Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at dsp.support@analog.com and at dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

Code Overlays on the Third Generation SHARC® Family of Processors

Contributed by Brian M. and Divya S.

Rev 2 – April 11, 2005

Introduction

The overlay facilities in the VisualDSP++® development tools provide an efficient method of managing DSP internal memory. Similar to previous generations of SHARC® family processors, you can use code overlays to exploit the relatively small internal memory blocks (between 1 and 3 Mbits total) found in third generation SHARC processors.

This article, which extends upon previous EE-Notes, focuses on the overlay management changes that are necessary to support third-generation SHARC DSPs. Before continuing with this article, be sure to be familiar with the concepts explained in these earlier documents:

- *Using Memory Overlays (EE-66)* [1] This document introduces the use of overlays on SHARC processors. It includes the concepts and details of using the code overlays from external RAM.
- *Using Code Overlays from ROM on the ADSP-21161N EZ-KIT Lite (EE-180)* [2] This document extends the concepts presented in EE-66 to include the use of code overlays from cheaper ROM parts.

For the third generation SHARC processors (ADSP-2126x/2136x), the I/O Processor can no longer access 48-bit addresses. This means that overlays must be copied to the equivalent 32-bit location. Since the beginning or end of a 48-bit word can lie in the middle of a 32-bit word, special measures are needed to ensure proper transfers.

This document demonstrates overlay managers with three projects that demonstrate a method to handle this new behavior:

- A simple overlay manager that uses external SRAM to store the overlays
- A project that stores overlays in a parallel EEPROM or Flash
- A project that stores overlays in an SPI Flash

A Quick Introduction to Overlays

When using the overlay facilities provided by VisualDSP++, the linker changes a call to a function located in an overlay section into a call to an overlay manager. The overlay manager then transfers the desired code from an external memory location (the live address) to an area in internal memory set aside for overlays (the run address) and jumps to the applicable location to execute the code. (See EE-66 for a thorough discussion of overlays on SHARC processors.)

When overlays are stored in external RAM, the overlay live space is initialized at boot time in the same manner as internal memory (by the boot loader kernel). However, when ROM or Flash devices are used instead of RAM, it is easier to use the loader to generate a boot image to burn into the memory (see EE-180). When this method is used, an initialization routine is run to determine where the overlays live, and how many loader sections comprise the entire overlay.

This document discusses how to extend the previous examples to deal with issues that are specific to third generation SHARC processors.

Basic Overlay Example

The overlay sections provided in the attached projects are intended for use on the EZ-KIT Lite evaluation system. Each overlay has one function that sets a unique subset of the 8 LEDs on the board to indicate the overlay that has just been executed.

Overlays from External RAM

The external memory support provided in VisualDSP++ for the third generation SHARC processors is different from the support provided for the ADSP-21161 and ADSP-21065L DSPs discussed in EE-180 and EE-66, respectively. When mapping the overlays to external memory, use the `PACKING()` command to place the overlay data properly in external memory. Figures 1 and 2 show the necessary `PACKING()` command for 8- and 16-bit external SRAM modules, respectively.

```
PACKING(6 B0 B0 B0 B6 B0 B0 \
          B0 B0 B0 B5 B0 B0 \
          B0 B0 B0 B4 B0 B0 \
          B0 B0 B0 B3 B0 B0 \
          B0 B0 B0 B2 B0 B0 \
          B0 B0 B0 B1 B0 B0)
```

Figure 1. `PACKING()` command for 8-bit memories

```
PACKING(6 B0 B0 B5 B6 B0 B0 \
          B0 B0 B3 B4 B0 B0 \
          B0 B0 B1 B2 B0 B0 )
```

Figure 2. `PACKING()` command for 16-bit memories

These `PACKING()` commands must appear in the `OVERLAY_INPUT()` section of the LDF file to instruct the linker how to place the data in the executable file. (See *Using External Memory with Third Generation SHARC Processors and the Parallel Port (EE-220)* [3] for a discussion of external memory on ADSP-2126x family DSPs.)

