0;
AX0=l0;             dm(ov_stack+6)=AX0;
```

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 `i1` and `I7` point to the first location of the arrays. The overlay ID is stored in the modify registers `m3` 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.                                */
i1 = runAddresses;
i0 = liveAddresses;

AX0=0;                /* Disable DMA */
dm(DM_ADDR) = AX0;

AX0=dm(m0,i0); /* Set DMA external pointer to overlay */
dm(DMOVLY)=AX0; /* live address */

AX0=pm(m3,i1); /* Set DMA internal pointer to overlay */
dm(PMOVLY)=AX0; /* run address */

i0=runWordSize; /* # of words stored in internal memory */
/* Word size is 24 bits for instructions */

AX0=1;           /* Set DMA external modifier */
dm(IDMAD)=AX0;

i1=liveWordSize; /* # of words stored in external memory. */
/* Most likely word size is 16 bits. */

dm(IDMAA)=AX0; /* Set DMA internal modify to 1 */

AX0=dm(m0,i0); /* Set DMA internal count to Overlay */
dm(PUCR)=AX0; /* run size */

AX0=pm(m3,i1); /* Set DMA external count to Overlay */
dm(ASTAT)=AX0; /* run size */

/* DMA enabled, instruction word, Master, 16-24 packing */
AX0=0x2e1;
dm(DM_ADDR)=AX0;

```

LDF Programming Examples

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

In summary, the overlay manager routine performs the following tasks:

1. maintains a run-time stack for registers being used by the overlay manager
2. compares the requested overlay's ID with that of the previously loaded overlay (stored in buffer `ov_id_loaded`)
3. sets up the DMA transfer of the overlay (if it is not already in internal memory)
4. jumps the PC to the run time location of the referenced function.

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

Reducing Overlay Manager Overhead (Example)

This 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.

This example uses the concept of overlay function loading and executing. A function **load** is a request to load the overlay function into internal memory but not execute the function. A function **execution** is a request to

execute an overlay function that may or may not be in internal memory at the time of the execution request. If the function is not in internal memory a transfer must occur before execution.

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

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

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

LDF Programming Examples

instructions immediately following the location of the function load reference.

Every overlay function call requires initializing the load/execute flag buffer. Here, the function calls are delayed branch calls. The two slots in the delayed branch contain instructions to initialize the flag buffer. Register AX0 is set to the value that is placed in the flag buffer, and the value in AX0 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);
    AX0=0;
    dm(prefetch) = AX0;
call fft_middle_stages (db);
    AX0=0;
    dm(prefetch) = AX0;
call fft_next_to_last (db);
    AX0=0;
    dm(prefetch) = AX0;
call fft_last_stage (db);
    AX0=0;
    dm(prefetch) = AX0;
```

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. */
AX0=1;
    dm(prefetch) = AX0;
    /* Set pre-fetch flag to 1 to indicate a load. */
```

The implementation executes the first function and transfers the second function and so on. In this implementation each function resides in a unique overlay and requires reserving two run time locations; while one

overlay is loading into one run time location, a second overlay function is executing in another run time location.

The following code segment allocates the functions to overlays and forces two runtime locations.

```

OVERLAY_INPUT
{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_one.ovl)
    INPUT_SECTIONS( Fft_ovl.doj(pm_code) )
    PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)
} >ovl_code // Overlay to live in section ovl_code

OVERLAY_INPUT
{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_three.ovl)
    INPUT_SECTIONS( Fft_ovl.doj(pm1_code) )
    PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)
} >ovl_code // 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(pm3_code) )
    PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)
} >ovl_code // Overlay to live in section ovl_code
OVERLAY_INPUT
{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(fft_last.ovl)
    INPUT_SECTIONS( Fft_ovl.doj(pm2_code) )
    PACKING( 6 B1 B2 B3 B0 B4 B5 B6 B0)
} >ovl_code // Overlay to live in section ovl_code

```

The first and third overlays share one runtime location and the second and fourth overlays share the second runtime location. By placing an Input

LDF Programming Examples

Section between overlay declarations, multiple runtime locations are allocated.

The overlay manager requires modification from that of example one. 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 runtime 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 complete overlay manager function is shown in the Example 2 section. The overlay manager pushes the appropriate registers on the runtime 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 runtime 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 runtime stack.

Linker Glossary

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

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

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

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.

Linker Glossary

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