ANALOG DEVICES

AN-572
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106 • 781/329-4700 • World Wide Web Site: http://www.analog.com

Overlay Linking on the ADSP-219x

By David Starr

OVERVIEW

This applications note is for software designers starting an ADSP-219x overlay design. Using this note and the information in the Linker and Utilities Manual for ADSP-21xx Family DSPs the following may be done:

Divide a program into overlays.
Write LDF files for simple and overlay links.
Write an overlay manager.

HARVARD VS. PRINCETON ARCHITECTURE AND THE LINKER

Early in the history of computing, the modern design of one memory to hold both data and program instructions acquired the name "Princeton." Such a machine is flexible because programs with lots of code and little data work, as well as programs with little code and lots of data. It's simple in that address is the only memory parameter of interest. Performance is adequate, even though it takes two memory cycle times to fetch data from memory: one cycle to fetch the instruction and the second cycle for the instruction to fetch the data.

A faster architecture has separate program and data memories. This is "Harvard" architecture, which improves speed because the machine can fetch program instructions and data in parallel; a data fetch from memory can thus be accomplished in a single memory cycle. All Analog Devices Inc. DSPs are Harvard architecture. They possess a separate program and data memory for speed and, for extra speed, data can be kept in program memory as well as data memory and there are instructions to fetch data from both memories simultaneously. To use the full memory bandwidth, the ADSP-219x family possesses an instruction cache. Instructions come from cache, freeing up the program memory bus for data fetch from program memory. This permits multiply accumulate instructions to fetch a multiplier and a multiplicand, compute a product, and add the new product to the accumulator, all in a single instruction. In this case, it is important to locate the multiplier data in program memory and the multiplicand data in data memory. It is the programmer's responsibility to assign data buffers to memory. This is done with instructions to the linker. Having a number of memories to deal with makes the DSP linker somewhat more complex than linkers for Princeton architecture machines.

In addition, each DSP project must fit into a different memory arrangement. Different DSP boards have different amounts of memory, located at different addresses. The ADSP-219x family supports an external memory interface, allowing rather large amounts of memory, but at a penalty in speed. Internal memory is ten times faster than external memory, so it may be desirable to keep large amounts of program code in external memory swap parts of it to internal memory for speed in execution. Such a program is said to run in "overlays."

For code reuse and portability, a program should not require modification to run in different machines or in different locations in memory. So the C or assembler source code does not specify the addresses of either code or data. Instead, the source code assigns names to *sections* of code and/or data at compile or assembly time, leaving it up to the linker to assign physical memory addresses to each section of code or data. The goal is to make the source program's position independent and let the linker assign all the addresses.

The address assignment task has become too much to handle with just command line switches to the linker. ADI has devised a "linker programming language" to allow full control of "what goes where." Each DSP software project consists of one or more source code modules and a "linker description file" (.ldf file).

At link time, the linker follows directions in the .ldf file to place code and data at the proper addresses. The "linker programming language" is quite powerful and fairly complex. The manual for it is as thick as the classic "C Programming Language" by Kernighan and Ritchie. The .ldf file can select which compiler libraries are searched, which C run time start-up code is linked, and can supply all command line arguments and override all defaults. A default .ldf file is shipped with each release of the VDSP tools, and kept in the VDSP directories. VDSP allows an .ldf file as part of a project, as does a C source file. If a project does not contain an .ldf file, the linker will use

the default one. If an .ldf file is included with the project, the project will link the same way each time, even if the default .ldf is edited or replaced by a new release of the VDSP tools, or the project is rebuilt on another computer with another version of the default .ldf file.

