Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Converting From Legacy Architecture Files To Linker Description Files for the ADSP-218x

Last modified: March 12, 2001

Introduction

This application note will explain how to describe an ADSP-218x hardware memory architecture using the latest version of the ADSP-218x development tools, VisualDSP version 7.0. We will go into excruciating detail to help explain the differences between the use of linker description files (.LDF) which are used by the VisualDSP toolset to describe the system's hardware memory architecture, and the legacy system (.SYS) and architecture (.ACH) files, which were supported by the 5.x and 6.x ADSP-2100 family development tools.

We'll begin this engineering note with a generic 218x example, showing the differences between the 7.0 and the 5.x/6.x tools versions. We'll then provide another example showing how hardware overlays are supported between the different tools versions. *The latest VisualDSP development tools (7.0) provide enhanced support for hardware memory overlays over the 5.x and 6.x development tools. At the current time in which this engineering note was written, the 7.0 tools version only supports internal hardware overlays; support for software overlays will be provided in an upcoming service pack release. Please check our website at the following address for development tools updates:*

http://www.analog.com/industry/dsp/tools/fixes.html

Understanding Legacy System Files (.SYS)

In the past (i.e. the 5.x and 6.x development tools releases), life was pretty simple for the 218x when declaring the memory used in your DSP system. Only the internal memory of your processor was declared. For processors with 16k/16k of PM and DM or greater (ADSP-2187/2188/2189), no additional on-chip hardware overlay memory regions or external memory overlay regions were declared. For processors with less than 16k/16k of PM and DM respectively (ADSP-2184/2186), only the proper amount of on-chip memory and overlay memory regions were declared for your specific processor was declared in your system file.

For example, if you were using an ADSP-2181 in your system design, the following example is one method to declare your DSP system file:

```
.system                          Example_2181_System_File;
.adsp2181;
.mmap0;

.seg/pm/ram/code/data/abs=0                  int_pm[0x3fff];
.seg/dm/ram/data/abs=0                       int_dm[0x3fe0];

.endsys;
```

Example 1: Basic Legacy 2181 System File

From example 1, we can then show a progression of how the hardware overlay regions and interrupt vector table are declared:

```
.system                    Robust_Example_2181_System_File;
.adsp2181;
.mmap0;

.seg/pm/ram/code/data/abs=0                  int_tbl[0x0030];
.seg/pm/ram/code/data/abs=0x0030             int_pm[0x1fe2];
.seg/pm/ram/code/data/abs=0x2000             pm_ovl[0x2000];

.seg/dm/ram/data/abs=0                       dm_ovl[0x2000];
.seg/dm/ram/data/abs=0x2000                  int_dm[0x1fe0];

.endsys;
```

Example 2: Segmented Legacy System File (with vector table segment)

Here in example 2, we can see how the interrupt vector table, 'int_tbl', is declared separately from the rest of the internal PM region. Also, the mapping of the PM/DM hardware overlay regions (both

internal and external) are declared. Using the 5.x and 6.x tools, declaring a code or data module to a specific memory region (which was declared in your system file) was done by using the "seg=*segment_name*" qualifier. For example, a code module to place the interrupt vector table in the memory segment "*int_tbl"* declared above would look like the following:

```
.module/ram/seg = int_tbl
Example_2181_Vector_Table;
.extern main;

jump main; rti; rti; rti;                    /* reset vector */
rti; rti; rti; rti;                        /* IRQ2 interrupt */
…
```

Example 3: Legacy syntax to force a code module to a memory segment

The linker would use this "*seg=*" directive to determine where in the DSP's memory this code segment should reside.

So if you've read this far, you're probably wondering where we're going next, and when we're going to talk about using LDFs. Brace yourself for some serious stuff below…

A Simple 2181 LDF Example (ASM only)

Example 4, titled "ADSP-2181 Assembly LDF Example" in Appendix A, shows a basic LDF example for a 2181 assembly system. We'll start off with just a simple example without any hardware overlay declarations. (We'll save the scary stuff for later!)

From a first inspection of this example LDF, we see that there are a few sections to this file. For the MEMORY section of the LDF, things look somewhat similar to the example 2 architecture file. But there is also additional information in the LDF which is used by the linker as well as the simulator and emulator.

The first line of the LDF defines which specific processor is being used in your system. The line,

```
ARCHITECTURE(ADSP-2181)
```

is analogous to the ".2181" processor type declaration in the legacy architecture file.

The second line in this LDF example, is an LDF macro which identifies the object and library files generated by the assembler (from the source files) to be used in their respective segments by the linker. (FYI, all LDF macros are preceded by the dollar sign "$".)

The macro "$COMMAND_LINE_OBJECTS" contains a listing of all of the object files (*.DOJ) and library files (*.DLB) which are generated by the assembler. This listing is then used by the $OBJECTS macro, which is used by the linker to perform specific placement of each object file to its specific memory region. (The specific memory region is defined by the user in the MEMORY section of the LDF.)

The "MEMORY" section is the most familiar section of the LDF for legacy 218x users. Here we see how the different memory declarations are performed.

Here we note that the start and end addresses of the memory segment are declared in this portion of the LDF. Also in an LDF file, there is a 5 hex digit field for the address field. Veteran 21xx users are going to say to themselves, "I thought that there were only 14 bits of addressing on the 218x? What's up with this 20-bit field?" For now all we'll say is that the most-significant hex digit represents the hardware overlay region of the memory segment, also known as the 'live' address of the memory overlay. We'll get into that later when we cover a hardware overlay example for an ADSP-2189 processor.

We also see in the LDF that the data width for the memory segment must also be declared in this section. Program Memory segments must be declared as 24 bits, using the "WIDTH(24)" LDF command, and Data Memory segments must be declared as 16 bits, using the "WIDTH(16)" LDF command. *Failure to declare these memory regions with the proper data width will result in a linker error.*

The "PROCESSOR" section is where the input source files and output object files are used by the linker to assign these objects to specific segments in memory which were declared in the "MEMORY" portion of the LDF. Since we're only declaring a single processor system, we only need one "PROCESSOR" declaration in our LDF. *(Note: Since the linker is a shared resource between the 218x and 21xxx SHARC families, there is multiprocessor support inherent to the LDF. Multiple processor declarations are allowed for a SHARC system, but they are not supported for a 218x LDF.)* The name "p0" in this LDF example is arbitrary; you can use any label or name for your own LDF.

The first line in the 'PROCESSOR' section *must* contain the following LDF command:

```
"OUTPUT(COMMAND_LINE_OUTPUT_FILE)"
```

Basically, this statement tells the linker to generate an output file (*.DXE), which corresponds to the name given in the project options page in the IDE or by the name defined by the "-o" linker switch in a DOS window.

