Engineer To Engineer Note EE-151

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

Implementing Software Data Overlays for the ADSP-21161 Using the EZ-KIT Lite

Last Modified: *1/25/02*

Contributed by: *John Tomarakos*

Overview

In many DSP applications, there may be system memory requirements where the DSP programmer wishes to assign a section of internal "scratchpad" memory to use for multiple temporary variables, then store the variables back to external SDRAM or FLASH memory for later usage. Often, variables arrays, or lookup tables are only used during a small fraction of time of the DSP's processing, but they often can take up a large portion of the internal memory space of the DSP. Or, an audio system developer may want to copy a block of recorded audio samples to storage and then retrieve the data for playback at a later time. Once the audio data is stored, the internal memory is free to use for other operations. Another example could be an application which requires using a temporary section of shared memory for sine/cosine lookup tables or FFT twiddle factors which are only required when the FFT algorithm is executed, but then perhaps during the FFT algorithm's "down time" the user wants to use the same memory for loading one out of a number of stored sets of filter coefficients for other post-filtering operations, only requiring one selected filter response at any given time upon request from a user-controlled interface. And finally, another DSP application could be a music synthesizer, where the DSP is performing a wavetable synthesis or sample playback algorithm, and when the musician selects a new "instrument" on command from a set of control knobs, telling the system's host microcontroller to direct the DSP to download a new block of wavetable data to the same section of memory to generate the new sound. To support such memory management tasks we can set up what is called a "*soft data overlay*" to automate the process of reusing a small section of internal memory efficiently.

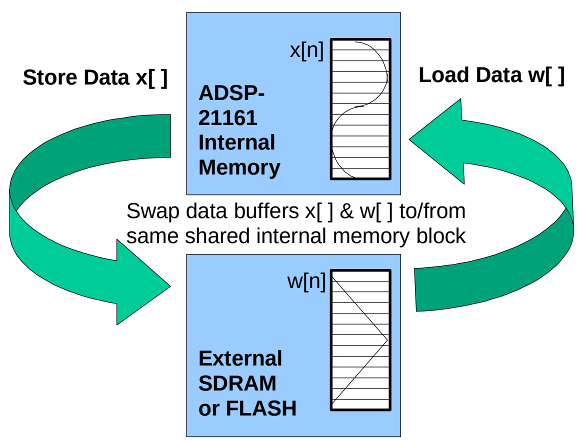


FIGURE 1: Basic Data Overlay Concept

Figure 1 demonstrates a basic concept of a soft data overlay, where information is loaded from external memory to a shared section if internal memory, while the previous data is stored simultaneously to external memory and recalled at a later time. This data overlay technique may provide support for applications where the total amount of data memory required is greater than the available amount of internal data memory provided by the processor, and the user is seeking a way to "make the code and data fit into the DSP's internal memory space."

The data overlay "*save-and-restore*" approach may be desirable because by keeping all the data buffers outside in the external memory and operating on them there, it can result in a significant reduction in available MIPS, due to external waitstates or SDRAM latencies. MAC operations can slow down significantly if the delay line data or filter coefficients are accessed from external memory, and the cycle overhead can also increase significantly depending on SDRAM latencies if we access non-consecutive addresses. However, the SHARC processor may often provide much available non-utilized I/O processor bandwidth to take advantage of. During these unused IOP bus cycles it would be desirable to initiate data overlay transfers which would execute with zero-overhead to internal memory from SDRAM, or vice versa. Perhaps the application is performing 5 active serial port DMAs or link port DMAs, but this still provides much IOP bus bandwidth available for quick 100 MHz external port SDRAM block transfers on a dedicated EPBx DMA channel. 100 MHz SDRAM DMA transfer throughput with 32-bit fixed or floating point data can make the use of data overlays attractive because the SDRAM controller is able to sustain 100 MHz throughput in consecutive reads/writes to the same page in SDRAM (of course, as long as there is no other core, host or EPBx DMA activity to slow down access to the external bus). Therefore, the DSP core can initiate a fast data overlay load, then go and execute another task in internal instruction space while the data is quickly loaded by the I/O processor in the background with no core intervention.

The advantage of data overlays to SDRAM is that SDRAM provides a low-cost, bulk-storage, high-speed interface to move the blocks of data at up to 100 MHz throughput. The disadvantage is that a system requires external memory in addition to a boot flash device, increasing system cost. If the system does not necessarily need fast access to swap data blocks, an external boot flash used for data overlays can could meet system cost constraints. It is possible to still use the unused upper sectors of the flash device which is not used for boot program storage to be able to temporarily load and store samples, or filter coefficients or data tables. In this case, we can set up data overlays with the necessary 8-bit DMA packing and flash memory commands. Table 1 below shows the tradeoffs for implementing data overlays in flash or SDRAM.

TABLE 1. DATA OVERLAY SDRAM/Boot FLASH comparisons:

MEMORY Type	Data Overlay Throughput	Bus Width	System Cost
SDRAM	100 MHz transfer throughput possible	32-bit	May require both SDRAM and boot memory
SBSRAM	50 MHz fixed transfer throughput	32-bit	May require both SBSRAM and boot memory
FLASH	Could require up to 7 CLKIN waitstates, slowest data throughput	8/16-bit	Only Boot Flash Required, can be mapped to both /BMS and /MSx space

So then the question becomes, how can we use the current VisualDSP++ code generation tools to provide the ability to load and store data variables and buffers? As one suggested solution, this Engineer-To-Engineer Note discusses a simple technique using Analog Devices' Visual DSP++ 2.0 tools and it's code overlay support to implement "*Data Software Overlays*."

The source code listing in this EE-Note demonstrates a way to accomplish swapping data to and from SDRAM using the 21161 EZ-KIT Lite Evaluation Platform (Figure 2) as the demonstration vehicle. For more information on implementing software overlays, refer to EE-Note # 66: *Using Memory Overlays.* Keep in mind the example demonstrated here is just a sample of one process that may be required in an actual application. Depending on the user's application, further customization of the overlay process may be required to suit your needs (through the modification of the overlay manager discussed later in this document).

FIGURE 2: ADDX-21161-EZ-LITE

The current VisualDSP 2.0++ tools support software code (instruction) overlays but no direct data overlay support built into the linker. By using some of the existing code overlay support methods supplied with our tools, we can alternatively develop a data overlay routine which copy the data variables and buffers from external memory to internal memory, and then back to the external memory after the variables have been modified. This is taken care by a routine called the *data overlay manager*, but when using data overlays (in the current VisualDSP++ 2.0 toolset) the data movement process is still required to be initiated by the programmer in order to start the DMA transfer (vs. code overlays, which are a more automatic process with the linker generating the necessary jump instructions via access to a jump table).

