

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

Using Software Overlays with the ADSP-219x and VisualDSP 2.0++
(Some assembly required...)

Last modified: February 6, 2002

Introduction

This application note will define and discuss the definition and uses for software overlays. Also, an in-depth discussion on overlay management techniques will be covered in an abstract level, as well as covering overlay management topics in further detail. Code examples will be provided to help illustrate these topics in detail as well.

What are Software Overlays?

As DSP software applications have grown more complex, system memory requirements have increased as a result of these newer applications. Because of this, an application may actually exceed the internal memory size of a particular DSP. This is where a software overlay system comes into play.

Also, in many cost-sensitive applications, it's to the hardware designer's advantage to design a system with the least expensive DSP (which typically means less on-chip memory is available.) Since the cost of bulk memory such as SRAMs and EPROMs are small compared to the cost of a DSP, it is sometimes more cost-efficient to have code and/or data reside in cheaper external memory. This is another scenario in which software overlays can be implemented in a system design.

Soft overlays are a "many to one" memory mapping system. Several overlays can 'live' (or

are stored) in unique locations in external memory, but they 'run' (or execute from) a common location in the internal memory of the DSP. Soft overlays are not physically present in the DSP's internal SRAM at all times; rather they are transferred/fetched into internal memory from external memory dynamically at runtime.

Say, for example, that your software system has 10 functions, all of which comprise a total of 120k words of Program Memory (PM), but your DSP only has a maximum of 32k PM locations. What do you do? With software overlays, you can fetch the desired function at runtime into the DSP's internal memory and then execute this function. Accessing code and/or data overlays dynamically gives you greater flexibility with your DSP's internal memory requirements.

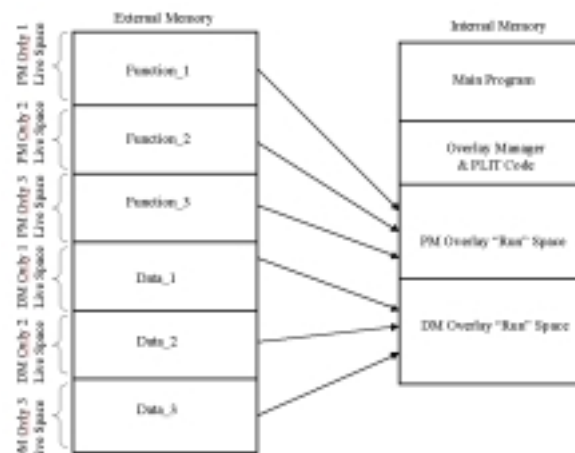


Figure 1: Simple Memory Overlay Example

Figure 1 demonstrates the concept of memory overlays. In this figure there are two total memory spaces; the internal memory of the DSP, and external memory. For this example, the external memory is partitioned into six overlays, comprised of three functions and three data buffers. The internal memory contains the

main program code, an overlay manager function, and two segments reserved for execution of overlay program instructions and data. From this example, we also see a “many to one” mapping, where Program Memory overlays 1, 2, and 3 map to the same overlay ‘run’ space. (Data overlays 4, 5, and 6 map to the Data Memory overlay ‘run’ space as well.)

In this example, overlays 1, 2, and 3 share the same runtime location within the DSP’s internal memory. If the main program calls the function ‘Function_1’, the overlay manager will be invoked to load overlay #1 into the memory segment within the DSP’s memory where overlay 2 has been designated to run. If the function ‘Function_3’ is requested by the main program, then the overlay manager will again be invoked to load overlay 3 into its designated run time memory segment. We will cover what the overlay manager’s role in a soft overlay system is and what an overlay manager is (and does) in more detail later on in this EE note.

Software Overlays for the ADSP-2191M

Software overlays are a very important software feature that can take advantage of the internal DSP memory resources and I/O bandwidth of the 2191’s external memory interface (EMI).

The ADSP-2191M has 32k words of Program Memory (PM) and 32k words of Data Memory (DM). Currently, there are also two additional 219x variants in the 219x family; the ADSP-2195 and ADSP-2196. The 2195 contains 16k words of PM and DM, while the 2196 contains 8k words for PM and DM, respectively. Because some software applications may require more memory than is available on-chip, software overlays become increasingly more important.

Another point to mention here is that the EMI of the 219x runs at a slower rate than the DSP’s

core. Therefore, executing code or fetching data from external memory will have an impact on overall system performance. The attractive feature of software overlays (for this case) is that you can execute code and access data via the DSP’s core, while simultaneously fetching and loading the desired overlay into internal memory in the background via the direct-memory access (DMA) controller of the ADSP-219x. *(For more detailed information on EMI throughput on the 219x family, please refer to table 7-10, page 7-26, of the “ADSP-219x/2191 DSP Hardware Reference”).*

What comprises a Soft Overlay?

Soft overlays have only a few attributes; an overlay ID#, ‘live’ address, ‘run’ address, ‘live’ size, and lastly, a ‘run’ size. Before explaining what these terms mean, let’s first talk about the two places where an overlay will exist in a system.

There are two terms associated with soft overlays; ‘live’ space and ‘run’ space. ‘Live’ space is the address range in external memory where the overlay resides. ‘Run’ space is the address range of the DSP’s internal memory where the overlay resides at runtime. *(For code overlays, the ‘run’ address is the target address of where the caller of the overlay will ‘jump’ or ‘call’ to in your program code. For data overlays, the ‘run’ address is the first location of the buffer or data type.)*

So getting back to the overlay’s attributes, the ‘run’ address is the address in the DSP’s internal memory where the code overlay will be executed from or where the data overlay will reside. The ‘live’ address is where in external memory the overlay will reside. One important point to mention here is that the ADSP-219x family supports up to 16M words of addressing (0x010000 – 0xfeffff) via its EMI; therefore the

‘run’ and ‘live’ addresses are 24-bit address fields.

The ‘run’ and ‘live’ size attributes define the size of the overlay module in words; for the ADSP-219x both of these values are the same. The last attribute is the overlay ID#. (An important note to mention here is that the run and live size attributes must not exceed 64k words in size, whether for a PM overlay or a DM overlay. This is because the EMI of the ADSP-219x will not automatically cross page boundaries.) This is a unique integer value which gets assigned to each overlay by the VisualDSP linker (linker.exe). *(The first overlay gets assigned an ID# of 1, the second gets assigned an ID of 2, etc.)*

All of these overlay attributes are linker-generated constants which will be used by our overlay manager. (We’ll cover this in much more detail later on in this application note.)

So you can see from the overlay attributes that soft overlays can reside at whatever internal memory ‘run’ space that you define; more than one overlay can map to a specific ‘run’ space. For more complex overlay managers and systems, a single overlay can map to more than one ‘run’ space also; we’ll cover this in more detail in the “Advanced Topics” section of this application note.

What is an Overlay Manager?

An overlay manager is responsible for controlling the fetching and loading of an overlay module into internal memory. For code overlay modules (or functions), the overlay manager is also responsible for telling the main program (or the ‘caller’ of the overlay function) the correct target address to ‘jump’ to after the overlay has been loaded. Also, an overlay manager is responsible for any housekeeping or

additional memory management required by the main program or calling function.

A simple model of an overlay manager would perform the following tasks:

- Identify the desired overlay module by getting the ID# of the overlay.
- Assign the appropriate live/run addresses and sizes to the DMA engine to properly load the overlay into internal ‘run’ space from external ‘live’ space.
- Invalidate and flush the instruction cache. (This is very important because we don’t want the overlay manager “polluting” the cache when we return back to our main program.)
- Return back to the main program or the ‘calling’ function of the overlay.

A more elaborate overlay manager would perform all of the above tasks as well as these additional tasks listed below:

- Perform a context save/restore of all of the registers used by the overlay manager (via a software stack located in the DSP’s internal Data Memory).
- Check to see if the requested overlay is already present in its ‘run’ space. If so, then jump to the target ‘run’ address of the overlay (if the desired overlay is a code overlay module), or return to the calling function of main program (if the desired overlay is a data overlay module.)
- Perform any memory management “housekeeping” tasks before returning to the main program or calling function.

An overlay manager should be written in an optimized manner to ensure that the minimum number of instruction cycles is required to execute it. The overlay manager is responsible

for managing the DSP's internal memory only; just like an Interrupt Subroutine (ISR), you want to spend the least amount of cycles in the overlay manager, and the majority of the processor's time running actual DSP code. Remember that we're developing code for a real-time system here! Another obvious point to mention here is that the overlay manager code should reside in the DSP's internal memory, not in an overlay run segment where it could get overwritten. The overlay manager also should not reside in external memory, since the latencies due to executing code through the EMI would incur too much system overhead.

VisualDSP Support for Overlays

The VisualDSP development tools generate overlay constants "automagically" which can be used by your overlay manager to configure the DMA parameter registers to load in the desired overlay module. Also, the VisualDSP linker automatically redirects overlay function calls to a jump table, called a PLIT (or Procedure Linkage Table; please refer to the section "What Is A PLIT?" on page 6 of this document for more information), which is used to setup the overlay ID# and overlay run address parameters which are passed from the PLIT to your overlay manager. Basically, the PLIT is just a function containing some user-generated assembly instructions that are used to setup the call to the overlay manager. We will explain the operation of the PLIT and where it is located later in this section.

The linker description file (LDF) is where the user defines the memory architecture of their system. It is within the LDF that you define both internal and external memory segments. Specifically for overlays, the LDF is where you define the 'live' and 'run' memory segments for each overlay module or file. (A complete explanation of the LDF is beyond the scope of

this EE note. Here we only wish to illustrate the LDF concepts that apply specifically to software overlays. For more information on LDFs, please refer to chapter 2 of the "Linker and Utilities Manual for ADSP-21xx DSPs".)