The next section, titled "SECTIONS", is where the code objects are explicitly linked to their desired memory locations. Let's look at the interrupt vector table section as an example.

```
dxe_int_tbl{
  INPUT_SECTIONS($OBJECTS(int_tbl))
} >mem_int_tbl
```

The first heading "dxe_int_tbl" is actually an intermediate label, which is used by the debugger, simulator, and emulator utilities to allow them to explicitly show the declared memory regions in their respective disassembly windows.

The "INPUT_SECTIONS" command tells the linker that the declared object files on this line will be linked into the physical memory segment "mem_int_tbl", which we've declared in the "MEMORY" section of the LDF. Here, the "$OBJECTS" macro tells the linker to link in the object file which contains the declaration:

```
.section/pm int_tbl;
```

This would be the first line of code in the source code module which describes the interrupt vector table. Here is what a 7.0 vector table declaration might look like:

```
.section/pm int_tbl;
.extern main;

jump main; rti; rti; rti;                    /* reset vector */
rti; rti; rti; rti;                        /* IRQ2 interrupt */
…
```

This excerpt would be analogous to the legacy 5.x/6.x example which we showed earlier in example 3:

```
.module/ram/seg = int_tbl
Example_2181_Vector_Table;
.extern main;

jump main; rti; rti; rti;                    /* reset vector */
rti; rti; rti; rti;                        /* IRQ2 interrupt */
…
```

You can also explicitly define which object files get linked to a specific section. For example, we may wish to have two code objects named "maincode.doj" and "function1.doj" reside in the memory segment declared as "mem_int_pm". This would be implemented as shown in example 5 below:

```
dxe_int_pm{
  INPUT_SECTIONS (maincode.doj function1.doj (int_pm))
} >mem_int_pm
```

Example 5: Forcing of multiple objects into specific memory segments

Now that we've covered the basics, let's delve further and examine a hardware overlay LDF example.

2189 ASM Hardware Overlay LDF Example

The information described in this portion of the EE note covers the 2189M assembly LDF example included in Appendix A Example 6. *(Due to the sheer size of this example, it is more convenient to include it in Appendix A and describe it in the sections below.)*

The first thing that you'll notice when looking at this LDF example is that there are a lot of memory segments declared in the "MEMORY" section of the LDF. The reason for this is simple: to define hardware overlay regions for each hardware overlay, you must declare a "live" and a "run" address.

Let's look at the Program Memory overlay declarations first. For the 2189M, internal Program Memory occupies the address range 0x0000-0x1fff, or 8k words of memory. All of the Program Memory overlay regions occupy addresses 0x2000-0x3fff.

(Note: The PMOVLAY register determines which overlay region is currently active, and determines whether the active overlay is internal or external. Please refer to the ADSP-218x Hardware Reference Manual for more information on this register.)

```
// Declare PM Overlay Segment ("Run Address")
mem_pm_ovly { TYPE(PM RAM) START(0x02000) END(0x03fff)
WIDTH(24)}

// Internal PM Overlay Segments ("Live Addresses")

// Internal 8k PM Overlay Segment 0
mem_pmpage0 { TYPE(PM RAM) START(0x02000) END(0x03fff)
WIDTH(24)}

// Internal 8k PM Overlay Segment 4
mem_pmpage4 { TYPE(PM RAM) START(0x42000) END(0x43fff)
WIDTH(24)}

// Internal 8k PM Overlay Segment 5
mem_pmpage5 { TYPE(PM RAM) START(0x52000) END(0x53fff)
WIDTH(24)}
```

Example 7: PM Overlay "*Live/Run*" memory declarations

From this LDF excerpt we can see that there are two distinct sections to the Program Memory overlay memory segment declarations. As mentioned above, here is where we declare both the "live" and the "run" memory segments for the hardware overlays. The "run" segment corresponds to the physical address of the 8k memory segment that the will be the address of the instruction which is fetched and executed by the Program Sequencer of the DSP, addresses 0x2000-0x3fff. The "live" segment declaration defines which physical hardware overlay page the overlay memory region resides in.

(Note: For you legacy 218x users, the method of defining the "live"overlay region was performed via the "-po" and "-do" ld21 linker via the DOS command line. The "run" address declarations were done using a group file and the "-group" linker command-line linker option.)

The segment declarations for the "live" addresses of the overlay regions show an important feature of the LDF. As mentioned earlier, you'll notice that the addresses are specified using 5 hex digits versus using 4 hex digits. All of you hardcore 218x users out there know that there's only 14-bits of addressing on the part, so what gives? The most significant hex digit represents the hardware overlay region (or the value of the PMOVLAY register) of where the overlay region actually physically exists on the DSP. For example the following LDF statement, assigns the memory segment named "mem_pmpage4" to Program Memory overlay four.

```
// Internal 8k PM Overlay Segment 4
mem_pmpage4 { TYPE(PM RAM) START(0x42000) END(0x43fff)
WIDTH(24)}
```

Example 8: Overlay memory region assignment

(This assignment is done by assigning the most-significant hex digit the value of four in our eample.) The remaining 4 hex digits are used to assign the 14-bit address of the overlay region, again addresses 0x2000-0x3fff, as was done before in the legacy architecture files.

For Data Memory overlay regions, basically things are done in the same manner, except for the fact that the Data Memory overlay regions are mapped to the address range 0x0000-0x1fff. *(Remember that the 32 memory-mapped control registers always reside in the upper addresses of DM, 0x3fe0-0x3fff, so this is an easy way to remember that this address range is always internal Data Memory.)*

Let's look at the Data Memory overlay memory segment declarations from our 2189 LDF example:

```
// Declare DM Overlay Segment ("Run Address") below
mem_dm_ovly{TYPE(DM RAM) START(0x00000)
END(0x01fff)WIDTH(16)}

// Internal DM Overlay Segments ("Live Addresses") below

// Internal 8k DM Overlay Segment 0
mem_dmpage0{TYPE(DM RAM) START(0x00000)
END(0x01fff)WIDTH(16)}

// Internal 8k DM Overlay Segment 4
mem_dmpage4{TYPE(DM RAM) START(0x40000)
END(0x41fff)WIDTH(16)}

// Internal 8k DM Overlay Segment 5
mem_dmpage5{TYPE(DM RAM) START(0x50000)
END(0x51fff)WIDTH(16)}

// Internal 8k DM Overlay Segment 6
mem_dmpage6{TYPE(DM RAM) START(0x60000)
END(0x61fff)WIDTH(16)}

// Internal 8k DM Overlay Segment 7
mem_dmpage7{TYPE(DM RAM) START(0x70000)
END(0x71fff)WIDTH(16)}
```