As mentioned earlier, it is also possible for the ADSP-21161 programmer to perform data overlays to flash memory. This can be done on the flash memory used on the EZ-KIT Lite, by simply modifying the overlay manager to perform 8-bit DMA accesses to flash memory while including all of the necessary housekeeping flash instructions in with the data overlay manager. For more information on programming the EZ-KIT flash, refer to EE-150*: In-circuit programming a boot-image into the FHASH on the ADDS-21161N-EZLITE*. Look for an additional update to this EE-NOTE to include data overlays to/from flash memory using the EZ-KIT Lite.

SDRAM Data Overlay System Design Assembly and C Examples:

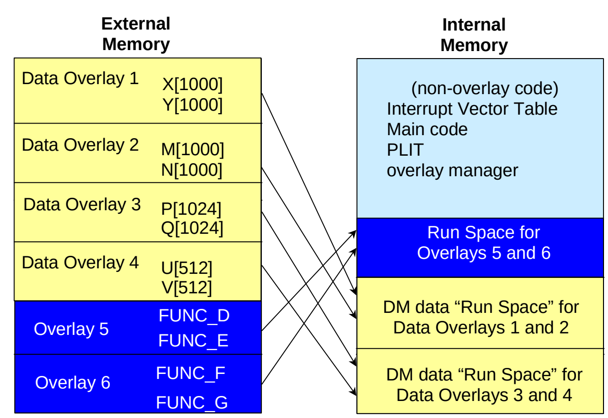


FIGURE 3: Data Overlay Live and Run Segments

Like code overlays, data overlays are a "many to one" memory mapping system. For example, Figure 3 shows how several data overlays "live" (or are stored) in unique external memory locations, but then "run" (or are executed/accessed) in a common shared location in internal memory. This demonstrates an application in which there are six "live" segments in external SDRAM. Four segments are data overlay segments, while the other two are code overlay segments. There are also two data overlay "run" segments and one code overlay "run" segment which is shared between the overlays in the internal memory of the DSP.

The data overlay concept is demonstrated in both an assembly version and a C version. Figures 3 and 4 show the project files for both the assembly and C implementations. Both projects consist of the linker description file, DSP system initialization files, the main routine, overlay, data files, macros/functions, three overlay variable declaration files, the data overlay manager and the data overlay test routine.

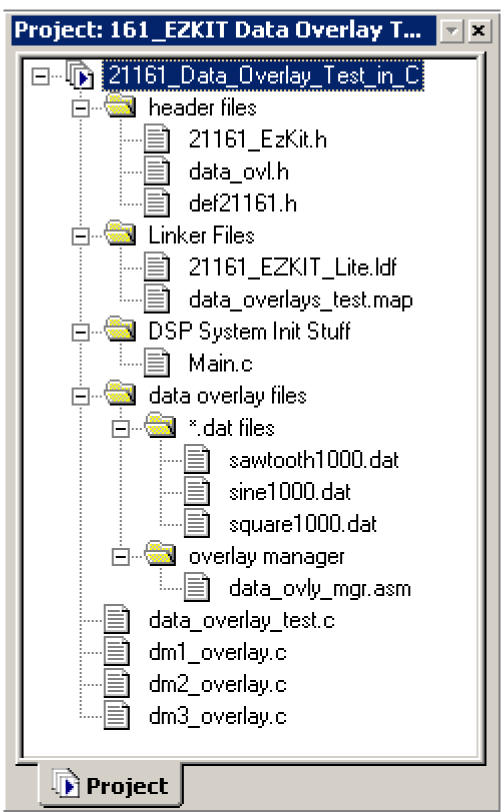


FIGURE 5. Project Files for C Language Data Overlay Example

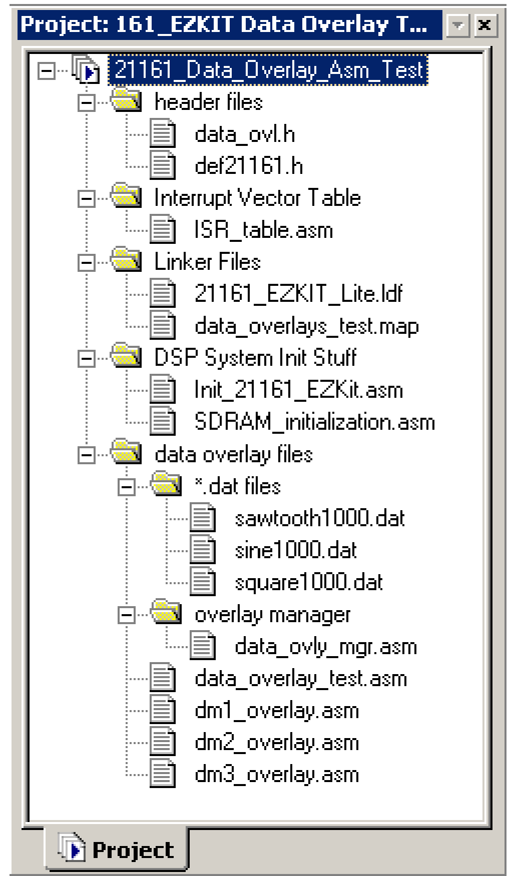


FIGURE 4. Project Files for Assembly Language Data Overlay Example

In these VisualDSP ++ project files, we implement three data overlay segments which share the same internal memory run space. The example defines six 1K buffers **x[], y[], m[], n[], p[], q[]** which will share a 2 k segment of internal memory run space.

In the ADSP-21161's 1 M-bit internal memory, we would allocate at least 2K words of internal memory in BLOCK1 as the "run" time memory for the data by defining this memory segment in the MEMORY{} section of the LDF file as follows:

```
rundmda  { TYPE(DM RAM) START(0x00051000)
END(0x00051fff) WIDTH(32) }
```

Next we want to define our "live" space in SDRAM mapped to bank0, where our data overlays will be stored. This is also defined in the MEMORY {} section of the LDF file:

```
ovlydmda  { TYPE(DM RAM) START(0x00200000)
END(0x0020FFFF) WIDTH(32) }
```

Each buffer within the two data overlay files are defined with a length of 1000 words, and each element contains 32 bit words. These are shown in the two source listings below. Note that Buffers **x[]** and **m[]** and **q[]** contain some initialized values.