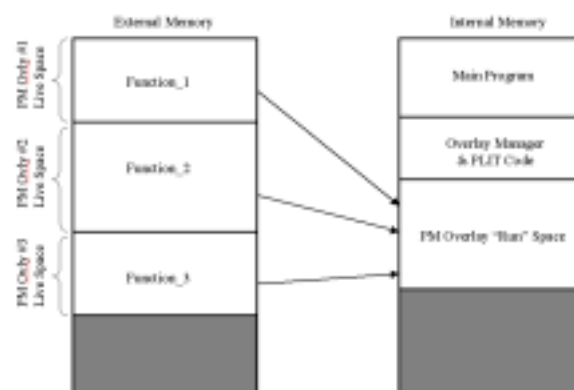


Figure 2: PM overlay 'live' address vs. 'run' address example

Figure 2 shows a simple software system where there are three PM overlays defined in their own individual 'live' segments in external memory. All three of these overlays 'run' in the same memory segment within the internal memory of the DSP. Let's look at an excerpt of what our LDF would look like for this example:

```
dxseg_pm_ovl{
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_1.ovl")
    INPUT_SECTIONS("Function_1.doj"(program))
  } >mem_pm_ovl1_liv_space

  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_2.ovl")
    INPUT_SECTIONS("Function_2.doj"(program))
  } >mem_pm_ovl2_liv_space

  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_3.ovl")
    INPUT_SECTIONS("Function_3.doj"(program))
  } >mem_pm_ovl3_liv_space
} >mem_pm_ovl_run_space
```

Example 1: LDF 'Live' and 'Run' Space Declarations

The first thing to notice is that each overlay declaration in the LDF has an input and an output section. The inputs to the overlay are declared within the scope of the overlay definition in the LDF via the curly braces. The main thing to be aware of is the use of the “INPUT_SECTIONS” LDF macro which tells the linker that this specific section from the specified input file is to be used as an input for this overlay segment.

The output of the overlay is defined using the redirect input symbol “>”; this redirection tells the linker where in memory to place this overlay (‘live’ space). For our example in Figure 2, the first overlay declaration links the object file “Function_1.doj” (which is the output file generated after assembling the source file “Function_1.asm”) into the overlay ‘live’ space named ‘mem_pm_ovl1_live_space’.

The overlay run space from Figure 2 is defined at the last line of this excerpt. Therefore, all three of the overlays declared in this section of the LDF are declared in the LDF to run in the overlay ‘run’ memory segment named “mem_pm_ovl_run_space”. The overlay ‘live’ and ‘run’ address segments are defined earlier in the MEMORY{} section of the LDF.

Linker Generated Overlay Constants

As mentioned earlier in this EE note, soft overlays have the following attributes; an overlay ID#, ‘live’ address, ‘run’ address, ‘live’ size, and lastly, a ‘run’ size. For each program memory overlay segment, the linker will generate the following constants, (where N is the ID# of the overlay):

```
_ov_startaddress_N
_ov_word_run_size_N
_ov_word_live_size_N
_ov_runtimestartaddress_N
```

Example 2: Linker Generated Overlay Constants Example

The first constant, “_ov_startaddress_N”, represents the ‘live’ or external address where the overlay resides. The second and third constants represent the ‘run’ and ‘live’ sizes of the desired overlay. For the ADSP-2191, these two values are (and *must*) be the same. The reason for this is simple; the hardware logic of the 2191’s EMI takes care of all of the data packing for you. So regardless of the external data bus width configuration (8-bits or 16-bits), or the memory access type (24-bit PM or 16-bit DM), the internal and external memory word sizes are the same as far as the ADSP-2191 is concerned. The last overlay constant represents the ‘run’ address where the overlay will reside in the DSP’s internal memory.

These linker generated overlay constants can be stored in arrays that can be accessed at runtime by your overlay manager to facilitate the loading of these overlays into internal memory. For example, if you had a system with two code overlays, you would declare the overlay constant arrays in a fashion like what is shown in the following example:

```
.section/pm seg_pmdata;

.extern _ov_startaddress_1, _ov_startaddress_2;
.extern _ov_word_run_size_1, _ov_word_run_size_2;
.extern _ov_word_live_size_1, _ov_word_live_size_2;
.extern _ov_runtimestartaddress_1, _ov_runtimestartaddress_2;

.var/init24 liveAddresses[2] = _ov_startaddress_1,
                               _ov_startaddress_2;

.var runAddresses[NUM_OVLY] = _ov_runtimestartaddress_1,
                              _ov_runtimestartaddress_2;

.var runWordSize[NUM_OVLY] = _ov_word_size_run_1,
                              _ov_word_size_run_2;

.var liveWordSize[NUM_OVLY] = _ov_word_size_live_1,
                              _ov_word_size_live_2;

.global liveAddresses;
.global runAddresses;
.global runWordSize;
.global liveWordSize;
```

Example 3: PM Buffer Declarations of Overlay Constants

From this example we see how these linker generated overlay constants are arranged for use by the overlay manager. The overlay constants are first declared as external data types to the scope of this file via the “.extern” assembler directive. This is necessary because these constants are generated by the linker in the overlay output file (.ovl) and are referenced from that file.

One very important point to mention here is with how the overlay ‘live’ address constants are defined. Since the ADSP-2191 has a 22-bit external address bus, and since the live space for the overlays in our system are declared to reside in external memory, the linker generated constants for the overlay ‘live’ addresses *must* be stored in a 24-bit PM buffer, and initialized using the “/init24” assembler directive.

(For this linker generated constants example, we’ve also declared all of these constant arrays as globals since the declaration of these arrays exists outside of the file which contains the overlay manager code, which references these arrays.)

What is a PLIT?

As defined by the “VisualDSP Linker and Utilities Manual for ADSP-21xx DSPs”, a PLIT is a template of instructions from which the linker generates code to set up the information necessary to support the DSP program’s overlay manager. Every branch instruction that references a global label defined in an overlay is replaced by a call to this generated code. For each overlay routine in the program, the linker builds and stores a list of PLIT instances according to that template, as it builds its executable.

The code for the PLIT is written by the programmer in the reserved PLIT section of the

linker description file (LDF) in the project. A simple PLIT merely copies the ‘run’ address of the called symbol that resides in the overlay and the overlay ID# into user-defined registers. Below is a simple PLIT example taken from an arbitrary overlay LDF:

```
PLIT{
    ax0 = PLIT_SYMBOL_OVERLAYID;
    ay0 = PLIT_SYMBOL_ADDRESS;
    jump Overlay_Manager;
}
```

Example 4: Simple LDF PLIT Entry Example

From the above example, we see that the registers ax0 and ay0 are used to fetch the overlay ID# and ‘run’ address, respectively.

A more practical example would be where the registers used by the PLIT function are saved off by the PLIT before consuming these registers to properly set up the overlay manager. Here is a simple example below:

```
PLIT{
    dm(save_ax0) = ax0;
    dm(save_ay0) = ay0;
    ax0 = PLIT_SYMBOL_OVERLAYID;
    ay0 = PLIT_SYMBOL_ADDRESS;
    jump Overlay_Manager;
}
```

Example 5: Context Save LDF PLIT Entry Example

Aside from containing user-defined code, the PLIT is also a section in the DSP’s Program Memory where the linker generates a jump table containing references to all of the overlay function labels. This jump table we refer to as a PLIT table. The next section explains how the linker adds additional code to facilitate the actual call and/or loading of the overlay function to allow the program sequencer to begin execution of the overlay code, and how the PLIT table is invoked and comes into play during program execution.

How is an overlay actually called?

In your source code, calling a function that resides in an overlay (in order to invoke the overlay manager to load the overlay code into the DSP's internal memory) is implemented in your source code in the same manner that you would call an ordinary (i.e. non-overlay) function. For example,

```
call my_function;
```

The difference between a non-overlay function call and an overlay function call is that the linker actually replaces the function call with a call to the PLIT entry for the desired overlay function. For example, an overlay function call from assembly like the following:

```
call function_1;
```

Actually gets replaced by a call to the PLIT by the following code:

```
call .plt_function_1;
```

Each overlay module that we declared in our LDF gets its own unique copy of the PLIT entry that was defined in our LDF. For example, let's say in our system we have three code overlays declared, and we have a simple PLIT declared in our LDF as shown in Example 4 above. Then the corresponding PLIT table for our three overlays would look like the following:

```
.plt_function_1:
  ax0 = 0x0001;
  ay0 = 0x0400;
  jump Overlay_Manager;

.plt_function_2:
  ax0 = 0x0002;
  ay0 = 0x0400;
  jump Overlay_Manager;
```

Example 6: Three Code Overlay Segments PLIT Table Example

```
.plt_function_3:
  ax0 = 0x0003;
  ay0 = 0x0400;
  jump Overlay_Manager;
```

Example 6: (continued)

Therefore, the total memory size of your PLIT table in the DSP's Program Memory will be the number of code overlays in your system multiplied by the number of assembly instructions contained in the PLIT{} declaration in your LDF.

Looking at Example 6 (and referring back to our PLIT declaration from Example 4, where the register ax0 is defined as containing the overlay ID number), we can see that each overlay has its own unique ID number; 'function_1' has the value '0x0001' as its overlay ID number, function 'function_2' has an ID of '0x002', etc. We also see that all three of these overlays share the same run address, which is passed as a parameter to the overlay manager in the register ay0. Lastly, the jump instruction for each PLIT table entry is a jump to the overlay manager itself.

Simple Overlay Manager Example

Example 7 below contains example assembly source code showing one implementation of a 'simple' overlay manager. This overlay manager example is broken up into six different sections; we'll discuss each section in detail to explain how this simple overlay manager example works.

```
Overlay_Manager:
  ar = ax0 - 1;
  m7 = ar;
  dm(curr_PM_ovly_ID) = ar;
  dm(WR_DMA_DESC_BLOCK+2) = ay0;

Get_Overlay_Run_Size:
  i7 = runWordSize;
  ax0 = dm(i7 + m7);
```

Example 7: Simple Overlay Manager

```

ar = ax0;
ar = ar - 1;
dm(WR_DMA_DESC_BLOCK+3) = ar;
dm(RD_DMA_DESC_BLOCK+3) = ar;

```

```

Parse_Live_Address:
i7 = liveAddresses;
ar = pm(i7 + m7);
sr = lshift ar by 0x8 (lo);
ar = px;
sr = sr or lshift ar by 0x0 (lo);
i7 = sr0;
ar = i7;
ar = ar + 1;

```

```

DMA_Config:
dm(RD_DMA_DESC_BLOCK+2) = ar;
dm(RD_DMA_DESC_BLOCK+1) = sr1;
ax0 = 0x800f;
dm(WR_DMA_DESC_BLOCK) = ax0;
ax0 = 0x800d;
dm(RD_DMA_DESC_BLOCK) = ax0;
ax0 = end_dma;
dm(WR_DMA_DESC_BLOCK+4) = ax0;
dm(RD_DMA_DESC_BLOCK+4) = ax0;
iopg = Memory_DMA_Controller_Page;
ax0 = WR_DMA_DESC_BLOCK;
io(DMACW_CP) = ax0;
ax0 = RD_DMA_DESC_BLOCK;
io(DMACR_CP) = ax0;
ax0 = 0x1;
io(DMACW_CPR) = ax0;
io(DMACR_CPR) = ax0;
io(DMACW_CFG) = ax0;
io(DMACR_CFG) = ax0;
ena int;

```

```

Wait_For_Overlay:
idle;

```

```

Jump_To_Overlay:
i7 = ay0;
jump (i7);

```

Example 7: Simple Overlay Manager (cont'd)

Let's take a look at this code section by section to determine what the overlay manager is actually doing. (Keep in mind that this is a simple overlay manager example; we'll build on this example and explain more overlay manager concepts in further detail later on in this EE note.)