Example 9: DM Overlay "*Live/Run*" memory declarations

From example 9 above, we see again that there are both "live" and "run" memory segments declared. Here we've declared Data Memory hardware overlay regions 0, 1, 2, 4, 5, 6, and 7. *(FYI: Program Memory Overlay 3 and Data Memory Overlay 3 are not supported by the assembler.)*

Again, from this excerpt from our example LDF, we see how the Data Memory overlay regions are mapped to their respective hardware overlay region. As with the Program Memory overlays, the most significant hex digit is used to define the hardware page where the memory segment resides. For example, the following line from the LDF example assigns a memory segment named "mem_dmpage7" to the hardware overlay region 7, which occupies the address range 0x0000-0x1fff. Again, the hardware page is defined by the most significant hex digit (7 in this case), and the address range is determined by the remaining 4 hex digits, which represent the 14-bit address available on the 218x family. *(Please refer to example 10 below.)*

```
// Internal 8k DM Overlay Segment 7
mem_dmpage7 { TYPE(DM RAM) START(0x70000) END(0x71fff)
WIDTH(16) }
```

Example 10: DM Overlay Segment 7 Declaration

Now that we've covered how to declare a hardware overlay memory segment, it's time to discuss how we can now link an object module to one of these specific memory regions.

Looking at the bottom of our 2189 LDF example (line 67 to be specific), in the "SECTIONS" portion of this file, you'll see declarations for each overlay region. Figure xx below is an example of how the "run" segment for a Data Memory overlay is declared.

From this excerpt, we see the declarations for two Data Memory overlay regions, external regions 1 and 2, respectively. The PAGE_INPUT command defines what information gets mapped to this overlay segment by the linker. Simply put, all of the information included between the curly braces of the PAGE_INPUT declaration get linked in to this specific overlay region.

The next LDF command, ALL_FIT, explicitly tells the linker to include all of the declared objects within the scope of the PAGE_INPUT command to be placed into this overlay region. The INPUT_SECTIONS command tells the linker which objects are to be mapped into this overlay segment.

```
dxe_dmpage{
   PAGE_INPUT{
   ALGORITHM(ALL_FIT)
   PAGE_OUTPUT(dm_page1.ovl)
   INPUT_SECTIONS(DMOvly1.doj(dm_ovly_1))
   }>mem_dmpage1 // assign to external DM overlay region #1

   PAGE_INPUT{
   ALGORITHM(ALL_FIT)
   PAGE_OUTPUT(dm_page2.ovl)
   INPUT_SECTIONS(DMOvly2.doj(dm_ovly_2))
   }>mem_dmpage2 // assign to external DM overlay region #2

   .
   .
   .
```

Example 11: Overlay Input & Output Sections

The last line within the scope of the PAGE_INPUT section, is actually declared outside of the curly braces of this section. This is where the explicit assignment of the overlay segment to its actual "live" memory region is defined.

For example xx, we see the following statement,

```
}>mem_dmpage2
```

which assigns the overlay region to the "run" segment for the overlay (for our example, DM overlay region 2), which is declared in the MEMORY section of the LDF.

Using LDFs With C

Now that we've discussed the "gory" details of the LDF for an assembler example, let us now discuss what is needed for C runtime support. First, we'll discuss how things worked in the past.

With the legacy 5.x and 6.x development tools, the architecture file definitions were fairly simple. The user was only responsible for declaring the actual memory architecture of the specific processor that they were using in their system. The heap and stack segments were linked in automatically by the linker *(ld21.exe)*, and were not declared in the architecture file. *(If the legacy 218x user wished to modify the length of the heap and stack memory segments, they were required to edit and reassemble the files heap.dsp and stack.dsp to reserve the desired amount of Data Memory for these two memory segments.)*

With our LDF, the first thing that you'll notice is that there are explicit memory segment declarations for both the heap and the stack, which again reside in data memory. Changing the length of these two segments is done by simply changing the address range for each segment. These two segments must be declared when using C.

For legacy users, the search path for the compiler to locate library files was defined in your autoexec.bat file with the "ADI_LIB" or "ADI_DSP" environment variables. With our LDF, the following line defines the search path for the compiler:

```
SEARCH_DIR($ADI_DSP\218x\lib)
```

Here, the LDF macro "$ADI_DSP" is assigned the value of the directory path where the VisualDSP development tools are installed. *(By default, the tools installation path is "C:\Program Files\Analog Devices\VisualDSP").*

Another thing to notice is that the "$OBJECTS" LDF macro has a lot more assigned to it. This macro tells the linker that all of the objects listed on this line are to be linked into the executable file (*.DXE). Here is a brief description of what each item on this line represents:

2189_int_tab.doj – This file is the interrupt vector table which supports the C runtime environment. Basically, it has C-style labels for each of the interrupts in the interrupt vector table. This file is analogous to the legacy file "2181_hdr.obj".

218x_hdr.doj – This file contains the code for C runtime environment initialization. For example, this file is responsible for setting up the C runtime stack and heap memory segments, assigning appropriate values to the registers used by the compiler, etc.

$COMMAND_LINE_OBJECTS – This LDF macro is used to assign all of the user input object files (i.e. all of your source modules) to the $OBJECTS LDF macro in order for the linker to link in everything into the executable.

Libio.dlb – This library file supports calls to the stdio.h library.

Libc.dlb – This library file includes all of the C runtime environment initialization library routines.

218x_exit.doj – This library file contains code which is used upon exiting from the scope of main.

All of the files listed above must be included in your LDF when using C.

Advanced Tips and Tricks For Legacy Users

Linking A Single Legacy Source File Declared With The "ABS=0x000" Directive

The example LDF files included with the VisualDSP development tools will directly support C source files or mixed C and assembly source files only. To support strict assembly source files, the default LDF file for your processor must be modified.