PHYSICAL MEMORY BLOCKS VS. LOGICAL MEMORY SECTIONS

What is the difference between a block and a section? A block is a named piece of the target system's memory. Blocks have attributes of address, size, and width (16-bit vs. 24-bit). Sections are named pieces of code or data. **.**SECTION directives in the source file mark code/data as belonging to one named section or another. Typical **.**SECTION names are "program," "dm_data," and "pm_data." Locate code and data at the desired address by putting code sections into memory blocks. The linker programming language uses ">" as the "put into" operator, (not to be confused with C's 'greater than' operator). The linker programming language has statements to define memory blocks, define code/data sections and marry the two together. When absolute addressing is needed (say to place an interrupt vector or to access memory-mapped I/O devices) define a memory block at the desired absolute address. Place the code or data into a **.**SECTION in the source file, and put the **.**SECTION into the memory block with the "put into" operator (">") in the .ldf file. The C compiler automatically places **.**SECTION directives into its output files. The compiler uses a series of default section names described in the compiler documentation.

THE LDF FILE

LDF files have two major parts (each delimited by curly brackets), the MEMORY part, and the PROCESSOR part. Statements inside the MEMORY part create memory blocks. The PROCESSOR part contains at least a SECTIONS command. The SECTIONS command uses more curly braces to enclose a number of statements. For example

```
MEMORY
{
    ram_blk {TYPE (PM RAM) START (0x000000)
END (0x00FFFF) WIDTH 24)}
}
PROCESSOR CORE1
{
      SECTIONS
      {
            dontcare0      {INPUT_SECTIONS
(frodo.doj (program))} > ram_blk
      }
}
```

This simple .ldf file defines one memory block (ram_blk) starting at 0 and ending at 64K. It defines one section named dontcare0 containing all the code in module frodo.doj, and puts the section into ram_blk, which starts at address 0. The section is named "dontcare0" to indicate that this name does NOT link to anything important. The name shows up in the link map but does not control which code goes where. This simple example has only one code module (frodo.doj). If the program had two modules, the INPUT_SECTIONS statement would look like:

```
dontcare0 {INPUT_SECTIONS
(frodo.doj,bilbo.doj(program)} > ram_blk
```

A practical program would, of course, contain a significant number of modules, and listing them in the INPUT_SECTIONS statement would be tiresome. The linker programming language provides a macro facility using syntax like the UNIX make program. A macro $OBJ is defined as:

```
$OBJ =
main.doj,frodo.doj,bilbo.doj,merry.doj,pippin.doj,
sam.doj,… ….  …..  gandalf.doj;
```

and the INPUT_SECTIONS statement might look like:

```
dontcare0 {INPUT_SECTIONS ($OBJ (program)}
 > ram_blk
```

and, finally, to simplify matters, the linker has a convenient and automatically defined macro $COMMAND_LINE_OBJECTS, which is written:

```
dontcare0 {INPUT_SECTIONS
($COMMAND_LINE_OBJECTS(program)} > ram_blk
```

In this way, modules can be added to the project without editing the .ldf file by hand each time a new module is added.

What does that (program) name do? It refers back to the source file. Assembly language module frodo.dsp looks like:

```
.section/CODE program

      ax0 = dm(this_and_that);
      I0 = input_buffer;
      ….
      ….
      rts;

.section/DATA dm_data;

.var   this_and_that;
.var   input_buffer[100];
      ….
```

This (program) means take only the code in the source module .SECTION-named program. In this example, ram_blk will receive just the program code and NOT the .var- defined data buffers. The linker matches the name in () with the names in the .SECTION pseudo ops in the source code. In this way code from one source file can be split into different sections, which are put into different memory blocks.

One final note. The linker programming language syntax does NOT use **;** as a statement terminator as does the assembler. Most linker programming language statements have a variable number of arguments enclosed in curly brackets. The closing curly bracket ends the statement. The syntax ignores white space (tab, space, CR, LF) in the same way that C does. The $MACROs, however, require a terminating semicolon.