The section "Overlay_Manager" is the address label where the overlay manager gets invoked

via a "jump Overlay_Manager;" instruction which is from the PLIT table.

Before we execute the first line of code from this overlay manager example, there is an important point to mention here first; the register ax0 contains the overlay ID# of the desired overlay that we wish to fetch. This point is important because we need to subtract 1 from the overlay ID# in order to use the register m7 as an index to the array of the linker generated overlay constants.

Think of this as the same manner you would with an array in C. The first element in a C array is actually index zero, even though it's still the first array element. (This is commonly known as an "off by one" error when indexing arrays in C; the same case applies here for our overlay constants and its appropriate ID value.)

Since the register ay0 already contains the 'run address' of the desired overlay (via the PLIT assignment, "ay0 = PLIT_SYMBOL_ADDRESS;"), we simply assign this value to the address field of our Write DMA Descriptor Block. (For more information on the configuration and use of DMA Descriptors, please refer to the sections titled "Descriptor-Based DMA Transfers" and "Code Example: Internal Memory DMA", on pages 6-4 and 6-33, respectively, of the "ADSP-219x/2191 DSP Hardware Reference".)

The next section of our overlay manager, labeled "Get_Overlay_Run_Size", is where we again use the m7 register as an offset into the 'runWordSize' array, which contains the 'run' size of the overlay we wish to fetch. One important point to mention here is that the overlay run size actually contains an additional word, which is the overlay ID number itself. Because of this, we must subtract one from the actual run size of the overlay stored in the 'runWordSize' array. This is performed in the

instruction “ar = ar – 1;”. Once we get the proper size for the overlay, we then place this value into the MEMDMA read and write descriptors at the last two instructions of this section.

In the next section, labeled Parse_Live_Address, we take the 24-bit ‘live’ address of the overlay and break it up into its respective 8-bit page value and 16-bit address offset. We implement this by taking the 16 MSBs of the 24-bit ‘live’ address value into the AR register, using AR as an input to the shifter register SR0, and then shifting this value 8 bit positions to the left, filling up the 8 LSBs of the SR1 register. We then “OR” the remaining 8-bits of the ‘live’ address that were stored in the PX register from the 24-bit PM fetch, to get the proper 16-bit address offset value. Figure 3 below shows the operation of this procedure to properly parse the live address into the appropriate page and offset values.

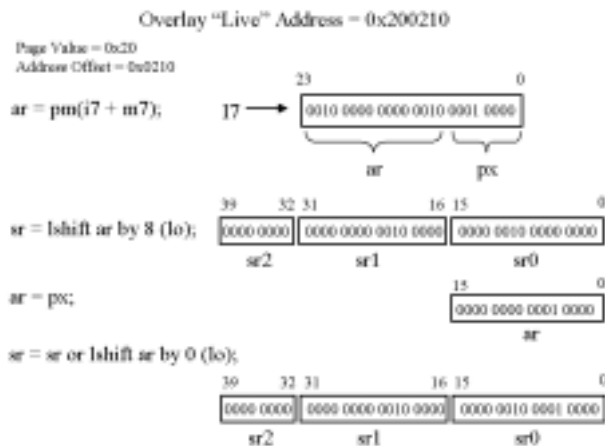


Figure 3: Parsing Overlay Live Address Example

(Remember that because the ADSP-2191’s opcodes are *always* 24-bits in length, the complete 24-bit address cannot be fully contained in a 16-bit register, nor can it be fully contained in an opcode. Therefore *all* external

memory addresses are broken up into an 8-bit page value and a 16-bit address offset.)

The next section labeled “DMA_Config” is where we initialize the remainder of the DMA Descriptors for the read and write DMA channels to kick off the DMA that will fetch the overlay from external memory and place it into the proper internal memory addresses. After kicking off the DMA, we simply sit at the “idle;” instruction and wait until we get a MEM DMA interrupt.

After servicing the interrupt, we return to execution of the overlay manager at the next instruction, which is in the section labeled, “Jump_To_Overlay”. In this section, we simply load the ‘run’ address of the overlay that we’ve just DMA’ed into internal memory into the I7 register, and then begin execution of the overlay via the “jump (i7);” instruction.

Overlay Manager Optimizations

The simple overlay manager example we just dissected is what its name implies; a “simple” overlay manager. In this section, we’ll talk about optimization techniques to make our overlay manager more robust for our system design.

The first thing to mention is that our overlay manager example didn’t perform a context save and restore of the registers that it corrupted. You have two options; you can both create and manage a software stack (used primarily by your overlay manager), or you can use the secondary computational and DAG registers. But, as with many things in life, there are trade-offs between both of these implementations.

Using the secondary registers is the fastest method for context switching, but since most real-time systems use the secondary registers for interrupt subroutines (ISRs) to minimize their

overhead, using these registers for your overlay manager may not be beneficial. Disabling interrupts for the sections of your overlay manager which use the secondary registers may not be practical unless you have a very low number of external interrupts occurring in your system or if your interrupts are deterministic (like the timer, for example.)

A software stack implementation is therefore more robust than using secondary registers, since it allows for the overlay manager to be interrupted during its execution. The overhead incurred by a software stack should only amount to the number of cycles needed to save and restore *only* the registers that are used by the overlay manager; saving all of the core registers is overkill.

```
Overlay_Manager:
    dm(ov_stack) = ax0;
    ax0 = dmpg2;
    dm(ov_stack+1) = ax0;
    dm(ov_stack+2) = ar;
    dm(ov_stack+3) = m7;
    dm(ov_stack+4) = i7;
    dm(ov_stack+5) = sr0;
    dm(ov_stack+6) = sr1;
    dm(ov_stack+7) = sr2;
    ax0 = iopg;
    dm(ov_stack+8) = ax0;
    dmpg2 = 0;
    ax0 = dm(ov_stack);
    ar = ax0 - 1;
    m7 = ar;
    dm(curr_PM_ovly_ID) = ar;
    dm(WR_DMA_DESC_BLOCK+2) = ay0;
    ...
```

Example 8: Overlay Stack Register Context Save Example

Example 8 above shows example code that we can add to the very beginning of the ‘simple’ overlay manager example that was given in example 7. Example 9 shows an implementation on performing a context restore of the registers used by our overlay manager from the overlay software stack:

```
...

Context_Restore:
    ax0 = dm(ov_stack+8);
    iopg = ax0;
    sr2 = dm(ov_stack+7);
    sr1 = dm(ov_stack+6);
    sr0 = dm(ov_stack+5);
    m7 = dm(ov_stack+3);
    ax0 = dm(ov_stack+1);
    dmpg2 = ax0;
    flush cache;

Jump_To_Overlay:
    i7 = ay0;
    jump (i7) (db);
    i7 = dm(ov_stack+4);
    ar = dm(ov_stack+2);
```

Example 9: Overlay Stack Register Context Restore Example

Another optimization implementation is to have your overlay manager check to see if the desired overlay function already resides in internal PM of the DSP. All that is needed to implement this is a data variable that can be updated by the overlay manager with the desired overlay’s ID number once the overlay has been loaded into the DSP’s internal memory. Also, some simple code to compare the overlay ID that we wish to fetch versus the overlay ID that already resides in the DSP’s internal memory is needed.

```
Is_Ovly_Loaded_Already:
    ax0 = m7;
    ay1 = dm(curr_PM_ovly_ID);
    ar = ax0 - ay1;
    dm(curr_PM_ovly_ID) = m7;
    if eq jump Goto_PM_Run_1;
    ...
```

Example 10: Overlay Load Verification Example

Example 10 above shows a simple method of comparing the overlay that we wish to fetch (stored in register ax0) with the current overlay ID number of the overlay that’s currently in the PM overlay ‘run’ segment of the DSP (stored in register ay1).

At the end of this code snippet, we simply compare registers ax0 and ay1 by subtracting

them; if they're the same, then subtracting them will result in a zero value. If the result is zero, then we jump to the section in the overlay manager code labeled "Goto_PM_Run_1", which is the section where the overlay manager loads the 'run' address of the overlay into register i7, which is used to begin execution of the overlay function via the "jump (i7);" instruction. If the result of subtracting the ax0 and ay0 registers is not zero, then we simply execute the overlay manager, like before, to load in the desired overlay via MEM DMA.

```
Goto_PM_Run_1:
    ax0 = dm(ov_stack+8);
    iopg = ax0;
    sr2 = dm(ov_stack+7);
    sr1 = dm(ov_stack+6);
    sr0 = dm(ov_stack+5);
    m7 = dm(ov_stack+3);
    ax0 = dm(ov_stack+1);
    dmpg2 = ax0;
```

flush cache;

```
i7 = ay0;
jump (i7) (db);
ar = dm(ov_stack+2);
i7 = dm(ov_stack+4);
```

Example 11: Overlay Manager for Overlay 'Run' Space 'Hit'

Example 11 above shows the portion of the overlay manager that gets executed if the desired overlay function has already been found to reside in the overlay 'run' memory space. Here we see that this code simply performs a context restore of the registers used by the overlay manager. Also, the 'run' address of the overlay function is found (from the 'runAddresses' buffer stored in DM), and this value is loaded into the i7 DAG register which is used to jump to the overlay function and begin its execution.