Another change from previous SHARC DSPs is the lack of 48-bit DMA transfers on third generation SHARC processors. This change greatly affects the overlay manager. All transfers that use the Parallel Port or SPI are limited to 32-bit words; therefore, the 48-bit run address must be translated into an equivalent 32-bit address. This is accomplished by multiplying the 48-bit address by 1.5. Figure 3 shows the assembly code that translates a 48-bit address into its 32-bit equivalent.

```
r3=0x80100;  /* 48-bit address */
r2=3;   /* Use R2 as a multiplier */

/* Extract lower bits from R4 into R3
(bits that need to be translated) */
r3=fext r4 by 0:16;

/* Place remaining bits in R0 (these
bits indicate word space) */
r0=r4-r3;

/* Multiply lower bits by 3 */
r3=r3*r2 (ssi);

/* Divide the result by 2 */
r3=lshift r3 by -1;

/* Insert the new lower bits */
r0=r0 or fdep r3 by 0:16;
/* Finished! R0=0x80180 */
```

Figure 3. Translating a 48-bit address to the 32-bit address required by the overlay manager

It is also necessary to transform the *number* of 48-bit words (run word size) to the equivalent number of 32-bit words. For a DMA transaction to complete correctly, the DMA must be set up to transfer a whole number of 32-bit words. Therefore, when translating from 48-bit to 32-bit lengths, round the calculation up.

Each 48-bit word ends or begins in the middle of the equivalent 32-bit address. If the 48-bit run address is odd, the first 16-bits of the equivalent 32-bit address must be saved before fetching the overlay and restored after the transfer is complete, since only the final half of the 32-bit word pertains to the run address. Similarly, if the run space ends on an even 48-bit address, the final 16-bits of the equivalent 32-bit address must be saved and restored in the same manner. In the examples included with this document, the starting and ending run address are checked for each overlay, and problem addresses are stored into a 1-location stack and restored as described above.

When the run address is odd, the starting live address must be decreased by 1 or 2 locations, respectively, depending on whether the external memory is 8- or 16-bit. This aligns the location of the 48-bit run and live addresses, since the 32-bit run address that will be accessed must always lie on a 32-bit boundary.

Since third generation SHARC DSPs support 8- and 16-bit external memories, the provided overlay manager automatically determines the width of the external memory being used. This is accomplished by comparing the run word size and live word size.

Overlays from ROM/Flash

Using overlays from ROM/Flash is very similar to using them from RAM. Set up the `OVERLAY_INPUT()` section of the LDF file as if the overlays are to reside in external memory. However, map the overlays to a dummy live address in the external memory map (between `0x2000000` and `0x2FFFFFF`) that is not being occupied by a physical memory device. (Remember the actual live address is a location in the ROM – this dummy live address is used only as a search string to locate each overlay in the ROM because this info is not available at link-time.) It does not matter whether the memory is set up as 8- or 16-bit memory, as the loader generates the proper format from either memory type.

When using an image generated by the loader, tags are inserted before data sections to identify different sections in the boot stream. The dummy live addresses precede each overlay as it sits in the boot-ROM, along with a data-type tag and word count. Before calling an overlay section, the overlay manager must parse the boot stream, looking for these dummy live addresses, saving their locations in the ROM and number of boot-sections for the entire address range devoted to the overlay section. The overlay sections are tagged (as shown in Table 1) with a normal external memory type at an address in the overlay range specified in the LDF file.

Memory Type	Tag for Zero Init	Tag for Regular
8-bit	0x7	0x9
16-bit	0x8	0xA

Table 1. Loader tags for ADSP-2126x DSPs

As described in EE-180, the overlay init subroutine parses the boot stream to find the locations of the overlay sections, and the overlay section info subroutine determines how many loader sections comprise each overlay section. The provided code assumes that the overlays live in the same device used to boot the DSP. Therefore, the overlay init routine begins parsing the loader file at the first byte after the kernel (location `0x600`).