LISTING 1: Data Overlay Declarations for ASM Example

dm1_overlay.asm

```
.SEGMENT/DM     run_dmda;
.GLOBAL x, y;

.VAR x[1000]="sine1000.dat";
.VAR y[1000];

.ENDSEG;
```

dm2_overlay.asm

```
.SEGMENT/DM     run_dmda;
.GLOBAL m, n;

.VAR m[1000]="sawtooth1000.dat";
.VAR n[1000];

.ENDSEG;
```

dm3_overlay.asm

```
.SEGMENT/DM     run_dmda;
.GLOBAL p, q;

.VAR p[1000]="square1000.dat";
.VAR q[1000];

.ENDSEG;
```

LISTING 2: Data Overlay Declarations for the C Example

dm1_overlay.c

```
/* Data Overlay 1 */
float sinetbl[1000] =
{
   #include "sine1000.dat"
};

float y_out[1000];
```

dm2_overlay.c

```
/* Data Overlay 2 */
float sawtbl[1000] =
{
   #include "sawtooth1000.dat"
};

float z_out[1000];
```

dm3_overlay.c

```
/* Data Overlay 3 */
float squartbl[1000] =
{
   #include "square1000.dat"
};

float q_out[1000];
```

The objective in the ADSP-21161 data overlay C and assembly tests shown here is to load in the data buffers **x[]** and **y[]** inside to the internal run space, copy the data from buffer **x[]** to buffer **y[]**, then store the current contents of both buffers back to external SDRAM memory. Once this process is over the run time memory can be freed. Thus, this same process is then repeated for buffers **m[]** and **n[]**, where buffers **m[]** and **n[]** are loaded to the same run space previously used by the **x[]** and **y[]** buffers. Buffer **m[]** is copied to buffer **n[]** and both buffers are stored back to SDRAM for later use. And again, the same process is repeated a third time for buffers **p[]** and **q[]**. This process is repeated over and over again, while the LED blink status gives an indication which data overlay load/store is executing.

Data Overlays Load/Store Functions:

As the VisualDSP++ tool only fully support code overlays, we are still able take advantage of this built in support to develop a scheme to handle data overlays. This can be accomplished by making a few modifications to the ADI-supplied instruction overlay manager which is provided in EE-66. It does, however, require a certain amount of house keeping on the programmer's part. Basically, this scheme requires the programmer to initiate the loading of an overlay containing the data buffers to be brought into internal memory from external SDRAM. This can be done by defining and calling an assembly macro called **Load_Data_Overlay( )**. For the C example, we take a similar approach but instead of implementing a macro, we can develop a small C function to initiate the data overlay process. The parameter the programmer must pass is the ID of the overlay to be loaded. This can easily be obtained from the map file.

To write the overlay back to external SDRAM memory with the changes, the programmer then calls a second macro or function called **Save_Data_Overlay( )**. Again, this macro or function requires the overlay ID as an input (this also is obtained from symbol map file).

LISTING 3. data_ovl.h for ASM Data Overlay Example

```
#define Load_Data_Overlay(SYMBOL_OVERLAYID)
               r0 = 1;\
               call _OverlayManager (DB);\
               dm(DataTransfer) = r0;\
               R0 = SYMBOL_OVERLAYID

#define Store_Data_Overlay(SYMBOL_OVERLAYID)
               r0 = 1;\
               dm(Internal2exttransfer)=r0;\
               call _OverlayManager (DB);\
               dm(DataTransfer) = r0;\
               R0 = SYMBOL_OVERLAYID
```

LISTING 4. data_ovl.h for C Data Overlay Example

```
extern int SYMBOL_OVERLAYID;

void load_data_overlay (int SYMBOL_OVERLAYID)
{
       DataTransfer = 1;
       asm volatile("R0 = R4;");
       OverlayManager();
}

void store_data_overlay(int SYMBOL_OVERLAYID)
{
       DataTransfer = 1;
       Internal2exttransfer = 1;
       asm volatile("R0 = R4;");
       OverlayManager();
}
```

Thus, these two macros (assembly version) or functions (C version) simply call the overlay manager with the proper parameters set. This is the only difference between data and code overlays (aside from the overlay manager DMA setup to initiate data vs. instruction transfers) as far as the linker is concerned. The macros take place of the PLIT table since a PLIT entry is not created for data overlays.

The overlay support provided is built in to the linker, and is partially designed by the user in the linker description file (LDF). To set up a data overlay, the user specifies which data overlays share runt time memory and which memory segments establish the live and run data spaces. The following LDF file fragments shows how we define three data overlays in the LDF. The first example is the LDF file portion contained in the assembly project. Listing 1 shows how to place the data overlay variables defined in the three overlay asm files to reside in the run space – *run_dmda.*

LISTING 3: Overlay Input/Output Sections for assembly example

```
no_dmda
{
    OVERLAY_INPUT
       {
        ALGORITHM(ALL_FIT)
        OVERLAY_OUTPUT(dm1_ovly.ovl)
        INPUT_SECTIONS(dm1_overlay.doj(run_dmda))
```

```
    }>ovlydmda

  OVERLAY_INPUT
    {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(dm2_ovly.ovl)
    INPUT_SECTIONS(dm2_overlay.doj(run_dmda))
    }>ovlydmda

  OVERLAY_INPUT
    {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(dm3_ovly.ovl)
    INPUT_SECTIONS(dm3_overlay.doj(run_dmda))
    }>ovlydmda
} >run_dmda
```

The LDF file data overlay declaration shown above demonstrates how to configure two data overlays to share a common run time data memory space. The first overlay, *dm1_ovly*, contains buffers **x[]** and **y[]** and lives somewhere in the external memory live segment *ovlydmda*. The second data overlay, *dm2_ovly*, contains buffers **m[]** and **n[]** and also lives in the external memory segment *ovlydmda*. The common run time data locations shared by overlays *dm1_ovly* and *dm2_ovly* is the segment *run_dmda*. The third data overlay, *dm3_ovly*, contains buffers **p[]** and **q[]** and also lives in the external memory segment *ovlydmda*. The common run time data locations shared by overlays *dm1_ovly*, *dm2_ovly*, and *dm3_ovly* is the segment *run_dmda*. The dummy output section *no_dmda* is a placeholder to keep the linker happy.