A Complete Overlay Manager Example

Now that we've broken up our "enhanced" overlay manager, let's list it here below in its entirety...

```
Overlay_Manager:
    dm(ov_stack) = ax0;
    ax0 = dmpg2;
    dm(ov_stack+1) = ax0;
    dm(ov_stack+2) = ar;
    dm(ov_stack+3) = m7;
    dm(ov_stack+4) = i7;
    dm(ov_stack+5) = sr0;
    dm(ov_stack+6) = sr1;
    dm(ov_stack+7) = sr2;
    ax0 = iopg;
    dm(ov_stack+8) = ax0;
```

```
Find_Overlay:
    dmpg2 = 0;
    ax0 = dm(ov_stack);
    ar = ax0 - 1;
    m7 = ar;
```

```
Is_Ovly_Loaded_Already:
    ax0 = m7;
    ay0 = dm(curr_PM_ovly_ID);
    ar = ax0 - ay1;
    dm(curr_PM_ovly_ID) = m7;
    if eq jump Goto_PM_Run_1;
```

```
Get_Run_Address:
    dm(WR_DMA_DESC_BLOCK+2) = ay0;
```

```
Parse_Live_Address:
    i7 = runWordSize;
    ax0 = dm(i7 + m7);
    ar = ax0;
    ar = ar - 1;
    dm(WR_DMA_DESC_BLOCK+3) = ar;
    dm(RD_DMA_DESC_BLOCK+3) = ar;
```

```
i7 = liveAddresses;
ar = pm(i7 + m7);
sr = lshift ar by 0x8 (lo);
ar = px;
sr = sr or lshift ar by 0x0 (lo);
i7 = sr0;
ar = i7;
ar = ar + 1;
dm(RD_DMA_DESC_BLOCK+2) = ar;
dm(RD_DMA_DESC_BLOCK+1) = sr1;
ax0 = 0x800f;
dm(WR_DMA_DESC_BLOCK) = ax0;
```

Example 12: Complete Overlay Manager Example

```

ax0 = 0x800d;
dm(RD_DMA_DESC_BLOCK) = ax0;
ax0 = end_dma;
dm(WR_DMA_DESC_BLOCK+4) = ax0;
dm(RD_DMA_DESC_BLOCK+4) = ax0;

```

Start_DMA:

```

iopg = Memory_DMA_Controller_Page;
ax0 = WR_DMA_DESC_BLOCK;
io(DMACW_CP) = ax0;
ax0 = RD_DMA_DESC_BLOCK;
io(DMACR_CP) = ax0;
ax0 = 0x1;
io(DMACW_CPR) = ax0;
io(DMACR_CPR) = ax0;
io(DMACW_CFG) = ax0;
io(DMACR_CFG) = ax0;
ena int;
idle;

```

Context_Restore:

```

ax0 = dm(ov_stack+8);
iopg = ax0;
sr2 = dm(ov_stack+7);
sr1 = dm(ov_stack+6);
sr0 = dm(ov_stack+5);
m7 = dm(ov_stack+3);
ax0 = dm(ov_stack+1);
dmpg2 = ax0;
flush cache;

```

Jump_To_Caller:

```

i7 = ay0;
jump (i7) (db);
i7 = dm(ov_stack+4);
ar = dm(ov_stack+2);

```

Goto_PM_Run_1:

```

ax0 = dm(ov_stack+8);
iopg = ax0;
sr2 = dm(ov_stack+7);
sr1 = dm(ov_stack+6);
sr0 = dm(ov_stack+5);
m7 = dm(ov_stack+3);
ax0 = dm(ov_stack+1);
dmpg2 = ax0;

```

flush cache;

```

i7 = ay0;
jump (i7) (db);
ar = dm(ov_stack+2);
i7 = dm(ov_stack+4);

```

Example 12: Continued

Advanced Tips and Tricks

Now that we've covered the basics as well as some more robust examples on how to add self-checking code into your overlay manager, let's take a look at some more advanced tips and tricks that can help improve the performance of your software overlay system. (The topics covered in this section apply to some special cases that can be used to your advantage when your software system meets these specific criteria.)

Having a greater number of overlay functions that are smaller in size has two main advantages. First, having smaller functions means that system latency will be minimized since the DMA controller has less data to fetch from external memory at any given time. (This is also important if your code requires the use of the external memory interface at the same time that a DMA may occur.)

Secondly, having smaller overlay functions means that you can have more than a single overlay 'run' space in the DSP. Having multiple run spaces not only allows for greater granularity with your memory partitioning, but also increases the associativity of your overlay run spaces, which can improve the amount of time that each overlay resides in internal DSP memory. Think of this example in the same manner as you would with an instruction cache; increasing the associativity of the cache (to a practical degree) will increase the cache hit performance ratio, since multiple code sections can reside within the cache at the same time. The same situation applies for overlays that reside in different run spaces of the DSP's program memory.

Multiple Overlay “Run” Segments

This next section will highlight an overlay example that uses two overlay ‘run’ space segments within the DSP’s internal program memory region. We’ll discuss how to configure the LDF and the overlay manager to implement this in detail as well.

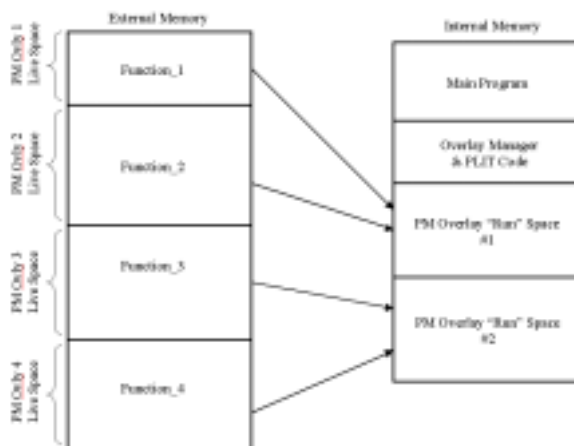


Figure 4: Two PM Overlay ‘Run’ Segments Example

Figure 4 shows a diagram depicting four overlay functions that reside in external memory. The DSP’s internal memory is partitioned into two distinct segments for the ‘run’ segments of the overlay functions. The overlay functions named “Function_1” and “Function_2” occupy the first overlay memory ‘run’ segment inside of the DSP at runtime, and the other two overlay functions, “Function_3” and “Function_4”, will execute from the second overlay ‘run’ segment.

Example 13 below shows an excerpt from our LDF file. Here we see the two internal memory segments that we define as the two distinct overlay ‘run’ segments; they’re labeled as `mem_ovl_run_space1` and `mem_ovl_run_space2`. For the ‘live’ address declarations of the overlay functions, we see that each function has its own external ‘live’ memory segment defined for it as well, therefore we have defined four external

‘live’ address regions for the four overlay functions.

```
MEMORY{
...
// "run" address regions for PM soft overlay functions
mem_ovl_run_space1{TYPE(PM RAM) START(0x000400)
  END(0x003fff) WIDTH(24)}
mem_ovl_run_space 2{TYPE(PM RAM) START(0x004000)
  END(0x007fff) WIDTH(24)}
...
// "live" address regions for PM soft overlay functions
mem_ovl1_liv_space{TYPE(PM RAM) START(0x200000)
  END(0x2000ff) WIDTH(24)}
mem_ovl2_liv_space{TYPE(PM RAM) START(0x200100)
  END(0x2001ff) WIDTH(24)}
mem_ovl3_liv_space{TYPE(PM RAM) START(0x200200)
  END(0x2002ff) WIDTH(24)}
mem_ovl4_liv_space{TYPE(PM RAM) START(0x200300)
  END(0x2003ff) WIDTH(24)}
}
```

Example 13: Two PM ‘Run’ Segments LDF Memory Example

Next, we’ll take a look at another excerpt from our LDF where we define which overlay functions get linked to which specific overlay ‘run’ segment within the internal memory of our DSP. Example 14 below shows how we implement this. We can see, for example how the overlay functions named “Function_1” and “Function_2” are declared within the scope of the memory segment “`mem_ovl_run_space1`”; this is how we map our overlay function objects (and their respective overlay ‘live’ segments) to the desired overlay ‘run’ segment internal to the DSP.

```
dxs_seg_pm_ovl1{
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_1.ovl")
    INPUT_SECTIONS("Function_1.doj"(seg_code))
  } >mem_ovl1_liv_space

  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_2.ovl")
    INPUT_SECTIONS("Function_2.doj"(seg_code))
  } >mem_ovl2_liv_space
}> mem_ovl_run_space1
```

Example 14: Mapping Overlay Run Spaces to Live Spaces

```

dxseg_pm_ovl2{
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_3.ovl")
    INPUT_SECTIONS("Function_3.doj"(seg_code))
  } >mem_ovl3_liv_space

  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Function_4.ovl")
    INPUT_SECTIONS("Function_4.doj"(seg_code))
  } >mem_ovl4_liv_space
}> mem_seg_pm_ovl2

```

Example 14: Continued

Next, we'll talk about what we need to do to our overlay manager in order for it to work with these additional 'run' segments. The first thing we need to do is define two data variables which will be responsible for containing the overlay ID for each of the respective overlay 'run' segments that these variables represent. We'll call them 'curr_PM_ovly_ID1' and 'curr_PM_ovly_ID2'. (We'll also initialize both of these variables to 0xffff, assuming that we're not going to have 65536 overlay functions for each run segment!)

The overlay manager needs to compare these two variables against the overlay ID number that we desire during runtime; if the desired ID matches one of the overlay ID numbers, then that means the desired overlay already resides within the DSP's memory, and therefore we simply jump to the run address of that overlay. Example 15 shows an excerpt from the overlay manager that performs this overlay ID checking:

```

Overlay_Manager:
  dm(ov_stack) = ax0;
  ax0 = dmpg2;
  dm(ov_stack+1) = ax0;
  dm(ov_stack+2) = ar;
  dm(ov_stack+3) = m7;
  dm(ov_stack+4) = i7;
  dm(ov_stack+5) = sr0;
  dm(ov_stack+6) = sr1;
  dm(ov_stack+7) = sr2;
  ax0 = iopg;
  dm(ov_stack+8) = ax0;

```

Example 15: ID Checking Overlay Manager Excerpt

```

Find_Overlay:
  dmpg2 = 0;
  ax0 = dm(ov_stack);
  ar = ax0 - 1;
  m7 = ar;

```

```

Is_Ovly1_Loaded_Already:
  ax0 = m7;
  ay0 = dm(curr_PM_ovly_ID1);
  ar = ax0 - ay0;
  if eq jump Goto_PM_Run_1;

```

```

Is_Ovly2_Loaded_Already:
  ay0 = dm(curr_PM_ovly_ID2);
  ar = ax0 - ay0;
  if eq jump Goto_PM_Run_2;

```

Example 15: Continued

If after executing this portion of the overlay manager we find that the overlay that we desire from our main program (or another calling function), then we must find the run address of the overlay function to determine which overlay run segment to place it. In example 16 below, at the label "Set_Ovly_ID", we know that the second overlay run segment starts at address PM 0x4000. Therefore, we assign this value to register ay0 and compare this to the run address of the desired overlay, which is stored in register ax0, by subtracting the two registers. If the result is zero, then we know that the desired overlay should be loaded into and execute from the second overlay run segment.

```

Get_Run_Address:
  i7 = runAddresses;
  ax0 = pm(i7 + m7);
  dm(WR_DMA_DESC_BLOCK+2) = ax0;
  dm(ov_stack+19) = ax0;

```

```

Set_Ovly_ID:
  ay0 = 0x4000;
  ar = ax0 - ay0;
  if eq jump Set_Ovly_ID2;

```

```

Set_Ovly_ID1:
  dm(curr_PM_ovly_ID1) = m7;
  jump Parse_Live_Address;

```

```

Set_Ovly_ID2:
  dm(curr_PM_ovly_ID2) = m7;

```

Example 16: Setting Overlay Run ID Variables

Since the size of the entire overlay manager example that we've discussed in this section is a bit too large to show here, please refer to Appendix A where this overlay manager is listed in its entirety.