A SIMPLE ASSEMBLER PROGRAM LINK

```
ARCHITECTURE(ADSP-219x)
// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
$OBJECTS =  $COMMAND_LINE_OBJECTS ;
MEMORY
{
   vector_blk   { TYPE(PM RAM) START(0x000000) END(0x0000ff) WIDTH(24) }
   program_blk  { TYPE(PM RAM) START(0x001000) END(0x06FFF) WIDTH(24) }
   dmdata_blk   { TYPE(DM RAM) START(0x008000) END(0x009fff) WIDTH(16) }
   pmdata_blk   { TYPE(PM RAM) START(0x07000) END(0x07FFF) WIDTH(16) }

}

PROCESSOR CORE1
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
        dont_care_0                     /* reset vector */
        {
                    INPUT_SECTIONS( $OBJECTS(IVreset))

        } > vector_blk
        dont_care_1                     /* code space  */
        {
            INPUT_SECTIONS( $OBJECTS(program) )
        } >program_blk

        dont_care_2                     /* data memory data */
        {
            INPUT_SECTIONS( $OBJECTS(dm_data) )
            INPUT_SECTIONS( $OBJECTS(dmdata) )
        } >dmdata_blk

        dont_care_3                     /* Program memory DATA */
        {
            INPUT_SECTIONS( $OBJECTS(pm_data) )
            INPUT_SECTIONS( $OBJECTS(pmdata) )
        } >pmdata_blk

    }                                   // end SECTIONS
}                                       // end PROCESSOR
```

Above is a simple .ldf for assembly work. It has four memory blocks, a vector block at 0 for the reset vector, a program code block, a program memory data block and a data memory block. All code in .SECTIONs named "program" from all object files in the project goes into program_blk starting at address 1000. This file links modules created by several programmers, some of whom used a .SECTION dmdata and others used .SECTION dm_data. No matter, both spellings are put into data memory, starting at 8000. Why 8000? This file is for an ADSP-2191 and data memory starts at 8000 in the ADSP-2191. Likewise we will accept two spellings of pmdata, one with and one without the underscore.

Notice program_blk and pmdata_blk both fit below 8000 hex. In the ADSP-2191, 24-bit program memory exists from address 0 to address 7FFF. You must store your program code there, and you can also store 16-bit data there. This .ldf file allocates 1000 words for data and 6000 words for code. A program with lots of pm_data and little code, will require reallocation of program memory, making pmdata_blk larger and program_blk smaller.

Or, a more flexible arrangement might be to eliminate pmdata_blk entirely and put INPUT_SECTION don't_care_3 into program_blk. Redefine program_blk to cover all of 24-bit program memory (0-7fff) and the linker will automatically allocate space for both code and data in the lower 16K of memory address space.

This .ldf file also has an ARCHITECTURE statement to tell the linker to link for the ADSP-219x family instead of the SHARC family, an OUTPUT statement that names the output file in accordance with command line input, and an LINK_AGAINST statement which is not necessary. There is one macro $OBJECT which serves as a short name for all the object modules passed in on the command line.