To set up our overlays in C for the floating point variables defined in Listing 2, we set up a similar redirection of the overlay input/output files, except we tell the linker to scan all *seg_dmda* sections in the 3 overlay object files and create separate overlay output files with the information pulled from each object file. However, the variables will still be directed to reside in the "run" space segment *run_dmda.*

LISTING 4: Overlay Input/Output Sections for C example

```
no_dmda
{
  OVERLAY_INPUT
    {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(dm1_ovly.ovl)
    INPUT_SECTIONS(dm1_overlay.doj(seg_dmda))
    }>ovlydmda

  OVERLAY_INPUT
    {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(dm2_ovly.ovl)
    INPUT_SECTIONS(dm2_overlay.doj(seg_dmda))
    }>ovlydmda

  OVERLAY_INPUT
    {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(dm3_ovly.ovl)
    INPUT_SECTIONS(dm3_overlay.doj(seg_dmda))
    }>ovlydmda
} >run_dmda
```

The LDF provides the linker with direction on how to configure overlays as well as the information necessary for the data overlay manager routine to load in the data overlays. The data overlay information resolved by the linker to initiate DMA transfers in our example includes the following constants:

Data Overlay 1
_ov_word_run_size_1
_ov_word_live_size_1
_ov_startaddress_1
_ov_runtimestartaddress_1

Data Overlay 2
_ov_word_run_size_2
_ov_word_live_size_2
_ov_startaddress_2
_ov_runtimestartaddress_2

Data Overlay 3
_ov_word_run_size_3
_ov_word_live_size_3
_ov_startaddress_3
_ov_runtimestartaddress_3

These linker-generated constants are stored specified locations in user-defined buffers for the overlay manager to access and initiate DMA transfers. In this overlay example the data is stored as follows:

```
liveAddresses[3] =        ← (EIEPx)
_ov_startaddress_1,
_ov_startaddress_2,
_ov_startaddress_3;

runAddresses[3]  =        ← (IIEPx)
_ov_runtimestartaddress_1,
_ov_runtimestartaddress_2,
_ov_runtimestartaddress_3;

runWordSize[3] =          ← (CEPx)
_ov_word_size_run_1,
_ov_word_size_run_2,
_ov_word_size_run_3;

liveWordSize[3] =         ← (ECEPx)
_ov_word_size_live_1,
_ov_word_size_live_2,
_ov_word_size_live_3;
```

The linker will replace these constants with actual internal memory run space addresses and sizes and external memory live space addresses and sizes, which are used to program the IIEPx, IMEPx, CEPx, EIEPx, EMEPx, ECEPx DMA parameter registers to initiate the loading or storing of a data overlay.

The Data Overlay Manager:

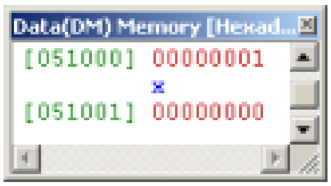
The data overlay manager is a user-defined routine that is responsible for loading a referenced overlay data variable or buffer into the internal memory "run" time space. The central task of the data overlay manager is to initiate an External Port DMA operation using one of the external port buffer DMA channels (DMA channels 10 to 13). This DMA operation is managed with the aid of the linker generated constants, in which the internal start address (IIEPx), external start address (EIEPx), modify (IMEPx/EMEPx) and DMA count (CEPx/ECEPx) are programmed from the linker-generated overlay constants. The main objective of the data overlay manager is to set up a DMA transfer to/from the "live" space to/from the "run" space. Thus, the main objective of the data overlay manager is to:

- Perform a context save/restore of all registers used by the overlay manager via a software stack.
- Identify the desired data overlay module by getting the ID# of the overlay.
- Sets up an EPBx DMA transfer. Before enabling the DMA transfer, the overlay manager determines if the data overlay module associated with the ID# will require a load from external memory, or a store to external memory. Thus, the DMA transfer set up in the DMACx register will either write the contents of internal memory to external memory, or read the contents from external memory to internal memory.
- Assign the appropriate live/run addresses and DMA count size to the I/O processor, so that the data overlay is properly loaded from or stored to external memory.

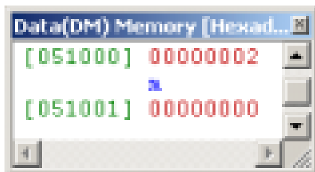
The overlay manager shown here is written such that the core processor sits at an "IDLE" instruction while the transfer occurs. However, there is additional instructional support which allows the user to initiate an overlay load/store while the DSP core executes another section of unrelated code. The user is referred to EE-66 for more information on how to implement more advanced predictive overlay managers.

Stored Live Space Overlay ID Identifiers for Visual DSP Debugger Symbol Support

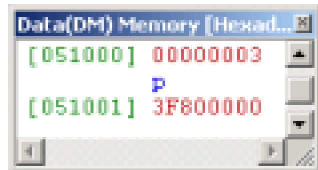
In order to support debugging of overlay variables and buffers, the latest VisualDSP ++ 2.0 linker overlay support also includes the storage of the overlay ID in the live and run spaces to assist with symbol support in the debugger. In our assembly data overlay example, since we define a data overlay run memory section to beginning at address 0x51000 to be run space for overlays, that address will always hold the Overlay ID, so then our data overlay actually begins at address 0x50001. And the image in live space will always hold the overlay id. For example, the following memory window of 0x51000 shows the Overlay ID identifier for the data overlay run space, followed by the actual data buffers:



Overlay 1 ID @ 0x51000



Overlay 2 ID @ 0x51000



Overlay 3 ID @ 0x51000

Overlay 1 ID stored in live space at address 0x0x200000

Overlay 2 ID stored in live space at address 0x0x2009C6

Overlay 3 ID stored in live space at address 0x0x20138C

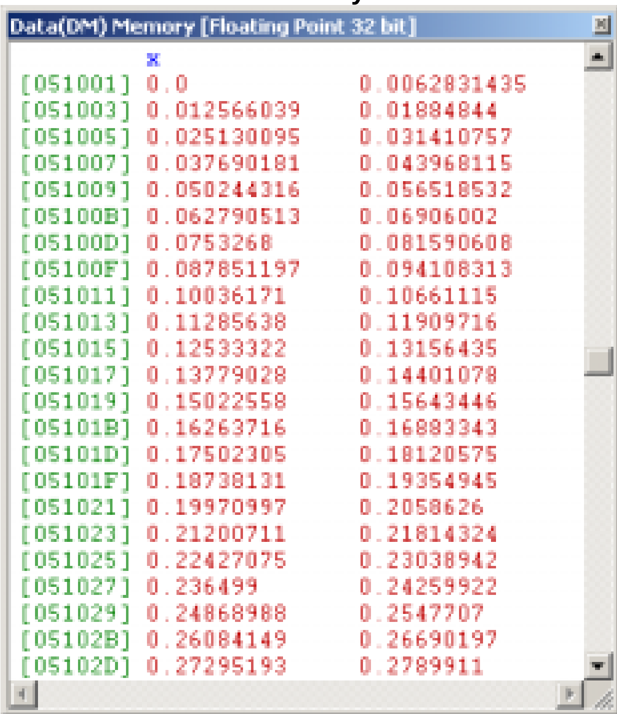
Since address 0x51000 contains the overlay ID, this allows the debugger to display the correct symbol names for the data overlay variables. If you wanted for some reason to avoid having the overlay ID in run memory you have to increment the live start address provided by the linker by 1… but doing do would defeat the symbol support. Since we in most cases reference all the code or data symbolically, this step of incrementing the live start address is not necessary.