Least Recently Used (LRU) Overlay Manager

For our next "advanced" overlay manager example, we'll discuss some topics that may not be suitable for every soft overlay application. But there are some useful techniques covered here to provide information for further performance benefits for your software overlay system.

First, you may have noticed that all the references to soft overlays thus far have been implicitly referring to code overlays only; nothing specific has been mentioned about using data overlays up to this point. At the time of this writing, the VisualDSP development tools don't properly support the initialization of external data overlays in memory. (Also, because the PLIT supports 'calls' to functions, it's not clear at this time how to implement a 'call' to a data segment; actually calling a data segment is meaningless. Therefore we will declare and initialize our data overlays as if they were just ordinary external memory segments.)

In most cases the data segments will be static; only the code will need to be dynamically loaded during runtime. But some systems may require the dynamic loading of data segments as well; for example, a non-deterministic system that needs to update its data segments at runtime due to a decision based upon reading in a value in an ISR that occurs at random intervals could take advantage of this implementation.

In this section, we'll discuss an example where we define two distinct overlay 'run' segments for code overlays, and two 'run' segments for

data overlays. The idea here is to use these different memory segments as cache memory, using the overlay manager as a crude cache controller.

For our example, we'll take advantage of a non-deterministic system, by applying an LRU scheme to our overlay manager to more efficiently make use of the two distinct overlay run segments for both code and data. The advantage of using an LRU scheme is that the code (or data) that you request most often will be stored in internal memory; the code or data that is used less-frequently will be replaced by the code or data that is requested more frequently. Therefore because the most-commonly requested data resides in memory, an increase in system performance is achieved.

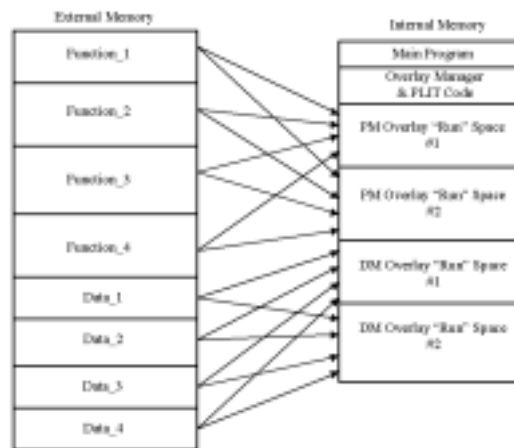


Figure 5: LRU Overlay 'Run' Segments Example

For this example, we'll have eight total overlay modules in our system; four code overlays and four data overlays. We'll also define two overlay 'run' segments for each of our code and data overlays, resulting in a total of four overlay 'run' segments residing in the internal memory of the DSP. Figure 5 above shows a conceptual block diagram of this software system. From this diagram, we can see how each overlay can map to any of its two possible overlay 'run'

spaces; this is quite different than the “Multiple Overlay Run Segments” example that we discussed in the previous section. The previous example had multiple overlay run segments which specified overlay modules were mapped to at link time. For this example, the placement of the overlay modules into the internal run segments is performed at runtime via an LRU memory replacement policy scheme.

One caveat to the LRU system that we wish to implement is that since we’re declaring our overlay functions as external functions versus declaring them as overlay functions, we no longer have the luxury of making the linker generate the linker generated overlay constants for us. Therefore, we must declare these constants and buffers manually, which can be quite cumbersome if a significant number of overlays are declared. (Currently, this is the only method for properly declaring data overlay segments, so there is no wasted effort here as far as data overlays are concerned.)

The ‘live’ and ‘run’ sizes for each of the overlays can be determined by the linker generated map file; therefore only the ‘live’ addresses for the overlay modules need to be calculated by hand. But this information is already known and defined in your LDF.)

Let us now take a look at our “LRU” overlay manager in detail. The first thing that we need to provide to the overlay manager are four memory variables used to store the overlay ID for each overlay ‘run’ region. We’ll declare these variables below:

```
.section/dm seg_data;
.var curr_PM_ovly_ID1 = 0xffff;
.var curr_PM_ovly_ID2 = 0xffff;
.var curr_DM_ovly_ID1 = 0xffff;
.var curr_DM_ovly_ID2 = 0xffff;
```

Example 17: Overlay Segment ID Variables

```
.global curr_PM_ovly_ID1;
.global curr_PM_ovly_ID2;
.global curr_DM_ovly_ID1;
.global curr_DM_ovly_ID2;
```

Example 17 (continued)

As with our previous overlay manager example, we’ll initialize these variables to the value 0xffff, since we’re assuming that there will be less than 64k (!) overlay modules defined for each overlay ‘run’ segment.

Next, our overlay manager can use these variables to determine whether the desired overlay already resides within its own ‘run’ space.

```
Is_Ovly_Loaded_Already:
    ar = dm(ov_stack);
    ar = ar - 1;
    ax0 = ar;
    dm(ov_stack) = ar;
    ay0 = dm(curr_PM_ovly_ID1);
    ar = ax0 - ay0;
    if eq jump Goto_PM_Run_1;

    ay0 = dm(curr_PM_ovly_ID2);
    ar = ax0 - ay0;
    if eq jump Goto_PM_Run_2;

    ay0 = dm(curr_DM_ovly_ID1);
    ar = ax0 - ay0;
    if eq jump DM_Context_Restore;

    ay0 = dm(curr_PM_ovly_ID2);
    ar = ax0 - ay0;
    if eq jump DM_Context_Restore;
```

Example 18: Test for Overlay ID Hit

Here we see that the ID number of the overlay that we desire is stored in register ax0, and the value of the overlay IDs that are present in their ‘run’ segments within the DSP are stored in register ay0. As before, we just subtract the two registers to determine whether we have a hit or not.

If the desired overlay ID doesn’t match any of the IDs of the overlays that already reside in the DSP’s memory, then the overlay manager must

then determine whether the desired overlay is a code or data overlay, and which overlay 'run' segment it can overwrite to load in this overlay module.

Determining whether the desired overlay is a code or data overlay is fairly simple, but there is a bit of user interaction required to achieve this. The user simply must list all of the attributes of the code overlays in the 'liveAddresses', 'runWordSize', and 'liveWordSize' buffers first before the data overlays. This way you know the total number of overlays in your system and the total number of code overlays in your system. Therefore, the difference between these two values is the overlay ID number of the first data overlay module. For example, if there are a total of fifteen overlays in your system and you know that ten of these overlays are code overlays, then the remaining five overlays are for data. Therefore, the overlay ID number of the first data overlay module would be eleven. We then can use this knowledge to find out if the desired overlay is a code or data overlay.

```
#define TOTAL_NUM_OF_PM_OVLYS 4
```

```
Find_Overlay:
    dmpg2 = 0;
    m7 = dm(ov_stack);
    dm(ovly_ID_temp) = m7;
    ay0 = TOTAL_NUM_OF_PM_OVLYS;
    ar = ax0 - ay0;
    if le jump Check_PM_LRU;
```

```
Check_DM_LRU:
    ...
```

Example 19: Code or Data Overlay Test Code Example

In example 19 above, we declared the total number of code overlays using the "#define" declaration at the top of the example. The ID number of the desired overlay is already stored in the ax0 register, so we simply subtract the two values to determine whether the desired overlay is a code (PM) or data (DM) module.

Now we must determine which overlay 'run' segment we should replace. This holds true whether the desired overlay is code or data. Here is where the LRU scheme comes into play. Since we now have two overlay 'run' spaces for code and two more for data, we need to define a total of four variables to define the LRU attribute for each overlay 'run' space.

```
.section/dm seg_data;
.var LRU_PM_Ovly_1 = 1;
.var LRU_PM_Ovly_2 = 1;
.var LRU_DM_Ovly_1 = 1;
.var LRU_DM_Ovly_2 = 1;

.global LRU_PM_Ovly_1 = 1;
.global LRU_PM_Ovly_2 = 1;
.global LRU_DM_Ovly_1 = 1;
.global LRU_DM_Ovly_2 = 1;
```

Example 20: LRU Variables For Overlay Run Segments

Assigning these LRU variables to one means that the specified overlay run segment that the LRU variable pertains to is "least recently used" and therefore can be overwritten with a new overlay module.

At this point, we've explained how the overlay manager determines how and where to load an overlay module into the DSP's internal memory, but we haven't yet touched on the subject of how the overlay manager exits itself to return to the normal program flow.

The next issue our overlay manager must deal with is exiting itself to either return to the main program, if the desired overlay we're requesting is data, or jump to the overlay function itself, if the desired overlay is program code. Here we have an underlying issue; since we call the overlay manager, a return address from the overlay manager is placed on the PC stack. (Remember that the address placed on the top of the PC stack after a call instruction is executed is the address of the instruction immediately following the call instruction.)

Therefore, in order to return to the normal program flow, we must execute an “rts;” instruction somewhere to restore the stack to its proper state. Having stated this, we now have two conditions; the first condition is if the desired overlay is a data overlay or it’s an overlay function that is not resident in an internal “run” space, then we simply can execute an “rts;” instruction at the end of the overlay manager to exit it.