The default VisualDSP LDF files separate the interrupt vector table into a distinct memory segment which occupies the address range PM 0x0000 – 0x002f. *(This specific segment contains memory declarations for each of the hardware interrupt vectors in the vector table. Please refer to example 12 below.)*

```
MEMORY{
   seg_inttab {TYPE(PM RAM) START(0x00000) END(0x0002f)
   WIDTH(24)}
   seg_code {TYPE(PM RAM) START(0x00030) END(0x037ff)
   WIDTH(24)}
   seg_data2 {TYPE(PM RAM) START(0x03800) END(0x03fff)
   WIDTH(24)}

   seg_data1{TYPE(DM RAM) START(0x00000) END(0x02fff)
   WIDTH(16)}
   seg_heap {TYPE(DM RAM) START(0x03000) END(0x037ff)
   WIDTH(16)}
   seg_stack {TYPE(DM RAM) START(0x03800) END(0x03fdf)
   WIDTH(16)}
}// end MEMORY declaration

PROCESSOR p0{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS{
      sec_inttab{
         INPUT_SECTIONS( $OBJECTS( IVreset ))              // 0x0000
         INPUT_SECTIONS( $OBJECTS( IVirq2 ))               // 0x0004
         INPUT_SECTIONS( $OBJECTS( IVirql1 ))              // 0x0008
         INPUT_SECTIONS( $OBJECTS( IVirql0 ))              // 0x000c
         INPUT_SECTIONS( $OBJECTS( IVsport0xmit ))    // 0x0010
         INPUT_SECTIONS( $OBJECTS( IVsport0recv ))    // 0x0014
         INPUT_SECTIONS( $OBJECTS( IVirqe ))               // 0x0018
         INPUT_SECTIONS( $OBJECTS( IVbdma ))               // 0x001c
         INPUT_SECTIONS( $OBJECTS( IVirq1 ))               // 0x0020
         INPUT_SECTIONS( $OBJECTS( IVirq0 ))               // 0x0024
         INPUT_SECTIONS( $OBJECTS( IVtimer ))              // 0x0028
         INPUT_SECTIONS( $OBJECTS( IVpwrdwn ))        // 0x002c
      } >seg_inttab
   …
```

Example 12: LDF interrupt vector table declaration for C

Often, legacy users have a .SYS file which was setup with one big PM code segment written in assembly (versus separating the code segment from the interrupt vector table segment, like the manner done in the default LDF files, or in our case as shown in example 12.)

If this scenario applies to you, you can modify the default LDF file or create your own to look like the following:

```
MEMORY{
   seg_code {TYPE(PM RAM) START(0x00000) END(0x03fff)
   WIDTH(24)}
   seg_data2 {TYPE(PM RAM) START(0x03800) END(0x03fff)
   WIDTH(24)}

   seg_data1{TYPE(DM RAM) START(0x00000) END(0x02fff)
   WIDTH(16)}
   seg_heap {TYPE(DM RAM) START(0x03000) END(0x037ff)
   WIDTH(16)}
   seg_stack {TYPE(DM RAM) START(0x03800) END(0x03fdf)
   WIDTH(16)}
}// end MEMORY declaration

PROCESSOR p0{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS{
      sec_code{
         INPUT_SECTIONS( $OBJECTS(Program))
      }>seg_code

      sec_data2{
         INPUT_SECTIONS( $OBJECTS(Program))
      }>seg_ data2

      sec_ data1{
         INPUT_SECTIONS( $OBJECTS(Program))
      }>seg_ data1

      sec_ heap {
         INPUT_SECTIONS( $OBJECTS(Program))
      }>seg_heap

      sec_stack{
         INPUT_SECTIONS( $OBJECTS(Program))
      }>seg_ stack
      …
```

Example 13: Single legacy source file LDF example

Linking A Legacy Source File Using The ".var/abs=addr" Assembler Directive

The method used by the VisualDSP 7.0 tools to implement absolute placement for variables is performed differently than with the legacy assembler and linker. If your legacy assembler source file has a ".var/abs=addr" declaration, the 7.0 assembler and linker need to generate "RESOLVE" commands for your LDF; these "RESOLVE" statements provide the same functionality as the "abs=addr" legacy qualifier.

Below are examples showing legacy syntax versus the RESOLVE LDF command for 7.0:

```
.var/dm/abs=0x1234 my_dm_variable;                    /* legacy dm
variable */
.var/dm/abs=0x2000 my_dm_buffer[100]; /* legacy dm buffer */

.var/pm/abs=0x2000 my_pm_buffer[0x20]; /* legacy pm buffer */




PROCESSOR p0{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
      RESOLVE(my_dm_variable, 0x1234)
      RESOLVE(my_dm_buffer, 0x2000)

      RESOLVE(my_pm_buffer, 0x2000)

      SECTIONS{
```

Please note that the placement of the "RESOLVE" command *must* be within the scope of the "PROCESSOR" LDF directive *before* the scope of the "SECTIONS" command.

Absolute placement can be done manually by the programmer, but this may be a monumental task if there are numerous variables declared as absolutes in the legacy source file(s). Thankfully, the IDE can provide absolute placement of all of your declared variables for you.

This automated procedure is performed by the IDE by the inclusion of the "-Ao" assembler command line option, which can be invoked in a DOS window or by including the *"-Ao filename"* option in the "additional options" text window of the "assemble" options tab of the IDE. Below is an example showing the invocation of the "-Ao" switch on a command line:

```
Easm218x test.dsp –o test.doj –legacy –Ao testlink.ldf
```

By invoking the "-Ao" assembler option in the above example, a text file arbitrarily named "testlink.ldf" is generated by the assembler that contains all of the "RESOLVE" statements for the absolutely declared variables from the source file "test.dsp". This text file then needs to be included in the LDF file to allow the linker to implement the absolute placement of those variables. Below is an LDF example where the inclusion of our generated file, "testlink.ldf", is performed by using the LDF command "INCLUDE":

```
PROCESSOR p0{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
   INCLUDE(testlink.ldf)

   SECTIONS{
   …
```

As with the "RESOLVE" LDF command, the "INCLUDE" command must lie within the scope of the "PROCESSOR" and "SECTIONS" LDF statements.

Use of the "-Ao" command line switch is optional. If this switch is omitted on the command line, then the default name for the LDF with the RESOLVE statements for your absolutely declared objects will be named the same as the input source file with "resolve_" attached to the front of the filename and ".ldf" used for the filename extension.

For example, if the input source filename was "myfile.dsp", the generated LDF filename generated by default would be "resolve_myfile.ldf". Also, the "-legacy" assembler option must be used in conjunction with the "-Ao" assembler option.

Another important point to mention here is that objects linked at absolute address locations using the linker's RESOLVE command *must* be declared as 'globals' because this RESOLVE command is only supported by the linker on global data.

This requirement of having objects declared at absolute addresses applies for new syntax and code modules as well as for legacy code modules. For example, failure to declare the following variable "myData" as a global,

```
.section/dm data1;
.var myData;
.global myData;      // global must be declared when using RESOLVE LDF
command
```

would result in a linker error which would say the following:

```
[Error E1069] The symbol used in a RESOLVE linker command: the symbol
'myData' was not found in the linker symbol table
```

Absolute Module Placement Using The ".module/abs=addr"Assembler Directive

Another point to mention is when a legacy assembly source module is declared using the ".module/abs = addr" qualifier. Since the 7.0 assembler ignores this legacy qualifier, it is up to the programmer to ensure that the source module gets linked into the proper memory segment and at the proper segment address.