A SIMPLE OVERLAY LINK

```
/*
    Overlay link file
*/

ARCHITECTURE(ADSP-219x)
//SEARCH_DIR( .\ )
//MAP(adi_over.map)           /* write out a map file */

MEMORY
/*  Define Physical memory blocks. Assign a name, addresses and a width
to each type of physical memory in the target machine. Sections (below)
are "put into" ( > is the "put into" operator)  physical memory blocks
*/
{
reset_vector_blk{ TYPE(PM RAM) START(0x000000) END(0x00001f) WIDTH(24)}
int_vector_blk  { TYPE(PM RAM) START(0x000020) END(0x000aff) WIDTH(24)}
rootcode_blk    { TYPE(PM RAM) START(0x000B00) END(0x000cFF) WIDTH(24)}
ovlrun_blk      { TYPE(PM RAM) START(0x000D00) END(0x000dFF) WIDTH(24)}
ovlstore_blk    { TYPE(PM RAM) START(0x000E00) END(0x007FFF) WIDTH(24)}
dmdata_blk      { TYPE(DM RAM) START(0x001000) END(0x00Ffff) WIDTH(16)}
pmdata_blk      { TYPE(PM RAM) START(0x018000) END(0x01ebff) WIDTH(16)}

}

/*  Procedure Linkage Table (PLIT) template. The PLIT is a jump table
constructed by Linker in root memory. Each call to an overlay section is
replaced with a call to the PLIT. This template tells link what instructions to put into each PLIT
entry. This example passes parameters to the overlay manager (over_man) in registers. Keyword
PLIT must be all capitals, the linker is case sensitive.
*/

PLIT    {
    ax0 = PLIT_SYMBOL_OVERLAYID;     /* overlay number */
    jump     (PC,over_man);           /* call overlay manager */
}
```

```
PROCESSOR CORE1            // Single Core DSP example.
{
       $BOOT_MODULE = adi_over.doj;
       $ROOT_OBJECTS = adi_over.doj;
       LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
      OUTPUT( $COMMAND_LINE_OUTPUT_FILE ) // write out root.dxe

/* Tell the linker which .doj code goes into which section. Then
put the sections into physical memory blocks (> == put into operator)
*/
       SECTIONS
       {
/*     Create the reset vector. Get it from the Boot module's .SECTION
       IVReset. Put it into (>) reset_vector_block defined in the
       MEMORY section above. In this example the reset vector is a jump
       start instruction. Reset vector is always at program memory address 0.
       The section name is a don't care.
*/
       dont_care_0   {                       // Reset Vector
              INPUT_SECTIONS( $BOOT_MODULE(IVreset))

       } > reset_vector_blk

       /*      Create the root code. Get it from all the root .doj files
       .SECTION seg_rootcode. Put it into (>) rootcode_blk defined in the
       MEMORY section above. The section name is a don't care.
       */
              dont_care_1 {
              INPUT_SECTIONS      ($ROOT_OBJECTS      (program) )
              }>rootcode_blk

/*      Create the overlay branches. The section name is a don't care.
*/
        dont_care_2 {
       /*  Create overlay branch 1. Write it out to file ovl1.ovl. Get
       the code from ovl1.doj  .SECTION seg_ovlrun. Store it in
       ovlstore_blk.
       */
              OVERLAY_INPUT {
                     OVERLAY_OUTPUT (ov11.ovl)
                     INPUT_SECTIONS (ovl1.doj(program))
                     ALGORITHM(ALL_FIT)
                     } >ovlstore_blk

             OVERLAY_INPUT {
/*           Likewise, create overlay branch 2 */
                     OVERLAY_OUTPUT (ovl2.ovl)
                     INPUT_SECTIONS (ovl2.doj(program))
                     ALGORITHM(ALL_FIT)
                     } >ovlstore_blk
              OVERLAY_INPUT {
/*               Likewise, create overlay branch 2 */
                     OVERLAY_OUTPUT (ovl3.ovl)
                     INPUT_SECTIONS (ovl3.doj(program))
                     ALGORITHM(ALL_FIT)
                     } >ovlstore_blk

              } > ovlrun_blk                  // Link overlays for ovlrun_blk
```

```
/*     Create the Program Linkage Table (PLIT). Put it into rootcode_blk. .plit MUST be lower case,
the linker is case sensitive.
*/
              .plit  {} > rootcode_blk

/*      Create the Data Memory data blk. Get it from all the root .doj files .SECTION dmdata. Put
it into (>) dmdata_blk defined in the MEMORY section above. The section name is a don't care.
*/
              dont_care_3 {

                     INPUT_SECTIONS($ROOT_OBJECTS     (dmdata) )
              }>dmdata_blk

    }                                    // end SECTIONS
}                                        // end PROCESSOR Core1
```

Dividing the Program into Overlays

This example uses a simple memory arrangement. The program has a single root (code always resident) and a single overlay block, (ovlrun_blk) into which three overlay "branches" are swapped in and out as required. For simplicity, the overlays are stored in ordinary program memory (ovlstore_blk). In a real application, the inactive overlays would be stored in external memory or host memory, and swapped in via the PCI interface, or in the boot EPROM or elsewhere.