The second condition is if the desired overlay is a function that is not resident in an internal “run” space. In this case, we need to DMA the overlay function into internal memory and then *jump* to this function from the overlay manager (instead of calling it). The only restriction here is that the overlay function *must* have an “rts;” instruction at the end of it to return our program flow back to the main program. Example 21 below shows how we can implement this type of scheme.

```
Jump_To_Caller:
    ax0 = dm(ov_stack);
    ay0 = TOTAL_NUM_OF_PM_OVLYS;
    ar = ax0 - ay0;
    if lt jump Goto_Target;
    rts (db);
    ax0 = dm(ov_stack+1);
    dmpg2 = ax0;

Goto_Target:
    dm(ov_stack+18) = i7;
    i7 = dm(ov_stack+19);
    jump (i7) (db);
    i7 = dm(ov_stack+18);
    nop;
```

Example 21: Overlay Manager Exit Routine

Putting It All Together

So after all this, we finally have our complete LRU Overlay manager. Example 22 below lists it in its entirety.

```
#define TOTAL_NUM_OF_PM_OVLYS 4
#include "2191asm.h"
```

Overlay_Manager:

```
ax0 = dmpg2;
dm(ov_stack+1) = ax0;
dm(ov_stack+2) = ar;
dm(ov_stack+3) = m7;
dm(ov_stack+4) = i7;
dm(ov_stack+5) = sr0;
dm(ov_stack+6) = sr1;
dm(ov_stack+7) = sr2;
ax0 = iopg;
dm(ov_stack+8) = ax0;
dm(ov_stack+9) = ay0;
```

Is_Ovly_Loaded_Already:

```
ar = dm(ov_stack);
ar = ar - 1;
ax0 = ar;
dm(ov_stack) = ar;
ay0 = dm(curr_PM_ovly_ID1);
ar = ax0 - ay0;
if eq jump Goto_PM_Run_1;
ay0 = dm(curr_PM_ovly_ID2);
ar = ax0 - ay0;
if eq jump Goto_PM_Run_2;
ay0 = dm(curr_DM_ovly_ID1);
ar = ax0 - ay0;
if eq jump DM_Context_Restore;
ay0 = dm(curr_DM_ovly_ID2);
ar = ax0 - ay0;
if eq jump DM_Context_Restore;
```

Find_Overlay:

```
dmpg2 = 0;
m7 = dm(ov_stack);
dm(ovly_ID_temp) = m7;
ay0 = TOTAL_NUM_OF_PM_OVLYS;
ar = ax0 - ay0;
if le jump Check_PM_LRU;
```

Check_DM_LRU:

```
ar = dm(LRU_DM_Ovly_1);
ar = ar - 1;
if eq jump Load_DM_Ovly_Run_1;
```

Example 22: LRU Overlay Manager Example

```

Load_DM_Ovly_Run_2:
    ax0 = 0xc000;
    dm(WR_DMA_DESC_BLOCK+2) = ax0;
    ax0 = 0x0000;
    dm(LRU_DM_Ovly_2) = ax0;
    ax0 = 0x0001;
    dm(LRU_DM_Ovly_1) = ax0;
    jump Load_DM_Ovly (db);
    ar = dm(ovly_ID_temp);
    dm(curr_DM_ovly_ID2) = ar;

```

```

Load_DM_Ovly_Run_1:
    ax0 = 0x9000;
    dm(WR_DMA_DESC_BLOCK+2) = ax0;
    ax0 = 0x0000;
    dm(LRU_DM_Ovly_1) = ax0;
    ax0 = 0x0001;
    dm(LRU_DM_Ovly_2) = ax0;
    ar = dm(ovly_ID_temp);
    dm(curr_DM_ovly_ID1) = ar;

```

```

Load_DM_Ovly:
    iopg = External_Memory_Interface_Page;
    ax0 = 0x0030;
    io(EMICTL) = ax0;
    iopg = External_Access_Bridge_Page;
    jump Parse_Live_Address (db);
    ax0 = 0x0000;
    io(E_STAT) = ax0;

```

```

Check_PM_LRU:
    ar = dm(LRU_PM_Ovly_1);
    ar = ar - 1;
    if eq jump Load_PM_Ovly_Run_1;

```

```

Load_PM_Ovly_Run_2:
    ax0 = 0x4000;
    dm(WR_DMA_DESC_BLOCK+2) = ax0;
    dm(ov_stack+19) = ax0;
    ax0 = 0x0000;
    dm(LRU_PM_Ovly_2) = ax0;
    ax0 = 0x0001;
    dm(LRU_PM_Ovly_1) = ax0;
    jump Load_PM_Ovly (db);
    ar = dm(ovly_ID_temp);
    dm(curr_PM_ovly_ID2) = ar;

```

```

Load_PM_Ovly_Run_1:
    ax0 = 0x0400;
    dm(WR_DMA_DESC_BLOCK+2) = ax0;
    dm(ov_stack+19) = ax0;
    ax0 = 0x0000;
    dm(LRU_PM_Ovly_1) = ax0;
    ax0 = 0x0001;
    dm(LRU_PM_Ovly_2) = ax0;
    ar = dm(ovly_ID_temp);
    dm(curr_PM_ovly_ID1) = ar;

```

Example 22: (continued)

```

Load_PM_Ovly:
    iopg = External_Memory_Interface_Page;
    ax0 = 0x0030;
    io(EMICTL) = ax0;
    iopg = External_Access_Bridge_Page;
    ax0 = 0x0008;
    io(E_STAT) = ax0;

```

```

Parse_Live_Address:
    i7 = runWordSize;
    ax0 = dm(i7 + m7);
    ar = ax0;
    dm(WR_DMA_DESC_BLOCK+3) = ar;
    dm(RD_DMA_DESC_BLOCK+3) = ar;
    i7 = liveAddresses;
    ar = pm(i7 + m7);
    sr = lshift ar by 0x8 (lo);
    ar = px;
    sr = sr or lshift ar by 0x0 (lo);
    i7 = sr0;
    ar = i7;
    dm(RD_DMA_DESC_BLOCK+2) = ar;
    dm(RD_DMA_DESC_BLOCK+1) = sr1;
    ax0 = 0x800f;
    dm(WR_DMA_DESC_BLOCK) = ax0;
    ax0 = 0x800d;
    dm(RD_DMA_DESC_BLOCK) = ax0;
    ax0 = end_dma;
    dm(WR_DMA_DESC_BLOCK+4) = ax0;
    dm(RD_DMA_DESC_BLOCK+4) = ax0;
    iopg = Memory_DMA_Controller_Page;
    ax0 = WR_DMA_DESC_BLOCK;
    io(DMACW_CP) = ax0;
    ax0 = RD_DMA_DESC_BLOCK;
    io(DMACR_CP) = ax0;
    ax0 = 0x1;
    io(DMACW_CPR) = ax0;
    io(DMACR_CPR) = ax0;
    io(DMACW_CFG) = ax0;
    io(DMACR_CFG) = ax0;
    ena int;
    idle;

```

```

Context_Restore:
    ay0 = dm(ov_stack+9);
    ax0 = dm(ov_stack+8);
    iopg = ax0;
    sr2 = dm(ov_stack+7);
    sr1 = dm(ov_stack+6);
    sr0 = dm(ov_stack+5);
    i7 = dm(ov_stack+4);
    m7 = dm(ov_stack+3);
    ar = dm(ov_stack+2);
    flush cache;

```

Example 22: (continued)

```

Jump_To_Caller:
  ax0 = dm(ov_stack);
  ay0 = TOTAL_NUM_OF_PM_OVLYS;
  ar = ax0 - ay0;
  if lt jump Goto_Target;
  rts (db);
  ax0 = dm(ov_stack+1);
  dmpg2 = ax0;

```

```

Goto_Target:
  dm(ov_stack+18) = i7;
  i7 = dm(ov_stack+19);
  jump (i7) (db);
  i7 = dm(ov_stack+18);
  nop;

```

```

Goto_PM_Run_1:
  ax0 = dm(ov_stack+8);
  iopg = ax0;
  sr2 = dm(ov_stack+7);
  sr1 = dm(ov_stack+6);
  sr0 = dm(ov_stack+5);
  m7 = dm(ov_stack+3);
  ax0 = dm(ov_stack+1);
  dmpg2 = ax0;
  ay0 = dm(ov_stack+9);
  flush cache;
  i7 = 0x0400;
  jump (i7) (db);
  ar = dm(ov_stack+2);
  i7 = dm(ov_stack+4);

```

```

Goto_PM_Run_2:
  ax0 = dm(ov_stack+8);
  iopg = ax0;
  sr2 = dm(ov_stack+7);
  sr1 = dm(ov_stack+6);
  sr0 = dm(ov_stack+5);
  m7 = dm(ov_stack+3);
  ax0 = dm(ov_stack+1);
  dmpg2 = ax0;
  ay0 = dm(ov_stack+9);
  flush cache;
  i7 = 0x4000;
  jump (i7) (db);
  ar = dm(ov_stack+2);
  i7 = dm(ov_stack+4);

```

```

DM_Context_Restore:
  ax0 = dm(ov_stack+8);
  iopg = ax0;
  sr2 = dm(ov_stack+7);
  sr1 = dm(ov_stack+6);
  sr0 = dm(ov_stack+5);
  m7 = dm(ov_stack+3);
  ax0 = dm(ov_stack+1);
  dmpg2 = ax0;

```

Example 22: (continued)

```

ay0 = dm(ov_stack+9);
flush cache;
rts (db);
i7 = dm(ov_stack+4);
ar = dm(ov_stack+2);

```

Example 22: (continued)

Pre-emptive Overlay Loading

Until now, we have only covered how to call or invoke an overlay when we need it. But in some cases, to increase system performance it may be beneficial to load in the overlay code or data *before* it is needed. Fortunately for us, we already have mechanisms built into our LRU overlay manager to implement this. The only part that requires some additional work on the programmer is to know what overlay ID number corresponds to which overlay code or data module.

The first thing that we need to do is pass the overlay ID number and the run address as input parameters to our overlay manager. If you remember from our PLIT entries, these two values are passed in the ax0 and ay0 registers, respectively. So now all we need to do is load the appropriate values into these two registers before invoking our overlay manager.

The next question is, “how do we decide which overlay ID number gets mapped to which overlay?” The answer is simple; the overlay ID numbers are assigned in the exact order that you define your overlays in the overlay input section of your LDF.

For an example, the LDF excerpt below declares three overlay functions named “Harry”, “Gus”, and “Joe”, contained in the files “Harry.asm”, “Joe.asm”, and “Gus.asm”, respectively. Since the overlay functions are declared in the exact order listed above, they will be assigned overlay ID numbers in that specific order as well. In other words, the code function “Harry” gets

assigned an ID number of 1, “Joe” an ID number of two, and so on.

```
dxseg_pm_ovl{
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Harry.ovl")
    INPUT_SECTIONS("Harry.doj"(seg_code))
  }>mem_ovl1_liv_space

  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Gus.ovl")
    INPUT_SECTIONS("Gus.doj"(seg_code))
  }>mem_ovl2_liv_space

  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Joe.ovl")
    INPUT_SECTIONS("Joe.doj"(seg_code))
  }>mem_ovl3_liv_space
}>mem_seg_pm_ovl
```

Example 23: Overlay Declarations and ID Mapping

The above LDF example explicitly links the input source file with the overlay input segment. Alternatively, these three overlay functions could be linked to the overlay input segment via the “\$OBJECTS” LDF macro. One important point to mention is that unique section names *must* be used in this case to specifically map the input file objects to the overlay input section. Example 24 below shows this in more detail:

LDF File Excerpt:

```
dxseg_pm_ovl{
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Harry.ovl")
    INPUT_SECTIONS($OBJECTS(seg_Harry))
  }>mem_ovl1_liv_space
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Gus.ovl")
    INPUT_SECTIONS($OBJECTS(seg_Gus))
  }>mem_ovl2_liv_space
  OVERLAY_INPUT{
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT("Joe.ovl")
    INPUT_SECTIONS($OBJECTS(seg_Joe))
  }>mem_ovl3_liv_space
}>mem_seg_pm_ovl
```

Harry.asm Excerpt:

```
.section/pm seg_Harry;
.global Harry;
Harry:
...
```

Gus.asm Excerpt:

```
.section/pm seg_Gus;
.global Gus;
Gus:
...
```

Joe.asm Excerpt:

```
.section/pm seg_Joe;
.global Joe;
Joe:
...
```

Example 24: LDF Overlay Mapping and Source File Segment Name Mapping

From the above example, we see that unique segment names are used in the “.section/pm” directive in each source file, as well as in the “INPUT_SECTIONS(\$OBJECTS(...))” LDF macro. This is how we explicitly map the objects from the input section(s) named “seg_Harry” into the overlay input section of the LDF. The same protocol is true for the other two segment names, “seg_Gus” and “seg_Joe”.

