

C LINKER LEGACY SUPPORT

Overview

The linker and utilities provide several types of support for code developed with previous development software tools. For more information on legacy support, see [“Utilities” on page B-1](#).

This section describes the differences between current and previous linker releases targeting the ADSP-218x DSPs, and code migration to the ADSP-219x DSPs. It also provides migration information to help you make the necessary code changes to use the newer tools for ongoing ADSP-218x family development.

For more information on updating assembler code, see the *VisualDSP++ 2.0 Assembler and Preprocessor Manuals for ADSP-21xx DSPs*. For more information on updating your C/C++ code, see the *VisualDSP++ 2.0 Compiler and Library Manual* for your target DSP. Those manuals are packaged with your software distribution.

The VisualDSP++ 2.0 linker has new commands and options that provide broader and more finely grained support for various DSP architectures. Specifically, it adds the following features, not available through the ADSP-21xx Release 6.1 linker:

- Definition of shared memory systems (ADSP-2191 only)
- Creation of multiple ELF-formatted executable files

Converting from .SYS to .LDF

To use the new linker, you must define the target system in a Linker Description File (LDF). You must include a definition of each object or object library in this LDF prior to linking them in an executable.

The .LDF replaces the system specification file (.SYS) used in Releases 6.1 and earlier.

Converting from .SYS to .LDF

This section describes in general terms how to convert from .SYS to .LDF syntax. The process is not complicated, but the syntax is noticeably different.

The commands in the LDF define the target system and order the processing of linker output for it. Within this file, you can specify libraries, search paths, and input files for defining the system and resolving variables and labels. You can write procedures for placing sections within the LDF, as shown in the sequence of vectors in the interrupt table (`sec_inttab`), with a location counter increment following each increment. You can also specify multiple output files (not shown in this example).

The following sections provide examples of the LDFs for the ADSP-218x, both for C and assembly code:

- [“C code SEGMENT\(“...” — LDF SECTIONS} Conversion” on page C-3](#)
- [“Assembler .SEGMENT — LDF SECTIONS} Conversion” on page C-11](#)
- [“SYS .SEGMENT — LDF MEMORY} Conversion” on page C-17](#)

C code SEGMENT("...") — LDF SECTIONS{} Conversion

C code can contain optional Input Section directives, of the form

```
section("wave_data") <data object declaration>;
```

Continuing from the example, the C compiler attaches `.section wave_data` to the assembler code produced from this piece of code.

The Input Section directives do not need to be specific. The `section(pm)` directs the linker eventually to place the results of this code in a Memory Segment of type PM.

This section provides two code samples:

- [“Sample ADSP-2181 LDF File for C Code” on page C-3](#)
- [“Sample Hardware Overlay LDF File for C Code” on page C-6](#)

 If you write C code, you must allocate segments for the stack and heap in the LDF, as well as enforce stack and heap size limits. See the sample LDFs packaged with your VisualDSP++ distribution for models.

Listing C-1. Sample ADSP-2181 LDF File for C Code

```
ARCHITECTURE(ADSP-218x)
SEARCH_DIR( $ADI_DSP\218x\lib )
// specific code and data

//$LIBRARIES = ;
// Libraries from the command line are included in
// COMMAND_LINE_OBJECTS.
$OBJECTS = 218x_int_tab.doj , 218x_hdr.doj ,
$COMMAND_LINE_OBJECTS , libio.dlb , libc.dlb , 218x_exit.doj ;
```

Converting from .SYS to .LDF

```
// 2181 has 16K words (24-bit) of Program RAM and 16K words
// (16-bit) of Data RAM

MEMORY
{
    seg_inttab { TYPE(PM RAM) START(0x00000) END(0x0002f)
    WIDTH(24) }           // hardware interrupts
    seg_code   { TYPE(PM RAM) START(0x00030) END(0x037ff)
    WIDTH(24) }           // compiled code
    seg_data2   { TYPE(PM RAM) START(0x03800) END(0x03fff)
    WIDTH(24) }           // pm data
    seg_data1   { TYPE(DM RAM) START(0x00000) END(0x02fff)
    WIDTH(16) }           // dm data
    seg_heap    { TYPE(DM RAM) START(0x03000) END(0x037ff)
    WIDTH(16) }           // dynamic memory
    seg_stack   { TYPE(DM RAM) START(0x03800) END(0x03fdf)
    WIDTH(16) }           // stack space
}

PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
    SECTIONS
    {
        sec_inttab
        {
            INPUT_SECTIONS( $OBJECTS( IVreset ))           // 0x0000
            INPUT_SECTIONS( $OBJECTS( IVirq2 ))             // 0x0004
            INPUT_SECTIONS( $OBJECTS( IVirq11 ))            // 0x0008
            INPUT_SECTIONS( $OBJECTS( IVirq10 ))            // 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( IVpwrdown ))           // 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
// 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
}
}
```