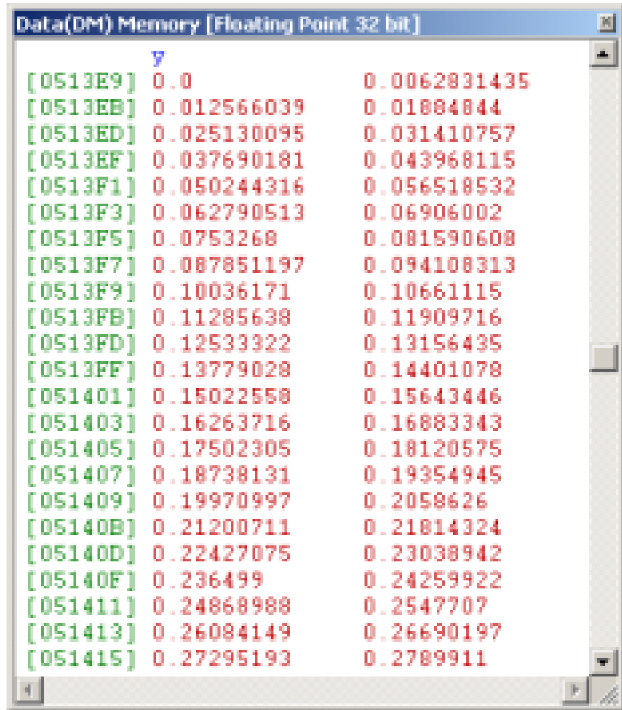
In our assembly example, the following addresses show where all of the variables and overlay symbol IDs reside in internal run space and external live space:

Address	What resides here:
0x200000	Overlay 1 Symbol ID live space storage
0x2009C6	Overlay 2 Symbol ID live space storage
0x20138C	Overlay 3 Symbol ID live space storage
0x51000	Symbol ID run space location for Data Overlays 1, 2 and 3
0x51001	Run space start address for data overlay variables x[ ], m[ ] and p[ ]
0x513E9	Run space start address for data overlay variables y[ ], n[ ] and q[ ]
0x200001	Live space start address for x[ ]
0x2003E9	Live space start address for y[ ]
0x2009c7	Live space start address for m[ ]
0x200DAF	Live space start address for n[ ]
0x20138D	Live space start address for p[ ]
0x20176E	Live space start address for q[ ]

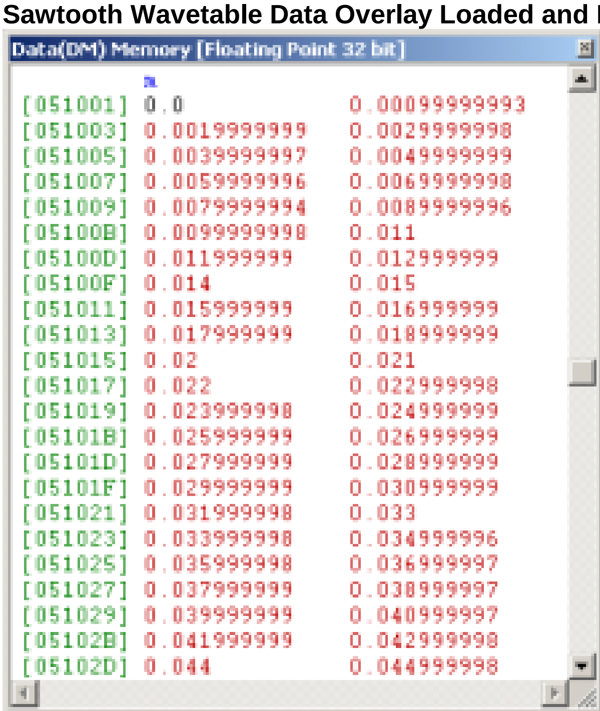
21161 EZ-KIT USB Debugger Session Results of the 3 Data Overlay Buffer Load/Copy operation in internal "RUN" space [Assembly Data Overlay Example]

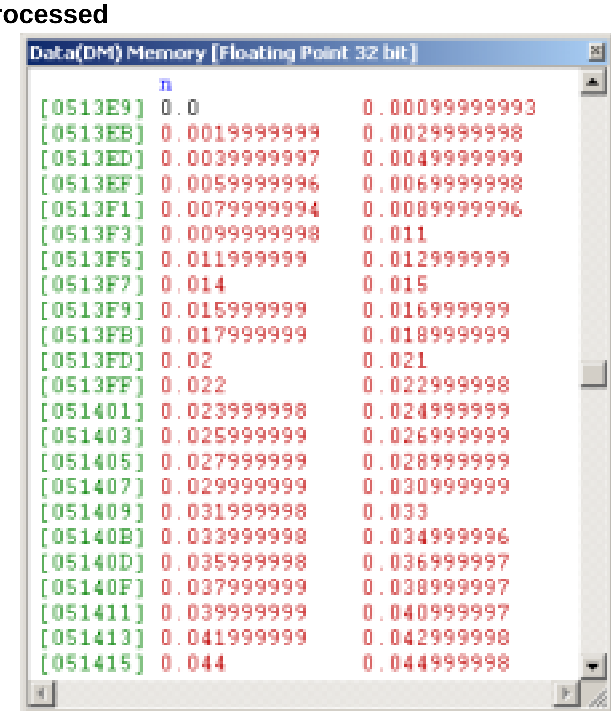
Sine Wavetable Data Overlay Loaded and Processed