The “\$OBJECTS” macro is assigned a listing of all of the source files in the project via the “\$OBJECTS = \$COMMAND_LINE_OBJECTS;” LDF declaration. The “\$COMMAND_LINE_OBJECTS” LDF macro expands to a complete listing of all of the source files in the project.

Again, in this example, the function “Harry” would be assigned an overlay ID number of one, “Gus” an ID of two, and “Joe” and ID of three. This overlay ID number assignment is generated in the specific order in which each overlay is declared in the LDF. Now that we know the ID numbers for all of the overlays in our system, we can easily fetch them via the overlay manager.

Listed below is a simple macro definition for invoking the overlay manager and some additional definitions to make the code more readable:

```
#define Harry 0x0001 // ID # for 'Harry' function
#define Gus 0x0002 // ID # for 'Gus' function
#define Joe 0x0003 // ID # for 'Joe' function

#define Fetch_Overlay(x) \
    dm(save_ax0) = ax0; \
    ax0 = x; \
    dm(ov_stack) = ax0; \
    call Overlay_Manager; \
    ax0 = dm(save_ax0);
```

Example 25: Overlay Macro Definitions

These macros can then be used in your main program to pre-fetch or to perform a pre-emptive load of the overlay. For example:

```
Main:
...
Fetch_Overlay(Harry); // pre-fetch overlay function #1
...
```

Example 26: Using User-defined Macros for Overlay Support

Taking a closer look at this “Fetch_Overlay” macro, we see that it performs a few simple steps. First, it performs a context save of the ax0 register, then it reads in the overlay ID number of the desired overlay and stores this into the ax0 register. Next, the ax0 register is placed onto the overlay stack, where this value will be used by the overlay manager. After performing all of this housekeeping, a call to the overlay manager function is made.

Once the overlay manager has completed execution, the program will return to the “ax0 = dm(save_ax0);” instruction, where this macro completes by performing a context save of the ax0 register. From here the program can continue, and if necessary, can make a call to the

function “Harry” without any performance penalties, since this function already resides in its overlay run space.

Least Frequently Used (LFU) Overlay Manager

For our final “advanced” overlay manager example, we’ll discuss a different type of replacement policy than LRU. The LRU replacement policy takes advantage of your application code’s temporal locality. In other words, the code (or data) that you used most recently will be present in memory thanks to the exploitation of this replacement policy. The downside of this is that because of your instruction flow, you may have an overlay module that you fetch or call more frequently than others. With an LRU replacement policy, this overlay module could be overwritten with another overlay which is less frequently used (but more recently used), which in some cases may be undesired behavior.

A further optimization would be to keep the more frequently used overlay module resident in its respective overlay ‘run’ space. If we request a new overlay, we overwrite the least frequently used overlay that is resident in its respective overlay run space instead. Let’s show a simple instruction execution example below that will help illustrate this scenario:

Instruction	LRU		LFU	
	Run Space #1	Run Space #2	Run Space #1	Run Space #2
call overlay1;	overlay1	-----	overlay1	-----
call overlay2;	overlay1	overlay2	overlay1	overlay2
call overlay1;	overlay1	overlay2	overlay1	overlay2
call overlay3;	overlay1	overlay3	overlay1	overlay3
call overlay2;	overlay2	overlay3	overlay1	overlay2
call overlay4;	overlay2	overlay4	overlay4	overlay2
call overlay1;	overlay1	overlay4	overlay1	overlay2

Figure 6: LRU vs. LFU Replacement Policy Example

From figure 6 above, we see that we have a particular ordering to our instruction flow (which may or may not be deterministic). (Also, please note for this example that we have a two-way associative overlay ‘run’ space. In other words, any overlay can be mapped to one of the two overlay run spaces at run time.) From this example, we see that overlay 1 gets called three times, overlay 2 gets called two times, and overlays 3 and 4 get called only once. Because overlays 1 and 2 get called more frequently than the other two overlays, it would be beneficial (from a system performance perspective) to keep overlays 1 and 2 resident in their overlay run spaces more frequently than overlays 3 and 4.

From figure 6 we can see from the instruction flow which overlay module resides in the two overlay run regions at that specific point in time. Here we can clearly see that the LFU replacement policy takes advantage of the fact that overlays 1 and 2 are called more frequently, since we can see that they are present in the run spaces more times than with the LRU replacement policy.

So how does this all work? First off, we’ll need to keep a count of how many times each overlay is called. We can implement this very easily by keeping an array of count variables for each overlay module. Here is an example below:

```
.section/dm seg_data;
.var ovly_module_fetch_count[7] = 0, 0, 0, 0, 0, 0, 0;
```

Example 27: Overlay Fetch Count Array Initialization

For this example, let’s say that we have a total of seven overlay modules, comprised of four overlay functions and three data overlay modules. (Please note that this may be quite cumbersome if your project has an appreciable number of overlay modules.) This array will allow us to keep a running count for each

overlay module every time we request it. An array is used here (versus keeping separate count variables for each overlay module) to simplify things for our overlay manager, since we can use a DAG index register and a modify register (which contains the ID# of the desired overlay module) to stride through this array to increment the desired We initialize each element of this array to zero, since at the beginning of our program, none of the overlay modules have been requested.

Now we need to implement an LFU algorithm for our overlay manager. There are two steps required to implement this:

1. Automatically increment the overlay count value for the overlay module that we’re fetching. (We automatically increment the count value here because we know that since we’re fetching the overlay module that we’re going to immediately access it whether it resides in an overlay run space or not.)
2. Compare the overlay count values for each of the overlay modules that currently reside in both of the overlay ‘run’ memory regions. Next, replace the overlay that has the smaller overlay count value with the new overlay that we desire. (If both of the currently loaded overlays have the same overlay count value, then use the LRU replacement policy.)

Step one can be easily implemented. Listed below are the instructions needed to implement this:

```
Update_Overlay_Fetch_Count:
    i0 = ovly_module_fetch_count;
    m0 = dm(ov_stack);
    ar = dm(i0 + m0);
    ar = ar + 1;
    dm(i0 + m0) = ar;
```

Example 28: Updating Overlay Fetch Count

```
ar = tstbit 14 of ar;  
if eq jump Is_Ovly_Loaded_Already;
```

```
Scale_Down_Count:  
    m1 = 0;  
    m2 = 1;  
    cntr = TOTAL_NUM_OF_OVLAYS;  
    do Kill_Overflow until ce;  
        ar = dm(i0, m1);  
        sr = lshift ar by -1 (hi);  
Kill_Overflow:  
    dm(i0, m2) = sr1;
```

```
Restore_M1_Reg:  
    m2 = dm(ov_stack+13);  
    m1 = dm(ov_stack+12);
```

```
Is_Ovly_Loaded_Already:  
...
```

Example 28: (continued)

From the code snippet in example 24, we see that upon entering the overlay manager, we automatically increment the desired overlay's fetch count. But you'll also notice that there is some additional code placed here for error checking, starting at the address label called "Scale_Down_Count".

We have the additional code added here because we want to ensure that our count value doesn't become a negative value (MSB = 1), which would screw things up here because we're performing subtractions to compare the count values of two overlay modules. Here we want to ensure that we aren't subtracting a negative value, which would cause incorrect behavior with our LFU replacement policy. Actually, this portion of the overlay manager checks the value of bit 14 to ensure that our overlay count value doesn't exceed 16k (16384). If it does, then we simply scale down all of our overlay count values by 2 in the loop labeled "Kill_Overflow".

Step two of our LFU algorithm is also fairly straightforward. All we need to do is get the overlay count values for the two overlay

modules that currently reside in the two overlay run spaces for our example. We simply load these two values into two registers and subtract them to find the larger value. We then just replace the overlay run space that contains the overlay with the smaller overlay count value with the new overlay module that we desire.

But there is one caveat here; the first time running through our code, both of the overlay run spaces are empty (or more accurately, filled with illegal data) *and* the current overlay ID# variables for each of these overlay run regions are initialized to 0xffff. Since we use the overlay ID# variables as an offset for our DAG index register when we're initializing the DMA parameter registers, we *must* ensure that we're not running off into the weeds and are actually accessing the proper initialization data!!

The first thing we check for is if we're executing our program for the first time. If we are, then the current overlay ID# variables for both overlay run spaces will be equal. (This is the *only* time during program execution that these variables will contain the same value.) We simply compare these two variables and if both of these values are the same, we know that we're just starting the program and therefore we arbitrarily decide to load the desired overlay into the first overlay run space.

The code from example 25 below shows the portion of the overlay manager for fetching a Program Memory overlay code module, but the same algorithm applies for fetching a Data Memory overlay module as well.