Converting from .SYS to .LDF

Listing C-2. Sample Hardware Overlay LDF File for C Code

```
ARCHITECTURE(ADSP-2189)
SEARCH_DIR($ADI_DSP\218x\lib)
$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{
        dxv_vect_tbl{ // C Runtime Interrupt Vector Declarations
            INPUT_SECTIONS( $OBJECTS( IVreset ))// 0x0000 Reset vector
            INPUT_SECTIONS( $OBJECTS( IVirq2 ))// 0x0004 IRQ2 interrupt
            INPUT_SECTIONS( $OBJECTS( IVirq11 ))// 0x0008 IRQ11 interrupt
            INPUT_SECTIONS( $OBJECTS( IVirq10 ))// 0x000c IRQ10 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( IVpwrdsn ))// 0x002c Powerdown
                interrupt
        }
        >mem_vect_tbl

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

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

```

Converting from .SYS to .LDF

```
// 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
dxo_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.D0J(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.D0J(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.D0J(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.D0J(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.D0J(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)

// PM Overlay "Run" Segment Declarations
dxm_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.D0J(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.D0J(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
```

Converting from .SYS to .LDF

```
        PAGE_OUTPUT(PM_PAGE5.OVL) // output file name of overlay
                                   segment for linker
        INPUT_SECTIONS(PMOVLY5.D0J(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
```

Assembler .SEGMENT — LDF SECTIONS{} Conversion

The program placement part of the .SEGMENT declarations in the .SYS corresponded to the .SECTION directives in the assembly code. The assembler's .SEGMENT directive defined the memory type for each data item's placement with the memory type qualifier /dm or /pm.

This section provides two code samples:

[“Sample ADSP-2181 LDF File for Assembly Code” on page C-12](#)

[“Sample Hardware Overlay LDF File for Assembly Code” on page C-14](#)

In VisualDSP++, the INPUT_SECTIONS{} command in the LDF uses the .SECTION declarations in each assembly source file to place each data object in an Input Section. This .SECTION declaration names and defines the section of code that the linker then uses to place each Output Section into memory. All the mappings can be many-to-one.

- .SECTION labels in assembly source (obligatory for each data item) are mapped into corresponding Input Sections in the assembled object code. Multiple assembler code sections with the same label accumulate in an Input Section in the order they are assembled during a build.
- The LDF maps Input Sections into Output Sections, in the INPUT_SECTIONS (...) subset of the SECTIONS command. This “intermediate” mapping is useful for assembling analogous objects into one Output Section, using the LDF's macro expansion capabilities. For example, to build the wave_data Output Section from a combination of source code and library files, you would write

```
SECTIONS {
    ...
    all_wave_data
    { INPUT_SECTIONS( $OBJECTS(wave_data)
                     $LIBRARIES(wave_data) ) }
```

Converting from .SYS to .LDF

This command fragment places all your code objects declared as `.SECTION wave_data` and all the library files labeled `wave_data` (originally assembled with `.SECTION wave_data` labels) into Output Section `all_wave_data`. The Output Section name is declared at the beginning of the line(s) defining its constituents.

- Output Sections are mapped into Memory Segments in the top-level directives of the LDF's `SECTIONS` command. Continuing the example above, add the following command fragment to the preceding code.

```
... > all_wave_info;
```

At this point, the line places Output Section `all_wave_data`, built up as shown above, into Memory Segment `all_wave_info`. You could add another Output Section, `all_wave_references`, to `all_wave_info` in another line, simply by declaring its name and components, and appending

```
... > all_wave_info;
```

Listing C-3. Sample ADSP-2181 LDF File for Assembly Code

```
ARCHITECTURE(ADSP-2181)

SEARCH_DIR( $ADI_DSP\218x\lib )

$OBJECTS = $COMMAND_LINE_OBJECTS;

// 2181 has 16K words (24-bit) of Program RAM and 16K words
// (16-bit) of Data RAM
MEMORY
{
    seg_inttab { TYPE(PM RAM) START(0x00000) END(0x0002f)
                WIDTH(24) }
    seg_code   { TYPE(PM RAM) START(0x00030) END(0x03fff)
                WIDTH(24) }

    seg_data1  { TYPE(DM RAM) START(0x00000) END(0x00fff)
                WIDTH(16) }
```

```
    seg_data2 { TYPE(DM RAM) START(0x01000) END(0x01fff)
              WIDTH(16) }
} // end MEMORY

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

    SECTIONS
    {
        sec_inttab
        {
            INPUT_SECTIONS( $OBJECTS(interrupts) )
RESOLVE(begin, 0x0000)
        } >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

        // support for initialization, including C++
        sec_ctor
        {
            INPUT_SECTIONS( $OBJECTS(ctor) )
        } >seg_data1
    } // end SECTIONS
} // end PROCESSOR p0
```

Converting from .SYS to .LDF

Listing C-4. Sample Hardware Overlay LDF File for Assembly Code

```
ARCHITECTURE(ADSP-2189)
$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 (for PLIT)

    // 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 (for PLIT)

    // 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 memory mapped control
registers
```

```
// 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)

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

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

        dxv_int_dm{
            INPUT_SECTIONS(DM0vly0.doj(int_dmda))
            >mem_int_dm // places the file object "DM0vly.doj" in
                internal non-overlay DM segment "mem_int_dm"
                (DM0x2000-0x3fdf)

// DM Hardware Overlay Page Declarations (assign data modules to
// their "run address" memory regions)
dxv_dmpage{
    PAGE_INPUT{
        ALGORITHM(ALL_FIT)
        PAGE_OUTPUT(dm_page1.ovl) // assembler overlay output
            file for linker
        INPUT_SECTIONS(DM0vly1.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)
```