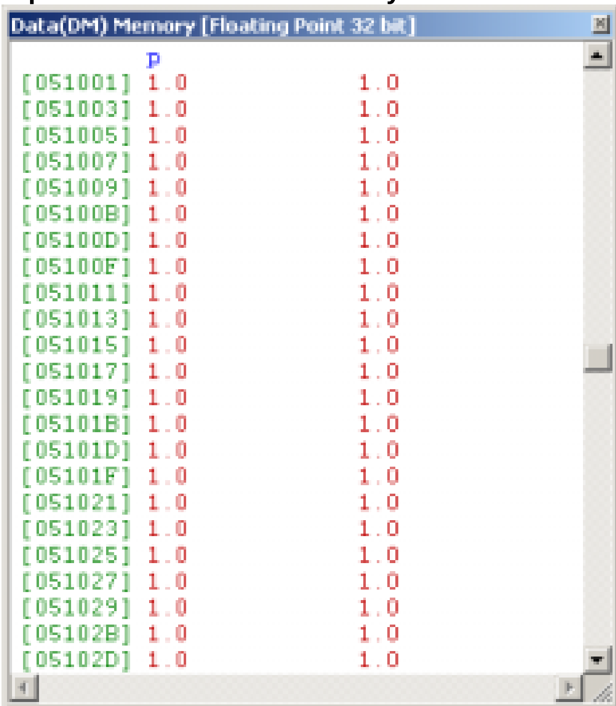


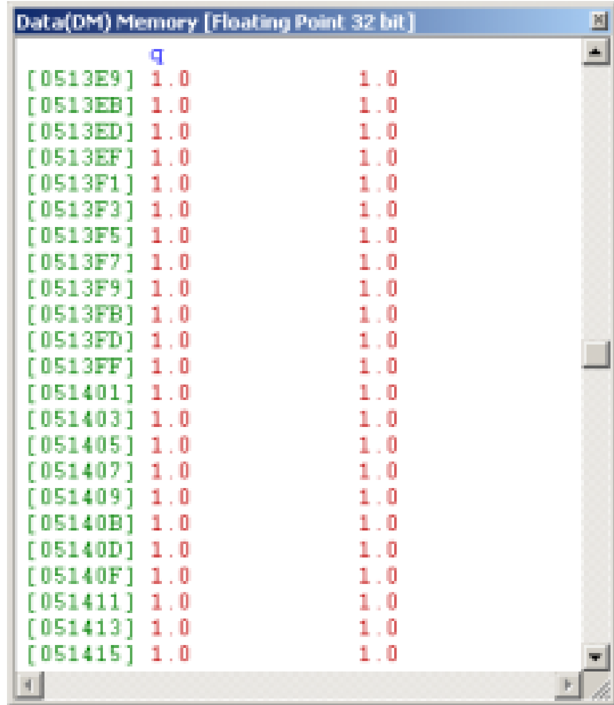
Sawtooth Wavetable Data Overlay Loaded and Processed





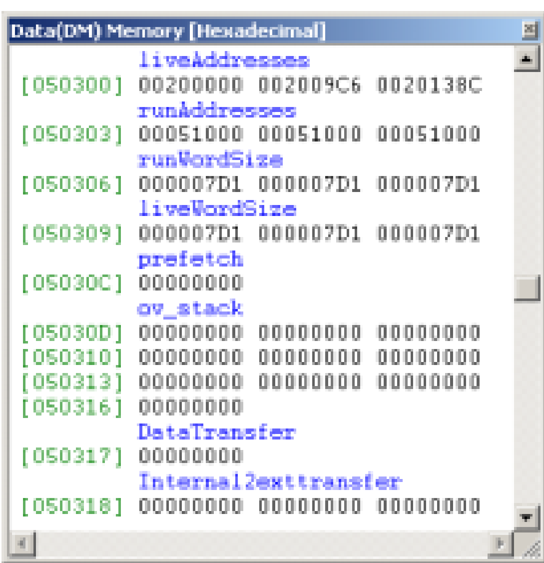
Square Wavetable Data Overlay Loaded and Processed



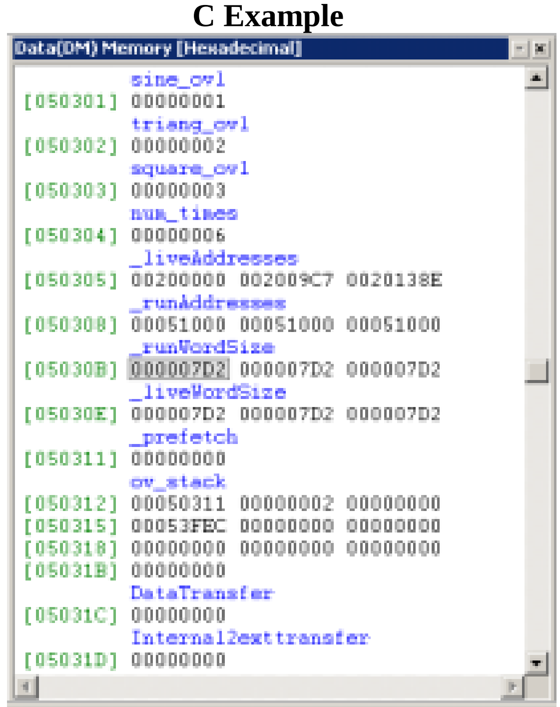


Linker-Generated Data Overlay "Live" and "Run" Address and Size Buffer Constants for the EZ-KIT Data Overlay Example

Assembly Example



C Example



Listing 5. Assembly Data Overlay Test: data_overlay_test.asm

```
///////////////////////////////////////////////////////////////////////////
//                                                                       //
//   In this example data overlay code there are four buffers x,y,m,n.   //
//   Each are of length 1000.(32 bit words)                              //
//                                                                       //
//   Buffers x and m contain some initialised values.                    //
//                                                                       //
//   Our objective is to copy x to y, m to n and p to q.                 //
//                                                                       //
//   To save internal memory, we allocate the same 2k internal memory as //
//   the run time data memory.                                           //
//                                                                       //
//   First we bring in x and y inside copy x to y and copy back          //
//   the current values to the external SDRAM memory.                    //
//   Once this process is over the run time memory can be freed.         //
//                                                                       //
//   The same process same block copy process is also repeated for       //
//   buffers m and n, and then again for p and q.                        //
//                                                                       //
///////////////////////////////////////////////////////////////////////////

#include "data_ovl.h"

#define xy_ovl 1
#define mn_ovl 2
#define pq_ovl 3

.GLOBAL data_overlay_test;
.EXTERN x, y, m, n, p, q;
.EXTERN Blink_LEDs_Twice, Blink_LEDs_Thrice, Blink_LEDs_4times;
.EXTERN prefetch;
.EXTERN DataTransfer;
.EXTERN Internal2exttransfer;

.SEGMENT/PM             pm_code;

data_overlay_test:
      Load_Data_Overlay(xy_ovl);        /* load  x & y data buffers from overlay live space
                                           in SDRAM */
      l0 = 0;
      i0 = x;
      l1 = 0;
      i1 = y;

      lcntr=1000, do copy_m2n until lce;
               r0 = dm(i0,1);
copy_x2y:      dm(i1,1) = r0;

      Store_Data_Overlay(xy_ovl);       /* store x & y data buffers to overlay live space in
                                           SDRAM */
      call Blink_LEDs_Twice;            /* Data Overlay x & y load/store LED confirm */

      Load_Data_Overlay(mn_ovl);        /* load  m & n data buffers from overlay live space
                                           in SDRAM */

      l0 = 0;
      i0 = m;
      l1 = 0;
      i1 = n;

      lcntr =1000, do copy_m2n until lce;
               r0 = dm(i0,1);
```

```
copy_m2n:      dm(i1,1) = r0;

      Store_Data_Overlay(mn_ovl);       /* store m & n data buffers to overlay live space in
                                           SDRAM */
      call Blink_LEDs_Thrice;           /* Data Overlay m & n load/store LED confirm */

      Load_Data_Overlay(pq_ovl);        /* load  p & q data buffers from overlay live space
                                           in SDRAM */

      l0 = 0;
      i0 = p;
      l1 = 0;
      i1 = q;

      lcntr =1000, do copy_p2q until lce;
               r0 = dm(i0,1);
copy_p2q:      dm(i1,1) = r0;

      Store_Data_Overlay(pq_ovl);       /* store p & q data buffers to overlay live space in
                                           SDRAM */

      call Blink_LEDs_4times;           /* Data Overlay p & q load/store LED confirm */

      RTS;

.ENDSEG;
```