The most straightforward method of implementing this is by generating a memory segment in your LDF which will be used to contain this specific module. For example, if your legacy assembly module "cross.dsp" was declared as below:

```
/* Filename: Cross.dsp */
.module/abs=0x3000 My_Example_Function;
.entry Cross_Product:

Cross_Product:
      nop;
      rts;
```

You would want to create a memory segment in your LDF that would look somewhat like the following:

```
MEMORY{
  …
  mem_cross_prod{TYPE(PM RAM) START(0x03000)
  END(0x0302f) WIDTH(24)}   // memory segment for cross
  product function
  …
} // end 'MEMORY' declaration

PROCESSOR p0{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS{
      …
      dxe_cross_prod{
       INPUT_SECTIONS(cross.doj (program))
     }>mem_vect_tbl
            …
```

Using Conditional Directives In Your LDF

Since the default LDF files that come with the VisualDSP development tools support C code, these files must be modified to work with assembly source files only. This section is where we'll discuss the use of the "#ifdef", "#else", and "#endif" conditional preprocessor directives that allow for the control of which sections of your LDF actually get utilized by the linker when building your project.

For the example that we'll discuss here, please refer to Example #14, which is included in Appendix A at the end of this EE Note. For this example LDF, we're declaring the memory segments for an ADSP-2181 system. Since the linker handles the interrupt vector table (and its specific interrupt signal names), and since the linker also includes the multiple run-time library and objects for C runtime support when using C source modules, we'll make this the default functionality of our LDF.

To force the linker to use the assembly declarations of our LDF, we'll need to invoke the linker command line option "-MD*macro*", where the label *"macro"* is the name of the conditional directive that you choose. For our example we'll be using the macro definition "ASM_TEST". The invocation of this command line option can be done by invoking the following command line example:

```
linker myfile.doj –T 2181.ldf –DADSP-2181 –o myfile.dxe –
```

This macro can also be defined via the IDE by entering the string "-MDASM_TEST" in the "Additional options" text box of the "Link" options tab of the "Project Options" property page.

Below is a brief excerpt from our conditional 2181 LDF example which shows the conditional declarations for C versus assembly support on the "$OBJECTS" command line:

```
ARCHITECTURE(ADSP-2181)

#ifdef ASM_TEST
$OBJECTS = $COMMAND_LINE_OBJECTS;
#else
SEARCH_DIR( $ADI_DSP\218x\lib )
$OBJECTS = 218x_int_tab.doj, 218x_hdr.doj,
$COMMAND_LINE_OBJECTS, libio.dlb, libc.dlb, 218x_exit.doj;
#endif
…
```

From this example we see that when the macro "ASM_TEST" is not defined, using the linker's "-MD*macro*" directive, that the C runtime libraries

and objects get linked into the executable via the "$OBJECTS" LDF macro.

Advanced LDF Tips and Tricks For C Users

There is support in the LDF as well as C compiler intrinsics that allow for placing C source modules, functions, and variables at specified memory segments. One method of implementing this is by explicitly declaring the object file created by the compiler (and the assembler) into a specified memory segment.

For example, the following C array declaration in the source file "array1.c"

```
section ("dm_ovly_0") int my_array [5] = {10, 20, 30 ,40, 50};
```

will be placed into the memory segment declaration "mem_dm_ovly0" which uses the LDF command "INPUT_SECTIONS", like the one shown below:

```
MEMORY{
      …
   mem_dm_ovly_0{TYPE(DM RAM) START(0x02000) END(0x03fff)
WIDTH(16)}
      …
}

SECTIONS{
      …
   dxe_dm_ovly_0{
      INPUT_SECTIONS($OBJECTS(dm_ovly_0))
   }>mem_ dm_ovly_0
      …
```

This array could also be declared without the C "sections" intrinsic. An alternate method would be to declare the C array in the file "array1.c" as a "normal" array declaration like the following:

```
Int my_array [5] = {10, 20, 30, 40, 50};
```

Then the following command would need to be entered in your LDF to force the linker to link the file object "array1.doj" into the memory segment "mem_dm_ovly0":

```
MEMORY{
      …
   mem_dm_ovly_0{TYPE(DM RAM) START(0x02000)
END(0x03fff) WIDTH(16)}
      …
}

SECTIONS{
      …
   dxe_dm_ovly_0{
      INPUT_SECTIONS(array1.doj(dm_ovly_0))
   }>mem_ dm_ovly_0
      …
```

Conclusion

Hopefully from all of these LDF examples you now have a better understanding of what an LDF file does and how the many functions of the LDF are implemented. We saw how a simple LDF can be written and then migrated on to more complex topics such as hardware overlay linking, forcing variables and code modules into specific memory segments using both C and assembler source.

For more information on all of these topics, please refer to the following 21xx VisualDSP development tools manuals:

- Assembler Manual for ADSP-218x Family DSPs
- Assembler Manual for ADSP-219x Family DSPs
- C Compiler & Library Manual for ADSP-218x Family DSPs
- C Compiler & Library Manual for ADSP-219x Family DSPs
- Linker & Utilities Manual for ADSP-21xx Family DSPs
- VisualDSP User's Guide for ADSP-21xx Family DSPs

For additional and related information, please refer to our website's 16-bit family development tools application notes page, which is located at the following URL:

http://www.analog.com/techsupt/application_notes/application_notes.html#1a

Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Appendix A LDF Examples

Example 3: ADSP-2181 Assembly LDF Example

```
ARCHITECTURE(ADSP-2181)
$OBJECTS = $COMMAND_LINE_OBJECTS;

// 2181 has 16K words of 24-bit Program RAM and 16K words of 16-bit Data RAM
MEMORY{
  mem_vect_tbl {TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24)}   // 48 locations for vector table (12*4)
  mem_pmcode {TYPE(PM RAM) START(0x00030) END(0x03fff) WIDTH(24)}  // 16k segment for internal PM segment (minus 48 locations)

  mem_dmdata {TYPE(DM RAM) START(0x00000) END(0x03fdf) WIDTH(16)}  // 16k segment for internal DM (minus 32 mem mapped cntrl regs)
} // end 'MEMORY' declaration

PROCESSOR p0{ // single processor declaration for processor 'p0'
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE ) // define output files for linker

  SECTIONS{ // declare input and output object file sections
    dxe_vect_tbl{
      INPUT_SECTIONS($OBJECTS(int_tbl)) // place module declared as '.section/pm int_tbl;'
    }>mem_vect_tbl                          // into this memory segment

    dxe_pmcode{
      INPUT_SECTIONS($OBJECTS(program)) // place module declared as '.section/pm program;'
    }>mem_pmcode                            // into this memory segment

    dxe_dmdata{
      INPUT_SECTIONS($OBJECTS(data))   // place module declared as '.section/pm data;'
    }>mem_dmdata                            // into this memory segment

  }// end 'SECTIONS' declaration
}// end 'PROCESSOR p0' declaration
```