The overlay init routine supplied with these examples assumes that the live space for all of the overlays is contiguous. The routine looks for tags that reside between the live start address of the first overlay and the live end address of the final overlay. It automatically uses the addresses supplied by the linker, but must be updated to account for the number of overlays being used.

When using the overlay manager, the overlay init and overlay section info routines automatically provide a whole number of 32-bit words, and a valid live address. The same calculation to obtain the run address described above must be performed when running from ROM, as do the checks for odd and even 48-bit problems. However, if the overlay starts on an odd address, the beginning of each loader section in the overlay to be fetched will fall in the middle of a 32-bit word; therefore, apply the save and restore scheme for the first 16-bits of the word to each loader section that is fetched.

Finally, as in the loader kernel, the overlay manager automatically initializes the correct number of words when a zero init section is detected, rather than transferring the zeroes.

Parallel ROM/Flash Overlay Manager

The overlay manager provided for a parallel ROM/Flash uses the Parallel Port DMA registers to store the internal run address. However, the address saved in the IIPP register must be in 32-bit normal word space, so the highest bits of address are not stored in the register. The starting addresses to save and restore is taken from this register, and therefore, the upper bits must be masked in. This example assumes that all of the overlays run in Block 0 (addresses `0x80000-0x9FFFF`), that is, the mask is always assumed to be `0x80000`.

The `read_data_bytes()` routine used in the overlay manager and overlay init routine assumes that the device being used is byte-wide, since this is the format that must be used to boot the DSP.

SPI ROM/Flash Overlay Manager

The basic concepts used to store overlays in an SPI Flash are identical to storing them in a Parallel Flash as described above. However, a few simple changes are necessary to support the way the SPI flash is addressed.

As described in *Programming an ST M25P80 SPI Flash with ADSP-21262 SHARC DSPs (EE-231)* [4], an SPI flash must be programmed in a least significant bit first (LSBF) format to be bootable. However, the commands sent to an SPI Flash must be sent in most significant bit first format. Therefore, when accessing overlays stored in an SPI Flash, bit-reverse the read command and address that is sent to the Flash to read the overlay data stored in LSBF format. The `read_data_bytes()` routine used to access the SPI Flash is taken from the SPI Flash programmer provided in EE-231.

Since the SPI Flash requires an extra 32-bit transfer (rather than direct address lines), one extra 32-bit word (corresponding to the 1-byte read command and 24-bit address) is transferred into the run space for each access to the SPI Flash. When transferring the overlay sections, this word overwrites the word directly preceding the section to be written. The affected word is saved and restored in the same manner as the words that fall on an odd 48-bit boundary.

Conclusion

Although the concepts of code overlays are described in detail in EE-66 and EE-180, this document extends these concepts to the third generation SHARC processors. The main difference in using overlays on third generation SHARC processors systems from its lack of 48-bit DMA support, but this is easily worked around.

In addition to using overlays from external RAM and ROM or Flash devices via the Parallel Port (as has been described for previous members of the SHARC family), DSP overlays can be stored in an SPI Flash device. This document provides a framework for using overlays from SPI Flash, supporting third generation SHARC DSPs. This same framework can also be extended to support the ADSP-21161.