Listing 6. C Data Overlay Code Test: data_overlay_test.c

```
///////////////////////////////////////////////////////////////////////////
//                                                                       //
//   In this example data overlay code there are six buffers, which      //
//      consist of 3 input wavetables: sine, triangle and square, and 3   //
//   output buffers used as output sources for each input wavetable.     //
//   Each buffer is of length 1000.(32 bit words)                        //
//                                                                       //
//   Buffers sinetbl[], sawtbl[] and squartbl[] contain initialized      //
//   floating point data                                                 //
//                                                                       //
//   Our objective is to copy sinetbl[] to y_out[],                      //
//   sawtbl[] to z_out[] and squartbl[] to q_out[],                      //
//                                                                       //
//   To save internal memory, we allocate the same 2k internal memory as //
//   the run time data memory.                                           //
//                                                                       //
//   First we bring in sinetbl[] and y_out[] inside copy sinetbl[]       //
//   to y_out[] and copy back the current values to the external         //
//   SDRAM memory. Once this process is over the run time memory         //
//   can be freed.                                                       //
//                                                                       //
//   The same process same block copy process is also repeated for       //
//   buffers sawtbl[], squartbl[], z_out[] and q_out[].                  //
//                                                                       //
//   Analog Devices, DSP Applications                                    //
//   Last Modified: 01/22/02                                             //
//                                                                       //
///////////////////////////////////////////////////////////////////////////

#include "21161_EZKit.h"
#include "data_ovl.h.h "

#define sine_ovl    1
#define triang_ovl  2
```

```
#define square_ovl  3

int num_times;
int OverlayID;

void   data_overlay_test()
{
       int i, j, k;

       OverlayID = sine_ovl;
       load_data_overlay(OverlayID);     /* load sine wave data buffers from overlay live
                                            space in SDRAM */
       for ( i = 0; i < 1000; ++i)
       {
              y_out[i] = sinetbl[i];
       }
       store_data_overlay(OverlayID);    /* store sine wave data buffers to overlay live space
       num_times = 4;                       in SDRAM */
       Blink_LEDs_Slow(num_times);       /* sine wave data overlay load/store LED
                                            confirm */


       OverlayID = triang_ovl;
       load_data_overlay(OverlayID);     /* load  m & n data buffers from overlay live
                                            space in SDRAM */
       for ( j = 1; j <= 1000; ++j)
       {
              z_out[j] = sawtbl[j];
       }
       store_data_overlay(OverlayID);    /* store triangle wave data buffers to overlay live
       num_times = 6;                       space in SDRAM */
       Blink_LEDs_Medium(num_times);     /* sawtooth wave data overlay load/store LED
                                            confirm */


       OverlayID = square_ovl;
       load_data_overlay(OverlayID);     /* load  m & n data buffers from overlay live space
                                            in SDRAM */
       for ( k = 1; k <= 1000; ++k)
       {
              q_out[k] = squartbl[k];
       }
       store_data_overlay(OverlayID);    /* store square wave data buffers to overlay live
       num_times = 8;                       space in SDRAM */
       Blink_LEDs_Fast(num_times);       /* Square Wave Data Overlay load/store LED confirm */
}
```