Example 5: ADSP-2189 Assembly LDF Example

```
ARCHITECTURE(ADSP-2189)                    // Define processor type for an ADSP-2189 memory architecture
$OBJECTS = $COMMAND_LINE_OBJECTS;

// The ADSP-2189M has 32K words (24-bit) of Program RAM and 48K words (16-bit) of Data RAM
MEMORY{
   // Internal PM Memory Segments (Non-overlay segments)
   mem_vect_tbl    { TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24) }   // 48 locations for vector table (12 * 4)
   mem_pmcode    { TYPE(PM RAM) START(0x00030) END(0x01fff) WIDTH(24) }   // 8k segment for internal PM

   /////////////// Declare PM Overlay Segment ("Live Address") below
   mem_pm_ovly   { TYPE(PM RAM) START(0x02000) END(0x03fff) WIDTH(24) }   // PM Overlay Segment "Live Address" Declaration

   // Internal PM Overlay Segments ("Run Addresses") below
   mem_pmpage0   { TYPE(PM RAM) START(0x02000) END(0x03fff) WIDTH(24) }   // Internal 8k PM Overlay Segment 0
   mem_pmpage4   { TYPE(PM RAM) START(0x42000) END(0x43fff) WIDTH(24) }   // Internal 8k PM Overlay Segment 4
   mem_pmpage5   { TYPE(PM RAM) START(0x52000) END(0x53fff) WIDTH(24) }   // Internal 8k PM Overlay Segment 5

   /////////////// Declare DM Overlay Segment ("Live Address") below
   mem_dm_ovly   { TYPE(DM RAM) START(0x00000) END(0x01fff) WIDTH(16) }   // DM Overlay Segment "Live Address" Declaration

   // Internal DM Overlay Segments ("Run Addresses") below
   mem_dmpage0   { TYPE(DM RAM) START(0x00000) END(0x01fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 0
   mem_dmpage4   { TYPE(DM RAM) START(0x40000) END(0x41fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 4
   mem_dmpage5   { TYPE(DM RAM) START(0x50000) END(0x51fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 5
   mem_dmpage6   { TYPE(DM RAM) START(0x60000) END(0x61fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 6
   mem_dmpage7   { TYPE(DM RAM) START(0x70000) END(0x71fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 7

   // Internal DM Memory Segment (Non-overlay segment)
   mem_int_dm {TYPE(DM RAM) START(0x02000) END(0x03fdf) WIDTH(16)} // Internal 8k segment minus 32 locations for mem mapped cntrl regs

   // External PM Overlay Segments  ("Live Addresses") below
   mem_pmpage1   { TYPE(PM RAM) START(0x12000) END(0x13fff) WIDTH(24) }   // External 8k PM Overlay Segment 1
   mem_pmpage2   { TYPE(PM RAM) START(0x22000) END(0x23fff) WIDTH(24) }   // External 8k PM Overlay Segment 2

   // External DM Overlay Segments  ("Live Addresses") below
   mem_dmpage1   { TYPE(DM RAM) START(0x10000) END(0x11fff) WIDTH(16) }   // External 8k DM Overlay Segment 1
   mem_dmpage2   { TYPE(DM RAM) START(0x20000) END(0x21fff) WIDTH(16) }   // External 8k DM Overlay Segment 2
}// End "MEMORY" declaration section

PROCESSOR p0{ // single processor declaration for processor "p0"

   OUTPUT($COMMAND_LINE_OUTPUT_FILE) // define output files for linker

   SECTIONS{
    dxe_vect_tbl{
       INPUT_SECTIONS($OBJECTS(interrupts))
       }>mem_vect_tbl // places the file object in internal  non-overlay PM segment "mem_vect_tbl" (PM0x0000-0x002f)

     dxe_code{
       INPUT_SECTIONS($OBJECTS(program))
       }>mem_pmcode // places the file object in internal non-overlay PM segment "mem_pmcode" (PM0x0030-0x1fff)

     dxe_int_dm{
       INPUT_SECTIONS(DMOvly0.doj(int_dmda))
       }>mem_int_dm // places the file object "DMOvly.doj" in internal non-overlay DM segment "mem_int_dm" (DM0x2000-0x3fdf)
```

(Example 5: continued)

```
    // DM Overay Declarations (assign data modules to their "run address" memory regions)
     dxe_dmpage{
        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(dm_page1.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(DMOvly1.doj(dm_ovly_1))       // input object file for this overlay region
        }>mem_dmpage1                                        // assign to external DM overlay region #1

        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(dm_page2.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(DMOvly2.doj(dm_ovly_2))       // input object file for this overlay region
        }>mem_dmpage2                                        // assign to external DM overlay region #2

        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(dm_page4.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(DMOvly4.doj(dm_ovly_4))       // input object file for this overlay region
        }>mem_dmpage4                                        // assign to internal DM overlay region #4

        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(dm_page5.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(DMOvly5.doj(dm_ovly_5))       // input object file for this overlay region
        }>mem_dmpage5                                        // assign to internal DM overlay region #5
    }>mem_dm_ovly                                            // assign overlay objects to DM overlay "run address" DM0x0000-0x1fff

       // PM Overlay Declarations (assign data modules to their "run address" memory regions)
      dxe_pmpage{
        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(pm_page0.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(PMOvly0.doj(pm_ovly_0))       // input object file for this overlay region
        }>mem_pmpage0                                        // assign to internal PM overlay region #0

        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(pm_page4.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(PMOvly4.doj(pm_ovly_4))       // input object file for this overlay region
        }>mem_pmpage4                                        // assign to internal PM overlay region #4

        PAGE_INPUT{
                ALGORITHM(ALL_FIT)
                PAGE_OUTPUT(pm_page5.ovl)                    // assembler overlay output file for linker
                INPUT_SECTIONS(PMOvly5.doj(pm_ovly_5))       // input object file for this overlay region
        }>mem_pmpage5                                        // assign to internal PM overlay region #5
      }>mem_pm_ovly                                          // assign overlay objects to PM overlay "run address" PM0x2000-0x3fff
    }// end "SECTIONS" declaration section of LDF
  }// "PROCESSOR" declaration section of LDF
```