Following is the list of input sections. Input section don't_care_0 is the boot block, which must reside at address 0, the power-on reset in the ADSP-219x starts fetching instructions from address 0. The boot block contains a single "jump start" instruction. Input section dont_care_1 is the root segment which, in this simple example, is a single module adi_over.doj. All program code from the $ROOT_OBJECTS list of object files will go into rootcode_blk. Input section don't_care_2 is the overlay run area mapped into ovlrun_blk. Link each of the three branches to run in ovlrun_blk. Linking three different functions to the SAME address is something proper linkers are ordinarily reluctant to do, but inside the don't_care_2 are three OVERLAY_INPUT instructions, one for each overlay. The three instructions are identical except that they name three different output files and three different lists of modules.

Inside each OVERLAY_INPUT instruction an OVERLAY_OUTPUT instruction is needed, otherwise nothing will be written to disk. An INPUT_SECTIONS instruction is needed to tell linker which modules go into which overlay branch, and we need the ALGORITHM(ALL_FIT) instruction because the linker demands it be there. Each overlay branch is put into (.>) the ovlstore_blk telling the linker where to store the overlay. The three OVERLAY_INPUT instructions are nested inside the single don't_care_2 instruction, which is put into (>) the ovlrun_blk telling the linker to resolve the addresses inside the overlay branch for the runtime addresses rather than the storage time addresses.

Notice that this example divides code into overlays just by module name. This permits reshuffling of the overlay structure without editing the module source code. From assembler, code could be assigned to overlay branches with the .SECTION directives. However, if it is necessary to rearrange the overlay structure, the user would have to go into the module code and edit those .SECTION directives.

Calling the Overlay Manager

After successfully dividing the program into overlays, the user must make the overlays load. When a "call frodo" instruction is coded, it should branch into frodo if frodo is part of root, but it has to branch to an overlay manager (loader) if frodo is an overlay branch. To reach the overlay manager, the linker replaces calls to the overlay entrance with a call to the Procedure Linkage Table (PLIT). This table has one item per overlay. A PLIT item is a short user-written bit of code that calls the user-written overlay manager and passes arguments to it. The linker automatically creates a PLIT, assembles the necessary instructions from source code found in the .ldf file, and aims all the overlay calls into the proper place in the PLIT.

In the .ldf file there is a PLIT template, containing the source code, and a .plit{} > rootcode instruction telling the linker where to put the PLIT. Specifically, the PLIT had better reside in the root. If it is put into ovlrun_blk, the overlay manager will overwrite it when he loads to first overlay. This example has a simple PLIT template,

```
ax0 = PLIT_SYMBOL_OVERLAY_ID ;
jump (PC, over_man);
```

These two instructions will transfer control into the overlay manager, with the required overlay number in ax0. There are no restrictions upon the size or contents of the PLIT template. You can have any amount or kind of DSP code required. For instance, a DMA based overlay manager could load the DMA registers with the proper addresses in the PLIT and then branch to

over_man, in effect, passing the arguments via the DMA control registers rather than in ax0. One restriction, however, is that the PLIT keyword must be all capitals. The .plit instruction simply tells the linker where to put the PLIT, and that instruction must be all lower case.

Symbol PLIT_SYMBOL_OVERLAY_ID is a special linker symbol that will evaluate as "1" for the first overlay, "2" for the second overlay and so on.