Appendix

ovly_mgr.asm

```
/*  The OVLY_MGR.ASM file is the overlay manager.  When a symbol   */
/*  residing in overlay is referenced, the overlay manager loads   */
/*  the overlay code and begins execution.  (This overlay manager  */
/*  does not check to see if the overlay is already in internal    */
/*  memory.)  A DMA transfer is performed to load in the memory    */
/*  overlay.                                                       */

#include <def21262.h>

.SECTION/DM             dm_data;

/*  The following constants are defined by the linker.    */
/*  These constants contain the word size, live location */
/*  and run location of the overlay functions.            */

.EXTERN _ov_word_size_run_1;
.EXTERN _ov_word_size_run_2;
.EXTERN _ov_word_size_run_3;
.EXTERN _ov_word_size_run_4;
.EXTERN _ov_word_size_live_1;
.EXTERN _ov_word_size_live_2;
.EXTERN _ov_word_size_live_3;
.EXTERN _ov_word_size_live_4;
.EXTERN _ov_startaddress_1;
.EXTERN _ov_startaddress_2;
.EXTERN _ov_startaddress_3;
.EXTERN _ov_startaddress_4;
.EXTERN _ov_runtimestartaddress_1;
.EXTERN _ov_runtimestartaddress_2;
.EXTERN _ov_runtimestartaddress_3;
.EXTERN _ov_runtimestartaddress_4;

/*  Placing the linker constants in a structure so the overlay  */
/*  manager can use the appropriate constant based on the    */
/*  overlay id.                          */
#define PHYS_WORD_SIZE(run_size,live_size) (48 / (live_size/run_size))
.import "OverlayStruct.h";

.struct OverlayConstantsList _OverlayConstants = {
_ov_startaddress_1, _ov_startaddress_2,_ov_startaddress_3, _ov_startaddress_4,
_ov_runtimestartaddress_1, _ov_runtimestartaddress_2,_ov_runtimestartaddress_3,
_ov_runtimestartaddress_4,
_ov_word_size_run_1,_ov_word_size_run_2,_ov_word_size_run_3,_ov_word_size_run_4,
_ov_word_size_live_1,_ov_word_size_live_2,_ov_word_size_live_3,
_ov_word_size_live_4,
PHYS_WORD_SIZE(_ov_word_size_run_1,_ov_word_size_live_1),
PHYS_WORD_SIZE(_ov_word_size_run_2,_ov_word_size_live_2),
PHYS_WORD_SIZE(_ov_word_size_run_3,_ov_word_size_live_3),
PHYS_WORD_SIZE(_ov_word_size_run_4,_ov_word_size_live_4),
};

/*  software stack to temporarily store registers corrupted by overlay manager */
.VAR  ov_stack[20];
```

```
.VAR  start_addr_stack [2] = 0,0;
.VAR  end_addr_stack [2] = 0,0;

/*************************************************************************/
/*          Overlay Manager Function                                     */

.SECTION/PM             pm_code;

_OverlayManager:
.GLOBAL _OverlayManager;

/*  _overlayID has been defined as R0.  R0 is set in the PLIT of LDF.   */
/*  Set up DMA transfer to internal memory through the external port.   */

/*  Store values of registers used by the overlay manager in to the     */
/*  software stack.                                                      */
dm(ov_stack)=i7;
dm(ov_stack+1)=m7;
dm(ov_stack+2)=l7;
i7=ov_stack+3;
m7=1;
l7=0;

dm(i7,m7)=i8;
dm(i7,m7)=m8;
dm(i7,m7)=l8;
dm(i7,m7)=r2;
dm(i7,m7)=r3;
dm(i7,m7)=r4;
dm(i7,m7)=r5;
dm(i7,m7)=r6;
dm(i7,m7)=ustat1;

/* Use the overlay id as an index (must subtract one)  */
R0=R0-1; /* Overlay ID -1 */
m8=R0;   /*  Offset into the arrays containing linker  */
         /*  defined overlay constants.                */

r0=i6;  dm(i7,m7)=r0;
r0=i0;  dm(i7,m7)=r0;
r0=m0;  dm(i7,m7)=r0;
r0=l0;  dm(0,i7)=r0;

l8=0; // Clear L0 & L8 to keep DAGs from using Circ Buffers
l0=0;
r2=3; // Save multiplier to convert 48-bit address to 32-bit
r6=1;

m0=m8; /* Overlay ID - 1 */

/*  Get overlay run and live addresses from memory and use to */
/*  set up the master mode DMA.                               */
i8 = _OverlayConstants->runAddresses;
i0 = _OverlayConstants->liveAddresses;

r0=0;           dm(PPCTL) = r0; dm(start_addr_stack)=r0; dm(end_addr_stack)=r0;
r0=dm(m0,i0);   dm(EIPP)=r0;
```