Example 6: ADSP-2189 C LDF Example

```
ARCHITECTURE(ADSP-2189)                    // Define processor type for an ADSP-2189 memory architecture
SEARCH_DIR($ADI_DSP\218x\lib)              // Default search directory for libraries
$OBJECTS = 2189_int_tab.doj, 218x_hdr.doj, $COMMAND_LINE_OBJECTS, libio.dlb, libc.dlb, 218x_exit.doj;

MEMORY{
   // Internal PM Memory Segments
   mem_vect_tbl  { TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24) }   // 48 locations for vector table (12 * 4)
   mem_pmcode    { TYPE(PM RAM) START(0x00030) END(0x01fff) WIDTH(24) }   // 8k segment for internal PM

   /////////////// Declare PM Overlay Segment here ("run" address)
   mem_pm_ovly   { TYPE(PM RAM) START(0x02000) END(0x03fff) WIDTH(24) }   // PM Overlay Segment Declaration for PLIT only

   // Internal PM Overlay Segments ("live" addresses)
   mem_pmpage0   { TYPE(PM RAM) START(0x02000) END(0x03fff) WIDTH(24) }   // Internal 8k PM Overlay Segment 0
   mem_pmpage4   { TYPE(PM RAM) START(0x42000) END(0x43fff) WIDTH(24) }   // Internal 8k PM Overlay Segment 4
   mem_pmpage5   { TYPE(PM RAM) START(0x52000) END(0x53fff) WIDTH(24) }   // Internal 8k PM Overlay Segment 5

   /////////////// Declare DM Overlay Segment here ("run" address)
   mem_dm_ovly   { TYPE(DM RAM) START(0x00000) END(0x01fff) WIDTH(16) }   // DM Overlay Segment Declaration for PLIT only

   // Internal DM Overlay Segments ("live" addresses)
   mem_dmpage0   { TYPE(DM RAM) START(0x00000) END(0x01fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 0
   mem_dmpage4   { TYPE(DM RAM) START(0x40000) END(0x41fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 4
   mem_dmpage5   { TYPE(DM RAM) START(0x50000) END(0x51fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 5
   mem_dmpage6   { TYPE(DM RAM) START(0x60000) END(0x61fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 6
   mem_dmpage7   { TYPE(DM RAM) START(0x70000) END(0x71fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 7

   // Internal DM Memory Segment
   mem_int_dm    { TYPE(DM RAM) START(0x02000) END(0x02fff) WIDTH(16) }   // internal (non-overlay) DM segment
   seg_heap   { TYPE(DM RAM) START(0x03000) END(0x037ff) WIDTH(16) }      // default heap segment declaration (in non-overlay DM)
   seg_stack  { TYPE(DM RAM) START(0x03800) END(0x03fdf) WIDTH(16) }      // default stack memory declaration (in non-overlay DM)

   // External PM Overlay Segments ("live" addresses)
   mem_pmpage1   { TYPE(PM RAM) START(0x12000) END(0x13fff) WIDTH(24) }   // External 8k PM Overlay Segment 1
   mem_pmpage2   { TYPE(PM RAM) START(0x22000) END(0x23fff) WIDTH(24) }   // External 8k PM Overlay Segment 2

   // External DM Overlay Segments ("live" addresses)
   mem_dmpage1   { TYPE(DM RAM) START(0x10000) END(0x11fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 1
   mem_dmpage2   { TYPE(DM RAM) START(0x20000) END(0x21fff) WIDTH(16) }   // Internal 8k DM Overlay Segment 2

}


PROCESSOR p0{

   OUTPUT($COMMAND_LINE_OUTPUT_FILE)

   SECTIONS{
    dxe_vect_tbl{ // C Runtime Interrupt Vector Declarations
        INPUT_SECTIONS( $OBJECTS( IVreset ))          // 0x0000 Reset vector
        INPUT_SECTIONS( $OBJECTS( IVirq2 ))               // 0x0004 IRQ2 interrupt
        INPUT_SECTIONS( $OBJECTS( IVirql1 ))          // 0x0008 IRQL1 interrupt
        INPUT_SECTIONS( $OBJECTS( IVirql0 ))          // 0x000c IRQL0 interrupt
        INPUT_SECTIONS( $OBJECTS( IVsport0xmit ))     // 0x0010 SPORT0 Tx interrupt
        INPUT_SECTIONS( $OBJECTS( IVsport0recv ))     // 0x0014 SPORT0 Rx interrupt
        INPUT_SECTIONS( $OBJECTS( IVirqe ))           // 0x0018 IRQE interrupt
        INPUT_SECTIONS( $OBJECTS( IVbdma ))           // 0x001c BDMA interrupt
        INPUT_SECTIONS( $OBJECTS( IVirq1 ))               // 0x0020 IRQ1 interrupt
        INPUT_SECTIONS( $OBJECTS( IVirq0 ))               // 0x0024 IRQ0 interrupt
        INPUT_SECTIONS( $OBJECTS( IVtimer89 ))        // 0x0028 Timer interrupt (internal)
        INPUT_SECTIONS( $OBJECTS( IVpwrdwn ))         // 0x002c Powerdown interrupt
       }>mem_vect_tbl
```

(Example 6: continued)

```
dxe_code{
    INPUT_SECTIONS($OBJECTS(program))
    }>mem_pmcode

dxe_int_dm{
            INPUT_SECTIONS($OBJECTS(data1))
    }>mem_int_dm

  // support for initialization
  sec_ctor
  {
    INPUT_SECTIONS( $OBJECTS(ctor) )
  } >mem_int_dm

  // 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

// DM Overlay "Run" Segment Declarations
dxe_dmpage{                                          // arbitrary label for linker for overlay segments
    PAGE_INPUT{                                      // define what material goes into the page image
            ALGORITHM(ALL_FIT)                       // "fit" all of the material into this page
            PAGE_OUTPUT(DM_PAGE0.OVL)                // output file name of overlay segment for linker
            INPUT_SECTIONS(DMOVLY0.DOJ(dm_ovly_0))   // specify what objects are inputs to this specific overlay region
    }>mem_dmpage0                                    // end of declaration for DM page 0

        PAGE_INPUT{                                  // define what material goes into the page image
            ALGORITHM(ALL_FIT)                       // "fit" all of the material into this page
            PAGE_OUTPUT(DM_PAGE1.OVL)                // output file name of overlay segment for linker
            INPUT_SECTIONS(DMOVLY1.DOJ(dm_ovly_1))   // specify what objects are inputs to this specific overlay region
    }>mem_dmpage1                                    // end of declaration for DM page 1

    PAGE_INPUT{                                      // define what material goes into the page image
            ALGORITHM(ALL_FIT)                       // "fit" all of the material into this page
            PAGE_OUTPUT(DM_PAGE2.OVL)                // output file name of overlay segment for linker
            INPUT_SECTIONS(DMOVLY2.DOJ(dm_ovly_2))   // specify what objects are inputs to this specific overlay region
    }>mem_dmpage2                                    // end of declaration for DM page 2

    PAGE_INPUT{                                      // define what material goes into the page image
            ALGORITHM(ALL_FIT)                       // "fit" all of the material into this page
            PAGE_OUTPUT(DM_PAGE4.OVL)                // output file name of overlay segment for linker
            INPUT_SECTIONS(DMOVLY4.DOJ(dm_ovly_4))   // specify what objects are inputs to this specific overlay region
    }>mem_dmpage4                                    // end of declaration for DM page 4

    PAGE_INPUT{                                      // define what material goes into the page image
            ALGORITHM(ALL_FIT)                       // "fit" all of the material into this page
            PAGE_OUTPUT(DM_PAGE5.OVL)                // output file name of overlay segment for linker
            INPUT_SECTIONS(DMOVLY5.DOJ(dm_ovly_5))   // specify what objects are inputs to this specific overlay region
    }>mem_dmpage5                                    // end of declaration for DM page 5
}>mem_dm_ovly                                        // assign name of "live" address for overlay region here (DM0x0000-0x1fff)
```