Listing 7. Overlay Manager Source File: data_ovly_mgr.asm

```
/////////////////////////////////////////////////////////////////////////////////
//                                                                             //
//  This code/data overlay manager sets up data overlays to/from SDRAM         //
//  mapped to Bank 0 on the 21161 EZ-KIT Lite.  Since the SDRAM data accesses  //
//  are 32-bits, a generic 32-to-32 non-packed is used to transfer of data     //
//  to/from interal/external memory.  Therefore, no PACKING{} command is       //
//  required in the overlay declaration section of the linker.                 //
//                                                                             //
//  This example could be modified to set up data overlays to/from Bank 1 Flash //
//  memory, using 8to32 or 8to48 DMA packing modes and the appropriate flash   //
//  write commands.  The Overlay Declaration in the LDF file would require     //
//  the necessary PACKING{} command to prestore data to 8-bit space.           //
//  Refer to EE-150 for information on how to write to the EZ-KIT Flash Memory. //
//                                                                             //
/////////////////////////////////////////////////////////////////////////////////

#include "def21161.h"

.SEGMENT/DM   dm_data;

.EXTERN       _ov_word_run_size_1;
.EXTERN       _ov_word_run_size_2;
.EXTERN       _ov_word_run_size_3;

.EXTERN       _ov_word_live_size_1;
.EXTERN       _ov_word_live_size_2;
.EXTERN       _ov_word_live_size_3;

.EXTERN       _ov_startaddress_1;
.EXTERN       _ov_startaddress_2;
.EXTERN       _ov_startaddress_3;

.EXTERN       _ov_runtimestartaddress_1;
.EXTERN       _ov_runtimestartaddress_2;
.EXTERN       _ov_runtimestartaddress_3;

.VAR          liveAddresses[2] =  _ov_startaddress_1,
                                  _ov_startaddress_2,
                                  _ov_startaddress_3;

.VAR          runAddresses[2]  =  _ov_runtimestartaddress_1,
                                  _ov_runtimestartaddress_2,
                                  _ov_runtimestartaddress_3;

.VAR          runWordSize[4] =    _ov_word_size_run_1,
                                  _ov_word_size_run_2,
                                  _ov_word_size_run_3;

.VAR          liveWordSize[4] =   _ov_word_size_live_1,
                                  _ov_word_size_live_2,
                                  _ov_word_size_live_3;

.VAR          prefetch=0;
.GLOBAL       prefetch;

.VAR          ov_stack[10];

.VAR          DataTransfer;
.VAR          Internal2exttransfer;

.GLOBAL       DataTransfer;
.GLOBAL       Internal2exttransfer;

.ENDSEG;
```

```
/* ---------------------------------------------------------------------------- */

.SEGMENT/PM   pm_code;

.GLOBAL _OverlayManager;

_OverlayManager:
      /* R0 register is the overlay ID 1,2,... */
      /* R1 register is the SYMBOL REFERENCE Address */
      dm(ov_stack) = i8;         /* save user registers temporarily to overlay stack */
      dm(ov_stack+1) = m8;
      dm(ov_stack+2) = l8;
      dm(ov_stack+3) = r2;

      m8 = R0;                   /* get user-selected overlay ID from table */

skip_if_overlay_loaded:
      i8 = runAddresses - 1;            /* pointer to previous address before runAddresses
                                           buffer */
      r0 = dm(Internal2exttransfer);    /* check if we are loading or storing data ovelay
                                           data */
      r0 = pass r0;                     /* if = 1, we store to ext mem, otherwise, = 0 loads
                                           from ext mem */

      if NE jump continue_ovly_load_process;

      i8 = runAddresses - 1;     /* pointer to previous address before runAddresses buffer */
      i8 = pm (m8,i8);           /* Can we skip overlay transfer if data already resident? */
      px = pm(0,i8);
      r2 = px1;                  /* Now check if already loaded */

      r0 = r0 - r2;              /* If equal to zero, data still there */
      if EQ jump TransferOver;   /* we can skip DMA setup for load to run space */

continue_ovly_load_process:
      i8 = runAddresses - 1;     /* pointer to previous address before runAddresses buffer */

      dm(ov_stack+4) = i0;       /* save more user registers temporarily to overlay stack */
      dm(ov_stack+5) = m0;
      dm(ov_stack+6) = l0;
      dm(ov_stack+7) = ustat1;

      l8 = 0;                    /* disable DAGs modulus logic, no circular buffering */
      l0 = 0;

      m0 = m8;                   /* premodify values are equal to the overlay symbol ID # */

      i0 = liveAddresses - 1;    /* pointer to previous address before live Addresses
                                    buffer */

      r0 = 0;
      dm(DMAC10) = r0;           /* initially disable and clear out DMA channel 10 */

      r0 = dm(m0,i0);      /* External Live Address from liveAddresses buffer */
      dm(EIEP0) = r0;      /* write Live Address to EPB0 external index address register */

      r0 = pm(m8,i8);      /* Internal Run Address from runAddresses buffer */
      dm(IIEP0) = r0;      /* write Run Address to EPB0 internal index address register */

      r0 = 1;              /* DMA int/ext modifys = 1 */
      dm(EMEP0) = r0;      /* setup EPB0 external and internal index modify registers */
      dm(IMEP0) = r0;

      i8 = liveWordSize - 1;     /* pointer to buffer for DMA external count size */
```

```
      i0 = runWordSize - 1;      /* pointer to buffer get DMA internal count size */

      r0 = dm(m0,i0);
      dm(CEP0) = r0;             /* set number of "Run" internal words to transfer */

      r0 = pm(m8,i8);
      pm(ECEP0) = r0;            /* set number of "Live" external words to transfer */

Program_or_Data_Transfer:
      r0 = dm(DataTransfer);     /* check for data overlay task */
      r0 = pass r0;              /* if DataTransfer = 1, we do a data overlay DMA */
      if eq jump LoadOpcodes;

      r1 = pcstk;                /* Jump back to location where call was made */
      pop pcstk;                 /* This will help us get back using the indirect delayed
                                    jump instruction */

LoadData:
      r0 = dm(Internal2exttransfer);    /* check if we are loading or storing data
                                           overlay data */
      r0 = pass r0;                     /* if = 1, we store to ext mem, otherwise, = 0
                                           loads from ext mem */
      if ne jump StoreData;

      /* master mode DMA, 32-32 no packing, data xfer, ext-to-int, DMA enable */
      r0 =  MASTER | PMODE4;
      dm(DMAC10) = r0;
      jump Start_DMA_Sequence;

StoreData:
      /* master mode DMA, 32-32 no packing, data xfer, int-to-ext, DMA enable */
      jump Start_DMA_Sequence (db);
      r0 =  MASTER | PMODE4 | TRAN;
      dm(DMAC10) = r0;

LoadOpcodes:
      /* NOTE: This data overlay example never calls the instruction overlay support.  This
      code is not executed in this example, but is included in case the programmer wants to
      do both code and data overlays */
      /* master mode DMA, 32-to-48 packing, instructions xfer, ext-to-int, DMA enable */
      r0 =  MASTER | PMODE3 | DTYPE | TRAN;
      dm(DMAC10) = r0;

Start_DMA_Sequence:
      IRPTL = 0x00000000;
      bit set mode1 IRPTEN;
      bit set imask EP0I;

      /* now turn on DMA channel 10 for code/data overlay loading or storing */
      ustat1 = dm(DMAC10);
      bit set ustat1 DEN;                       /* DMA enable bit on */
      dm(DMAC10) = ustat1;

      r0 = 0;                           /* Used to reset key variables */
      dm(DataTransfer) = r0;            /* Reset to 0 */
      dm(Internal2exttransfer) = r0;    /* Reset to 0 */

      ustat1 = dm(ov_stack+7);          /* restore overlay modified registers */
      l0 = dm(ov_stack+6);
      m0 = dm(ov_stack+5);
      i0 = dm(ov_stack+4);

      i8 = prefetch;                    /* determines if we wish to sit at IDLE or return */
      r2 = pm(0,i8);                    /* to do some other core accesses while load
                                           completes */
```

```
      r2 = pass r2;

      if NE jump dont_wait_for_DMA_IRQ;

wait_for_EPB0_DMA_end:
      idle;                                /* Overlay Load/Store complete */
      bit clr imask EP0I;                  /* temporarily disable EPB0 interrupts */

      jump TransferOver;

dont_wait_for_DMA_IRQ:
      r2 = 0;
      dm(prefetch) = r2;
      r2 = dm(ov_stack+3);                 /* restore the rest of overlay-modified variables */
      l8 = dm(ov_stack+2);
      r0 = dm(ov_stack);
      i8 = r0;
      m8 = r1;

      rts;                                 /* return if we don't need to wait for EPB0 DMA
                                              Interrupt */

TransferOver:
      r2 = dm(ov_stack+3);                 /* restore the rest of overlay-modified variables */
      l8 = dm(ov_stack+2);
      i8 = r1;
      m8 = 0;
      r1 = dm(ov_stack+1);
      r0 = dm(ov_stack);

      flush cache;                         /* this instruction not required for data overlay
                                              transfers */

      jump (m8,i8) (db);                   /* indirect jump and return from overlay manager */
      i8 = r0;                             /* remember we already popped the PC Stack earlier */
      m8 = r1;

.ENDSEG;
```