```
//Convert 48-bit run addr to 32-bit addr (48 * 3/2 = 32)
r4=pm(m8,i8);

r3=fext r4 by 0:16; r0=r4-r3;
r3=r3*r2 (ssi);
r3=lshift r3 by -1;
r0=r0 or fdep r3 by 0:16;
dm(IIPP)=r0;

/* If 48-bit address is odd, save first 16-bits to restore after the
   overlay is loaded.*/
btst r4 by 0; if not sz jump addr_end;
r0=lshift r0 by 1;  dm(start_addr_stack)=r0;   // Store short word address
i6=r0; r0=dm(0,i6); dm(start_addr_stack+1)=r0; // Store the affected word
r6=lshift r6 by 1;

addr_end:
i0=_OverlayConstants->runNumWords;  /* Number of words stored in internal memory */
                 /*  Most likely the word size will be 48 bits  */
                 /*  for instructions.              */

i8=_OverlayConstants->liveNumWords; /* Number of words stored in external memory */

r0=pm(m8,i8);
i8=_OverlayConstants->physicalLen;
ustat1=pm(m8,i8);
/* Make sure that there is a whole number of 32-bit words in ECPP
   (and round up if there is not)*/
r0=r0+1; r0=lshift r0 by -1; bit tst ustat1 16; if tf jump (pc,4);
r0=r0+1; r0=lshift r0 by -1; r0=lshift r0 by 1;
r0=lshift r0 by 1;   dm(ECPP)=r0;
r0=dm(m0,i0);
/* If the overlay section will end on an even address,
   save the last 16-bits to restore*/
r4=r4+r0; btst r4 by 0; if not sz jump save_internal_count;
r3=fext r4 by 0:16; r4=r4-r3;   r3=r3*r2 (ssi); r3=lshift r3 by -1;
r4=r4 or fdep r3 by 0:16;
r4=lshift r4 by 1;  r4=r4+1; dm(end_addr_stack)=r4;
i6=r4; r4=dm(0,i6); dm(end_addr_stack+1)=r4;
save_internal_count:
r0=r0*r2 (ssi); r0=r0+1; r0=lshift r0 by -1;   dm(ICPP)=r0;

r0=1;           dm(EMPP)=r0;
dm(IMPP)=r0;

//Determine if the external memory is 8- or 16-bit width
r6=lshift r6 by -1;
r0=dm(ICPP);
r3=dm(ECPP);
r0=lshift r0 by 1;
comp(r0,r3);
if lt jump external8;

//Set up for 16-bit external memory
external16:
r0=dm(ICPP);
r0=r0+r6;
dm(ICPP)=r0;
```

```
r0=dm(EIPP);
r0=r0-r6;
dm(EIPP)=r0;
r6=lshift r6 by 1;
r3=r3+r6;
dm(ECPP)=r3;
ustat1=PPBHC|PPDUR4|PP16;
dm(PPCTL)=ustat1;
jump startdma;

//Set up for 8-bit external memory
external8:
r0=dm(ICPP);
r0=r0+r6;
dm(ICPP)=r0;
r6=lshift r6 by 1;
r0=dm(EIPP);
r0=r0-r6;
dm(EIPP)=r0;
r6=lshift r6 by 1;
r3=r3+r6;
dm(ECPP)=r3;
ustat1=PPBHC|PPDUR4;
dm(PPCTL)=ustat1;

startdma:
bit set ustat1 PPEN|PPDEN;
dm(PPCTL)=ustat1;
nop;nop;nop;
/*  Wait for DMA to complete.  Note that, in this example's
    code, the DMA may not complete if another interrupt fires
    before the DMA's completion.  If this is consideration
    in your system, be sure to add this check to this code.  */
dma1_wait:      idle;

    //-------- Wait for PPI DMA to complete using polling------
    r0=dm(PPCTL);
    btst r0 by 17;
    if not sz jump (pc,-2);

/*  Restore register values from stack  */
m7=-1;
r0=dm(i7,m7);   l0=r0;
r0=dm(i7,m7);   m0=r0;
r0=dm(i7,m7);   i0=r0;
r0=dm(i7,m7);   i6=r0;
ustat1=dm(i7,m7);
r6=dm(i7,m7);
r5=dm(i7,m7);
r4=dm(i7,m7);
r3=dm(i7,m7);
r2=dm(i7,m7);
l8=dm(i7,m7);
i8=r1;
m8=0;

/*  Flush the cache.  If an instruction in previous overlay  */
/*  had been cached, it may be executed instead of the       */
/*  current overlays instruction.  (If pm transfers align.)  */
flush cache;

//Restore the saved 16-bit words corrupted by 32-bit to 48-bit conversion
r0=dm(start_addr_stack);
r1=0;
comp(r1,r0);
if eq jump check_end_stack;
r1=i6;
i6=r0;
r0=dm(start_addr_stack+1);
dm(0,i6)=r0;
i6=r1;

check_end_stack:
r0=dm(end_addr_stack);
r1=0;
comp(r1,r0);
if eq jump overlay_manager_end;
r1=i6;
i6=r0;
r0=dm(end_addr_stack+1);
dm(0,i6)=r0;
i6=r1;

overlay_manager_end:
r1=dm(i7,m7);
r0=dm(0,i7);

i7=dm(ov_stack);
m7=dm(ov_stack+1);
l7=dm(ov_stack+2);
nop;
/*  Jump to the location of the function to be executed.  */
jump (m8,i8) (db);
i8=r0;
m8=r1;

_OverlayManager.end:

/**************************************************************************/
```