(Example 6: continued)

```
      // PM Overlay "Run" Segment Declarations
       dxe_pmpage{
          PAGE_INPUT{                                    // define what material goes into the page image
                    ALGORITHM(ALL_FIT)                   // "fit" all of the material into this page
                    PAGE_OUTPUT(PM_PAGE0.OVL)            // output file name of overlay segment for linker
                    INPUT_SECTIONS(PMOVLY0.DOJ(pm_ovly_0))  // specify what objects are inputs to this specific overlay region
          }>mem_pmpage0                                  // end of declaration for PM page 0

          PAGE_INPUT{                                    // define what material goes into the page image
                    ALGORITHM(ALL_FIT)                   // "fit" all of the material into this page
                    PAGE_OUTPUT(PM_PAGE4.OVL)            // output file name of overlay segment for linker
                    INPUT_SECTIONS(PMOVLY4.DOJ(pm_ovly_4))  // specify what objects are inputs to this specific overlay region
          }>mem_pmpage4                                  // end of declaration for PM page 0

          PAGE_INPUT{                                    // define what material goes into the page image
                    ALGORITHM(ALL_FIT)                   // "fit" all of the material into this page
                    PAGE_OUTPUT(PM_PAGE5.OVL)            // output file name of overlay segment for linker
                    INPUT_SECTIONS(PMOVLY5.DOJ(pm_ovly_5))  // specify what objects are inputs to this specific overlay region
          }>mem_pmpage5                                  // end of declaration for PM page 0
       }>mem_pm_ovly                                     // assign name of "live" address for overlay region here (PM0x2000-0x3fff)
    } // end SECTIONS
} // end PROCESSOR p0
```

Example 14: ADSP-2181 Conditional LDF Example

```
ARCHITECTURE(ADSP-2181)

#ifdef ASM_TEST          // if using ASM define this LDF macro (use "-MDASM_TEST" linker option)
$OBJECTS = $COMMAND_LINE_OBJECTS;
#else                                    // otherwise use C runtime support (default setting)
SEARCH_DIR( $ADI_DSP\218x\lib )
$OBJECTS = 218x_int_tab.doj, 218x_hdr.doj, $COMMAND_LINE_OBJECTS, libio.dlb, libc.dlb, 218x_exit.doj;
#endif                                   // end macro definition

#ifdef ASM_TEST          // if using ASM use the following memory segment and LDF section declarations
MEMORY{
  mem_vect_tbl {TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24)}
  mem_pmcode {TYPE(PM RAM) START(0x00030) END(0x03fff) WIDTH(24)}

  mem_dmdata {TYPE(DM RAM) START(0x00000) END(0x03fdf) WIDTH(16)}
}

PROCESSOR p0{
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

  SECTIONS{
    dxe_vect_tbl{
      INPUT_SECTIONS($OBJECTS(interrupts))
    }>mem_vect_tbl

    dxe_pmcode{
      INPUT_SECTIONS($OBJECTS(program))
    }>mem_pmcode

    dxe_dmdata{
      INPUT_SECTIONS($OBJECTS(data))
    }>mem_dmdata

  }// end 'SECTIONS' declaration
}// end 'PROCESSOR p0' declaration

#else     // if using C use the following memory segment and LDF section declarations
MEMORY
{
   seg_inttab { TYPE(PM RAM) START(0x00000) END(0x0002f) WIDTH(24) }
   seg_code   { TYPE(PM RAM) START(0x00030) END(0x037ff) WIDTH(24) }
   seg_data2  { TYPE(PM RAM) START(0x03800) END(0x03fff) WIDTH(24) }

   seg_data1  { TYPE(DM RAM) START(0x00000) END(0x02fff) WIDTH(16) }
   seg_heap   { TYPE(DM RAM) START(0x03000) END(0x037ff) WIDTH(16) }
   seg_stack  { TYPE(DM RAM) START(0x03800) END(0x03fdf) WIDTH(16) }
}

PROCESSOR p0
{
   OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

   SECTIONS
   {
      sec_inttab
      {
         INPUT_SECTIONS( $OBJECTS( IVreset ))            // 0x0000
         INPUT_SECTIONS( $OBJECTS( IVirq2 ))                  // 0x0004
         INPUT_SECTIONS( $OBJECTS( IVirql1 ))            // 0x0008
         INPUT_SECTIONS( $OBJECTS( IVirql0 ))            // 0x000c
         INPUT_SECTIONS( $OBJECTS( IVsport0xmit ))            // 0x0010
         INPUT_SECTIONS( $OBJECTS( IVsport0recv ))       // 0x0014
         INPUT_SECTIONS( $OBJECTS( IVirqe ))                  // 0x0018
```

(Example 14: continued)

```
        INPUT_SECTIONS( $OBJECTS( IVbdma ))                    // 0x001c
        INPUT_SECTIONS( $OBJECTS( IVirq1 ))                    // 0x0020
        INPUT_SECTIONS( $OBJECTS( IVirq0 ))                    // 0x0024
        INPUT_SECTIONS( $OBJECTS( IVtimer ))                // 0x0028
        INPUT_SECTIONS( $OBJECTS( IVpwrdwn ))               // 0x002c
    } >seg_inttab

    sec_code
    {
        INPUT_SECTIONS( $OBJECTS(program) )
    } >seg_code

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

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

    sec_ctor
    {
        INPUT_SECTIONS( $OBJECTS(ctor) )
    } >seg_data1

    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
  }
}
#endif
```