Example Overlay Manager

Since the overlays might be loaded from anywhere, the overlay manager is user-written code. The following example is a simple one that can serve as a skeleton.

```
section/pm IVreset;
        JUMP start;                     /* power on reset vector */

.SECTION/PM program;

.GLOBAL      task_table,wait_loop;
.GLOBAL root_util;                      /* common utility routine in root */
.GLOBAL over_man;                       /* overlay manager */

.EXTERN      number_one_overlay;
.EXTERN number_two_overlay;
.EXTERN number_three_overlay;

start:
       m0=0;m1=1;m2=-1;m3=0;m4=0;m5=1;m6=-1;m7=0;
       L0=0;L1=0;L2=0;L3=0;L4=0;L5=0;L6=0;L7=0;

wait_loop:
              call number_one_overlay;
              nop;
         nop;
         call number_two_overlay;
         nop;
         nop;
              call number_three_overlay;
         nop;
         nop;
              I4 = number_one_overlay;
              I5 = number_two_overlay;
              I6 = number_three_overlay;
              jump(PC,wait_loop);

root_util:
              nop;
              nop;
              rts;

over_man:
/*
    Processing:    load overlays
    Inputs:        ax0 = overlay ID,
    Outputs:
*/
    ay0 = dm(ovl_id);
    ar = ax0 -ay0;
    dm(ovl_id) = ax0;                   /* update active ovl id #  */
    if ne call ovl_load;                /*  */
    i0 = dm(ovl_run_add);
    jump    (i0);
```

```
ovl_load:

    ar = dm(ovl_id);
    ar = ar -1;                          /* map 1,2,3... to 0,1,2... */
    sr = LSHIFT ar by 2 (LO);            /* four words per table entry */
    ay0 = ovl_table;

    ar = sr0 + ay0 ;                     /* ar = table base + 4 * ovl_id */
    I1 = ar;                             /* I1 points to overlay table entry  */
    M1 = 1;
    L1 = 0;                              /* Linear addressing */
    I0 = dm(I1,m1);                      /* I0 wptr points to ovl dest address*/
    dm(ovl_run_add) = I0;
    L0 = 0;                              /* Linear addressing */
    ar = dm(I1,m1);                     /*  cntr =  overlay size */
    dm(ovl_len) = ar;
    cntr = ar;
    I1 = dm(I1,M1);                      /*  I1 rptr points to ovl source address */

    do move_code until ce;
        ar = pm(i1,m1);
move_code:
        pm(I0,m1) = ar;
    rts;

.section/DATA   dmdata;

.var   task_table[4]=0,0,0,0;

.var    ovl_id = 0xffff;
.var    ovl_run_add;
.var    ovl_len;

.var    ovl_table[] =
        _ov_runtimestartaddress_1, _ov_word_size_live_1
       ,_ov_startaddress_1,_ov_word_size_run_1
       ,_ov_runtimestartaddress_2, _ov_word_size_live_2
       ,_ov_startaddress_2,_ov_word_size_run_2
       ,_ov_runtimestartaddress_3, _ov_word_size_live_3
       ,_ov_startaddress_3,_ov_word_size_run_3
    ;
```

The key to writing an overlay manager is ovl_table down in .SECTION dmdata. This program has three overlays. For each overlay, the linker will automatically evaluate four symbols with names starting _ov_.

These four symbols tell an overlay manager where to fetch the overlay, where to put it, how many reads to do, and how many writes to do. In this simple example, overlays are kept in 24-bit program memory, so the number of reads and writes is equal. If the overlays were stored in an 8-bit-wide EPROM you would have to do three reads for each write, hence the separate _word_size symbols.

The rest of over_man is pretty straightforward. First he checks to see if the desired overlay is already loaded (does ax0 = dm(ovl_id). If not (the usual case), he calls ovl_load. Ovl_load takes the overlay id number, maps it down from 1,2,3 to 0,1,2, and uses it to compute the address of the needed overlay's information in ovl_table. Then he does the copy and uses the register jump to access the newly loaded overlay.

E02362–1–1/01 (rev. 0)

PRINTED IN U.S.A.