Listing 1. Basic overlay manager for overlays stored in SRAM

```
/*******************************************************************************/
/*                                                                             */
/* File Name:  Ovl_Init.asm                                                    */
/*                                                                             */
/* Date:  August 29, 2003                                                      */
/*                                                                             */
/* Purpose:  This file runs through the ROM to look for where the overlays'    */
/* sections reside.  It then places the info for each section (live address,   */
/* count size, and type) into respective buffers.  It also checks the types and */
/* accounts for how many words to increment in the ROM to look for the next     */
/* section's info.                                                             */
/*******************************************************************************/
```

```
#include "ovlay.h"
#include <def21262.h>

.extern num_ovl_sec;
.extern total_live_addr;
.extern total_sec_size;
.extern total_sec_type;
.extern read_data_bytes;

.section/dm dm_data;
.var scratch[3];

.section/pm pm_code;
.global _OvlInit;

_OvlInit:
                ustat1 = PPBHC|PPDUR23;
                dm(PPCTL)=ustat1;

                r0=0x1000600;
                dm(EIPP)=r0;
                M12=1;   //DAG2
                dm(EMPP)=m12;
                dm(IMPP)=m12;

                //Init index registers
                I9=total_live_addr;
                I10=total_sec_size;
                I13=total_sec_type;
                I14=num_ovl_sec;



// ==========================  READ_BOOT_INFO ====================================
// Places TAG in R0, Internal Count in R2, and Destination Address in R3
// -------------------------------------------------------------------------------
read_boot_info: CALL read_prom_word;    // read first tag and count
                jump check_routine;

// ==========================  read_prom_word  ===================================
// read_prom_word is a callable subroutine that DMA's in one 48-bit word from PROM
// It places the MS 32-bits in R3 and the LS 16 (right justified) in R2.
// For example, given the 48-bit word 0x112233445566,
//      R2 holds 0x00005566
//      R3 holds 0x11223344
// -------------------------------------------------------------------------------
read_prom_word: R0=scratch;
                DM(IIPP)=R0;           // 0x40004 = DMA destination address
                R0=3;
                DM(ICPP)=R0;
                R0=12;
                DM(ECPP)=R0;

                call read_data_bytes;

                R0=dm(scratch);                  // Copy TAG to R0
                RTS (DB);
```

```
                    R2=dm(scratch+1);    // Copy count to R2
read_prom_word.end: R3=dm(scratch+2);     // Copy address to R3



// =============================  CHECK_ROUTINE  =================================
// R0 holds the TAG, R2 holds the Word Count, and R3 holds the Destination Address
// Overlays have been mapped to a dummy live address in SDRAM space,