Converting from .SYS to .LDF

```
        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 Hardware Overlay Page Declarations (assign code modules
// to their "run address" memory regions)
dxp_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(PM0vly5.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

```

SYS .SEGMENT — LDF MEMORY{} Conversion

Each `.SEGMENT` directive in the `.SYS` is replaced by a `MEMORY` and `SECTION` command in the `.LDF`. The memory definition part of the `.SEGMENT` directive in the `.SYS` corresponds to a `MEMORY{}` command in the `LDF`, which specifies your system's physical memory. [Table C-1 on page C-18](#) shows how to transform each part of an `.SYS` segment directives to its equivalent `LDF MEMORY{}` command.

Here's a legacy `.SYS` file from the 6.1 tools release, which works for all of the 218x family with on-chip 16K byte PM/DM (or greater).

Listing C-5. Legacy SYSTEM File Example

```

.SYSTEM Example_2181_Architecture;
.ADSP2181;
.SEG/PM/RAM/ABS=0x0000/CODE/DATA    PM_INT[8192];
.SEG/PM/RAM/ABS=0x2000/CODE/DATA    PM_OVLY[8192];

.SEG/DM/RAM/ABS=0x0000/DATA         DM_OVLY[8192];
.SEG/DM/RAM/ABS=0x2000/DATA         DM_INT[8160];
.ENDSYS;

```

In contrast, the `.SYS` file syntax had a limited set of commands — a `.processor` declaration, followed by a series of `.SEGMENT` declarations to identify memory, type, address range and word size. No commands were available to specify to specify search paths, libraries, inputs, or multiple output files, nor was expression-handling syntax or macro expansion possible in `.SYS` files.

Converting from .SYS to .LDF

Table C-1. .SYS .SEGMENT/LDF MEMORY{} Conversion

.SYS Example: .segment	LDF Equivalent
<pre> /pm /ram /begin=A /end=B /width C NAME; (<i>segment name</i>) </pre>	<pre> MEMORY {  { TYPE(PM RAM)   ... } } </pre>
	<pre>  START(A) END(B) </pre>
	<pre>  WIDTH(C) </pre>
	<pre>  NAME </pre>
<pre> .SYS line: .segment /pm /ram /begin=A /end=B /width C NAME; LDF line: MEMORY{NAME { TYPE(PM RAM) START(A) END(B) WIDTH(C)}}... // .. more segment declarations ... } // End MEMORY command </pre>	

Legacy Assembly of Sources with Absolute Placement Directives

The `-Ao` (absolute resolve) switch directs the assembler to write resolve commands to the default or specified header Linker Description File (`.ldf`).

The assembler generates an LDF's `RESOLVE()` command for each `.VAR/ABS=address` directive in a legacy source program (a program developed under Release 6.1). The assembler outputs a `resolve.ldf` file, which you must manually include into your project's LDF via the `INCLUDE()` command.

You can use a default `resolve.ldf` file name, composed of the 'resolve_' prefix and the `.ldf` extension applied to the source filename, or override it with the `filename` parameter.

The following code examples show the translation of assembly `.VAR/ABS=` directives into corresponding `RESOLVE` commands.

Listing C-6. VAR/ABS Legacy Directives

```
/* Filename:  varAbs.dsp
Requires -legacy. Expect the creation of the LDF file.
*/

#define CMN_BASE0x010000

.module test;
nop;
.var/DM/ABS=CMN_BASE+0x20  eq_outp; // expect Resolve()
.var/DM/ABS=CMN_BASE+0x22  eq_outq; // expect Resolve()
.endmod;
```

When this code is assembled with

```
-easm218x -c -legacy varAbs.dsp -Ao resolve1.ldf
```

the assembler overrides `resolve_varAbs.ldf` with `resolve1.ldf`.

Legacy Assembly of Sources with Absolute Placement Directives



To successfully compile a `resolve.ldf`, the ‘InByte’ symbol in the `RESOLVE()` linker command must be declared global. This symbol, having no scope beyond a single module, does not appear in the symbol table.

Listing C-7. Generated RESOLVE() Commands

```
// Filename: resolve1.ldf
// These are assembler generated LDF RESOLVE() commands for
// -legacy .var directives with /ABS qualifiers of the previous
// assembly example.

// .var eq_outp in "varAbs.dsp", line 9:
RESOLVE( eq_outp, 0x10020 )

// .var eq_outq in "varAbs.dsp", line 10:
RESOLVE( eq_outq, 0x10022 )
```

For more information about the `RESOLVE()` command, see the corresponding section in [“PROCESSOR{}” on page 2-48](#). For more information about the `-Ao` assembler switch, refer to the *Visual++ 2.0 Assembler and Preprocessor Manual for ADSP-218x DSPs*.