If we're not executing the program for the first time, then we simply compare the count values for the two overlays that currently reside in their respective run spaces. Here we see that if

both overlay count values are equal, then we jump to the LRU subroutine; if one of the values is greater than or less than the other, then we simply jump to the subroutine to overwrite the least-frequently used overlay with the desired overlay via the DMA parameter registers.

```
Check_PM_Overlay_First_Time:
    ax0 = dm(curr_PM_ovly_ID1);
    ay0 = dm(curr_PM_ovly_ID2);
    ar = ax0 - ay0;
    if eq jump Load_PM_Ovly_Run_1;
```

```
Compare_PM_Ovly_Count:
    m0 = dm(curr_PM_ovly_ID1);
    ax0 = dm(i0 + m0);
    m0 = dm(curr_PM_ovly_ID2);
    ay0 = dm(i0 + m0);
    ar = ax0 - ay0;
    if lt jump Load_PM_Ovly_Run_1;
    if gt jump Load_PM_Ovly_Run_2;
```

```
Check_PM_LRU:
...
```

Example 29: LFU Comparison Algorithm Code Example

Putting It All Together (Again)

So after all this, we finally have our complete LRU Overlay manager. Example 26 below lists it in its entirety.

```
#define TOTAL_NUM_OF_OVLYS 7
#define TOTAL_NUM_OF_PM_OVLYS 4

#include "2191asm.h"

.section/pm_seg_code;

Overlay_Manager:
    ax0 = dmpg2;
    dm(ov_stack+1) = ax0;
    dm(ov_stack+2) = ar;
    dm(ov_stack+3) = m7;
    dm(ov_stack+4) = i7;
    dm(ov_stack+5) = sr0;
    dm(ov_stack+6) = sr1;
    dm(ov_stack+7) = sr2;
    ax0 = iopg;
    dm(ov_stack+8) = ax0;
```

Example 30: LFU Overlay Manager

```
dm(ov_stack+9) = ay0;
dm(ov_stack+10) = i0;
dm(ov_stack+11) = m0;
dm(ov_stack+12) = m1;
dm(ov_stack+13) = m2;
```

```
Get_Overlay_ID_value:
    ar = dm(ov_stack);
    ar = ar - 1;
    ax0 = ar;
    dm(ov_stack) = ar;

Update_Overlay_Fetch_Count:
    i0 = ovly_module_fetch_count;
    m0 = dm(ov_stack);
    ar = dm(i0 + m0);
    ar = ar + 1;
    dm(i0 + m0) = ar;
    ar = tstbit 14 of ar;
    if eq jump Is_Ovly_Loaded_Already;
```

```
Scale_Down_Count:
    m1 = 0;
    m2 = 1;
```

```
cntr = TOTAL_NUM_OF_OVLYS;
do Kill_Overflow until ce;
```

```
    ar = dm(i0, m1);
    sr = lshift ar by -1 (hi);
```

```
Kill_Overflow:
    dm(i0, m2) = sr1;
```

```
Restore_M1_Reg:
    m2 = dm(ov_stack+13);
    m1 = dm(ov_stack+12);
```

```
Is_Ovly_Loaded_Already:
    ay0 = dm(curr_PM_ovly_ID1);
    ar = ax0 - ay0;
    if eq jump Goto_PM_Run_1;
    ay0 = dm(curr_PM_ovly_ID2);
    ar = ax0 - ay0;
    if eq jump Goto_PM_Run_2;
    ay0 = dm(curr_DM_ovly_ID1);
    ar = ax0 - ay0;
    if eq jump DM_Context_Restore;
    ay0 = dm(curr_DM_ovly_ID2);
    ar = ax0 - ay0;
    if eq jump DM_Context_Restore;
```

```
Find_Overlay:
    dmpg2 = 0;
    m7 = dm(ov_stack);
    dm(ovly_ID_temp) = m7;
    ay0 = TOTAL_NUM_OF_PM_OVLYS;
    ar = ax0 - ay0;
```

Example 30: (continued)

if le jump Check_PM_Overlay_First_Time;

Check_DM_Overlay_First_Time:

```
ax0 = dm(curr_DM_ovly_ID1);
ay0 = dm(curr_DM_ovly_ID2);
ar = ax0 - ay0;
if eq jump Load_DM_Ovly_Run_1;
```

Compare_DM_Ovly_Count:

```
m0 = dm(curr_DM_ovly_ID1);
ax0 = dm(i0 + m0);
m0 = dm(curr_DM_ovly_ID2);
ay0 = dm(i0 + m0);
ar = ax0 - ay0;
```

```
if lt jump Load_DM_Ovly_Run_1;
if gt jump Load_DM_Ovly_Run_2;
```

Check_DM_LRU:

```
ar = dm(LRU_DM_Ovly_1);
ar = ar - 1;
if eq jump Load_DM_Ovly_Run_1;
```

Load_DM_Ovly_Run_2:

```
ax0 = 0xc000;
dm(WR_DMA_DESC_BLOCK+2) = ax0;
ax0 = 0x0000;
dm(LRU_DM_Ovly_2) = ax0;
ax0 = 0x0001;
dm(LRU_DM_Ovly_1) = ax0;
jump Load_DM_Ovly_1(db);
ar = dm(ovly_ID_temp);
dm(curr_DM_ovly_ID2) = ar;
```

Load_DM_Ovly_Run_1:

```
ax0 = 0x9000;
dm(WR_DMA_DESC_BLOCK+2) = ax0;
ax0 = 0x0000;
dm(LRU_DM_Ovly_1) = ax0;
ax0 = 0x0001;
dm(LRU_DM_Ovly_2) = ax0;
ar = dm(ovly_ID_temp);
dm(curr_DM_ovly_ID1) = ar;
```

Load_DM_Ovly:

```
iopg = External_Memory_Interface_Page;
ax0 = 0x0030;
io(EMICTL) = ax0;
iopg = External_Access_Bridge_Page;
jump Parse_Live_Address(db);
ax0 = 0x0000;
io(E_STAT) = ax0;
```

Check_PM_Overlay_First_Time:

```
ax0 = dm(curr_PM_ovly_ID1);
ay0 = dm(curr_PM_ovly_ID2);
ar = ax0 - ay0;
```

Example 30: (continued)

if eq jump Load_PM_Ovly_Run_1;

Compare_PM_Ovly_Count:

```
m0 = dm(curr_PM_ovly_ID1);
ax0 = dm(i0 + m0);
m0 = dm(curr_PM_ovly_ID2);
ay0 = dm(i0 + m0);
ar = ax0 - ay0;
if lt jump Load_PM_Ovly_Run_1;
if gt jump Load_PM_Ovly_Run_2;
```

Check_PM_LRU:

```
ar = dm(LRU_PM_Ovly_1);
ar = ar - 1;
if eq jump Load_PM_Ovly_Run_1;
```

Load_PM_Ovly_Run_2:

```
ax0 = 0x4000;
dm(WR_DMA_DESC_BLOCK+2) = ax0;
dm(ov_stack+19) = ax0;
ax0 = 0x0000;
dm(LRU_PM_Ovly_2) = ax0;
ax0 = 0x0001;
dm(LRU_PM_Ovly_1) = ax0;
jump Load_PM_Ovly_1(db);
ar = dm(ovly_ID_temp);
dm(curr_PM_ovly_ID2) = ar;
```

Load_PM_Ovly_Run_1:

```
ax0 = 0x0400;
dm(WR_DMA_DESC_BLOCK+2) = ax0;
dm(ov_stack+19) = ax0;
ax0 = 0x0000;
dm(LRU_PM_Ovly_1) = ax0;
ax0 = 0x0001;
dm(LRU_PM_Ovly_2) = ax0;
ar = dm(ovly_ID_temp);
dm(curr_PM_ovly_ID1) = ar;
```

Load_PM_Ovly:

```
iopg = External_Memory_Interface_Page;
ax0 = 0x0030;
io(EMICTL) = ax0;
iopg = External_Access_Bridge_Page;
ax0 = 0x0008;
io(E_STAT) = ax0;
```

Parse_Live_Address:

```
i7 = runWordSize;
ax0 = dm(i7 + m7);
ar = ax0;
dm(WR_DMA_DESC_BLOCK+3) = ar;
dm(RD_DMA_DESC_BLOCK+3) = ar;
i7 = liveAddresses;
ar = pm(i7 + m7);
sr = lshift ar by 0x8 (lo);
ar = px;
```

Example 30: (continued)

```

sr = sr or lshift ar by 0x0 (lo);
i7 = sr0;
ar = i7;

dm(RD_DMA_DESC_BLOCK+2) = ar;
dm(RD_DMA_DESC_BLOCK+1) = sr1;
ax0 = 0x800f;
dm(WR_DMA_DESC_BLOCK) = ax0;
ax0 = 0x800d;
dm(RD_DMA_DESC_BLOCK) = ax0;
ax0 = end_dma;
dm(WR_DMA_DESC_BLOCK+4) = ax0;
dm(RD_DMA_DESC_BLOCK+4) = ax0;

iopg = Memory_DMA_Controller_Page;
ax0 = WR_DMA_DESC_BLOCK;
io(DMACW_CP) = ax0;
ax0 = RD_DMA_DESC_BLOCK;
io(DMACR_CP) = ax0;
ax0 = 0x1;
io(DMACW_CPR) = ax0;
io(DMACR_CPR) = ax0;
io(DMACW_CFG) = ax0;
io(DMACR_CFG) = ax0;

ena int;
idle;

Context_Restore:
ay0 = dm(ov_stack+9);
ax0 = dm(ov_stack+8);
iopg = ax0;
sr2 = dm(ov_stack+7);
sr1 = dm(ov_stack+6);
sr0 = dm(ov_stack+5);
i7 = dm(ov_stack+4);
m7 = dm(ov_stack+3);
ar = dm(ov_stack+2);

flush cache;

Jump_To_Caller:
ax0 = dm(ov_stack);
ay0 = TOTAL_NUM_OF_PM_OVLAYS;
ar = ax0 - ay0;
if lt jump Goto_Target;
rts (db);
ax0 = dm(ov_stack+1);
dmpg2 = ax0;

Goto_Target:
dm(ov_stack+18) = i7;
i7 = dm(ov_stack+19);
jump (i7) (db);
i7 = dm(ov_stack+18);
nop;

```

Example 30: (continued)

```

Goto_PM_Run_1:
ax0 = dm(ov_stack+8);
iopg = ax0;
sr2 = dm(ov_stack+7);
sr1 = dm(ov_stack+6);
sr0 = dm(ov_stack+5);
m7 = dm(ov_stack+3);
ax0 = dm(ov_stack+1);
dmpg2 = ax0;
ay0 = dm(ov_stack+9);

flush cache;

i7 = 0x0400;
jump (i7) (db);
ar = dm(ov_stack+2);
i7 = dm(ov_stack+4);

Goto_PM_Run_2:
ax0 = dm(ov_stack+8);
iopg = ax0;
sr2 = dm(ov_stack+7);
sr1 = dm(ov_stack+6);
sr0 = dm(ov_stack+5);
m7 = dm(ov_stack+3);
ax0 = dm(ov_stack+1);
dmpg2 = ax0;
ay0 = dm(ov_stack+9);

flush cache;

i7 = 0x4000;
jump (i7) (db);
ar = dm(ov_stack+2);
i7 = dm(ov_stack+4);

DM_Context_Restore:
ax0 = dm(ov_stack+8);
iopg = ax0;
sr2 = dm(ov_stack+7);
sr1 = dm(ov_stack+6);
sr0 = dm(ov_stack+5);
m7 = dm(ov_stack+3);
ax0 = dm(ov_stack+1);
dmpg2 = ax0;
ay0 = dm(ov_stack+9);

flush cache;

rts (db);
i7 = dm(ov_stack+4);
ar = dm(ov_stack+2);

Example 30: (continued)

```