// Therefore, check for a destination address of greater than or equal to 0x600000
// in R3.  To advance the count by which to increment readings from the ROM, check
// each tag info and determine whether it needs to be incremented by 12 words, 6
// words, or none.
// Also check for final init.  When final init tag comes across, then the
// end of program is coming and there are no more overlays.
// ------------------------------------------------------------------------------

check_routine:
R9=PASS R0;             // check TAG

IF EQ JUMP final_init;  r9=r9-1;          // jump if fetched tag was 0, else tag--
IF EQ JUMP ZERO_init;  r9=r9-1;           // jump if fetched tag was 1, else tag--
IF EQ JUMP ZERO_init;    r9=r9-1;         // jump if fetched tag was 2, else tag--
IF EQ JUMP Internal_16;    r9=r9-1;       // jump if fetched tag was 3, else tag--
IF EQ JUMP Internal_32;    r9=r9-1;       // jump if fetched tag was 4, else tag--
IF EQ JUMP Internal_48;    r9=r9-1;       // jump if fetched tag was 5, else tag--
IF EQ JUMP Internal_64;    r9=r9-1;       // jump if fetched tag was 6, else tag--
IF EQ JUMP initliveinfozeros;   r9=r9-1;  // jump if fetched tag was 7, else tag--
IF EQ JUMP initliveinfozeros;  r9=r9-1;   // jump if fetched tag was 8, else tag--
IF EQ JUMP initliveinfo;   r9=r9-1;       // jump if fetched tag was 9, else tag--
IF EQ JUMP initliveinfo;  jump(pc,0);     // jump if fetched tag was A, else it
                                          // was an invalid TAG, so trap for debug

initliveinfo:
    R8 = OVLY_LIVE_START;
    R7 = R3-R8;
    if lt jump Internal_32;
    R8 = OVLY_LIVE_END;
    R7 = R3-R8;
    if gt jump Internal_32;
    r12=dm(EIPP);
    PM(I9,M12) = R12;       //write ROM live address to total_live_addr buffer
    PM(I10,M12) = R2;       //write word count to total_sec_size buffer
    PM(I13,M12) = R0;

    jump Internal_32;

initliveinfozeros:
    R8 = OVLY_LIVE_START;
    R7 = R3-R8;
    if lt jump ZERO_init;
    R8 = OVLY_LIVE_END;
    R7 = R3-R8;
    if gt jump ZERO_init;
    PM(I9,M12) = R12;   //write ROM live address to total_live_addr buffer
    PM(I10,M12) = R2;       //write word count to total_sec_size buffer
    PM(I13,M12) = R0;
    jump ZERO_init;
```

```
//no increment needed
ZERO_init:
    R1=0;   // No data to skip
    R2=0;   // No data to skip
    jump Update_PROM_address;

Internal_16:
    R2=R2+1;       // If count is odd, round up to make it even
    R1=0xFFFFFFFE; // because the loader pads an extra word
    R2=R2 AND R1;  // when the count is odd
    R1=2;          // Count is in 16-bit words, so multiply by 2
    jump Update_PROM_address;

Internal_32:
    R1=4;   // Count is in 32-bit words, so multiply by 4
    jump Update_PROM_address;

Internal_48:
    R2=R2+1;       // If count is odd, round up to make it even
    R1=0xFFFFFFFE; // because the loader pads an extra word
    R2=R2 AND R1;  // when the count is odd
    R1=6;          // Count is in 48-bit words, so multiply by 6
    jump Update_PROM_address;

Internal_64:
    R1=8;          // Count is in 64-bit words, so multiply by 8
    jump Update_PROM_address;

Update_PROM_address:
    R0=dm(EIPP);
    R1=R1*R2(SSI);
    R0=R0+R1;
    DM(EIPP)=R0;
    jump read_boot_info;

final_init:
    rts;
_OvlInit.end:
```

Listing 2. Parse data stored in a flash to obtain overlay information

```
/*********************************************************************************/
/*                                                                               */
/* File Name:  Ovl_Sec_Info.asm                                                  */
/*                                                                               */
/* Date:  August 27, 2003                                                        */
/*                                                                               */
/* Purpose:  This file parses all the info collected in the Ovl_Init.asm file    */
/*           for each overlay.  It checks for the number of sections within each */
/*           overlay.  It also checks each overlay section's type and size.  It  */
/*           accounts for the overlay id that's included in the data for each    */
/*           overlay by adding 6 words to the live address.  It accounts for     */
/*           the overlay id by subtracting 1 from the count size of the overlay. */
/*           Then, the information is written to the respective address and size */
/*           buffers.                                                            */
/*********************************************************************************/
```

```
#include "ovlay.h"

#include <def21262.h>
.extern runWordSize;

.extern num_ovl_sec;
.extern read_base_addr;
.extern read_buffer_addr;
.extern read_buffer_length;
.extern read_data_bytes;
.extern total_live_addr;
.extern total_sec_size;
.extern total_sec_type;

.segment/pm pm_code;
.global _OvlSecInfo;

_OvlSecInfo:

/* This first section of code checks the number of sections there are within each
   overlay */

lcntr = NUMBER_OF_OVERLAYS;             //count of overlays in this project

I12 = runWordSize;
I10 = total_sec_size;
I13 = num_ovl_sec;
I9 = total_live_addr;

do check_ovl.end until lce;
check_ovl:  R12=0;       //R12 stores number of sections.  Initiate to 0 to start.

R9 = PM(I12,1);     //read the total overlay size
repeat_check:   R8 = PM(I10,1);     //read the individual section size

R12 = R12+1;            //increment the total number of sections by 1

R9 = R9-R8;                  //subtract section size from total overlay size
if gt jump repeat_check;     //if not equal, then there are more sections in this
                             //overlay.  repeat the check.
nop;nop;

check_ovl.end:  PM(I13,1)=R12;  /*if equal, then there are no more sections in this
                                  overlay write number of sections in each overlay
                                  to buffer */
rts;

/* The next section of code accounts for the 1 extra address in the total_live_addr
   buffer holding the address of the overlay id information and the 1 extra word
   count in the total_sec_size buffer. */

I10 = total_sec_size;   //pointer to beginning of section size buffer
R10 = 0x4;              //R10 holds loop counter.  In this case, 4 overlays
R4=0x6;                 //6 8-bit words in ROM to increment count by
remove_id:      R2 = PM(I13,M12);       //Read number of sections in overlay
M14 = R2;               //Set M14 to number of sections in overlay
R6 = PM(I10,M9);        //Read individual section size buffer
```

```
R3 = PM(I9,M9);          //Read live address buffer
R6 = R6-R5;              //Subtract the section size by 1 (account for ovl id)
R3 = R3+R4;              //Add live address by 0x6 (account for ovl id)
PM(I10,M14)=R6;          //Write back section size to buffer
PM(I9,M14)=R3;           //Write back live address to buffer
R10 = R10-1;             //Do this for all four overlays.
if ne jump remove_id;    //Repeat if not done for all four overlays.

done:       rts;
_OvlSecInfo.end:
```

Listing 3. Parse Tags stored in the flash to generate information for each overlay

References

[1] *Using Memory Overlays (EE-66).* March 1999. Analog Devices, Inc.

[2] *Using Code Overlays from ROM on the ADSP-21161N EZ-KIT Lite (EE-180).* December 2002. Analog Devices, Inc.

[3] *Using External Memory with Third Generation SHARC Processors and the Parallel Port (EE-220).* Rev 2. February 2005. Analog Devices, Inc.

[4] *Programming an ST M25P80 SPI Flash with ADSP-21262 SHARC DSPs (EE-231).* Rev 1. February 2004. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 2 – March 11, 2005 by Brian M.and Divya S.	Generalized the discussion to third generation SHARC processors. Title changed to *Code Overlays on the Third Generation SHARC Family of Processors*
Rev 1 –February 17, 2004 by Brian M.	Initial Release of *Code Overlays on the ADSP-2126x SHARC Family of DSPs*