



Technical notes on using Analog Devices DSPs, processors and development tools
 Visit our Web resources <http://www.analog.com/ee-notes> and <http://www.analog.com/processors> or
 e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support

Implementing Overlays on ADSP-21160 SHARC® Processors

Contributed by Jeyanthi Jegadeesan

Rev 1 – March 9, 2006

Introduction

This EE-Note discusses the implementation of software overlays on ADSP-21160 SHARC® processors. It also describes the external memory packing of ADSP-21160 used for overlays and demonstrates the software overlay technique on ADSP-21160 using a simple example code on the ADSP-21160 EZ-KIT Lite® board. An introduction to overlays using SHARC processors can be found in *Using Memory Overlays (EE-66)*^[1], which discusses the details and concepts of using the code overlays from external RAM on SHARC processors.

Overlays on SHARC Processors

Overlays are used when a processor does not have sufficient internal memory to hold all of the application's data and/or program code. In these situations, part of data/program code is loaded into internal memory during the booting process and the remaining part is placed in external memory. When program code (or data) in external memory has to be executed, it is loaded into internal memory and executed. The memory address range in which the overlay function resides in the external memory is called “live address space”, and the memory address where the program is executed in internal memory is called “run address space”. The code responsible for transferring the data/program code from external memory to internal memory at runtime is called an *overlay manager*. Typically, it

resides in a reserved space of the processor’s internal memory. The VisualDSP++® tools provide the overlay support for all SHARC processors. Because of this, the linker changes calls to functions located in an overlay section into calls to the overlay manager.

Overlays on ADSP-21160 SHARC Processors

Overlays can be implemented using the ADSP-21160 processor’s external memory space. Figure 1 shows a block diagram of the external memory interface of the ADSP-21160. The external port data, address, MSx, and RDH/L and WRH/L lines are used for the external memory interface.

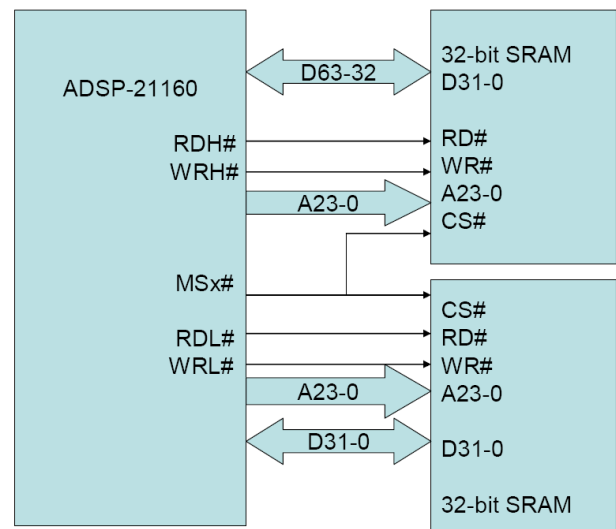


Figure 1. External Memory Interface

ADSP-21160 External Memory Interface and DMA Packing Modes

The external port of the ADSP-21160 has a 64-bit external data bus, 32-bit address bus, and the other external port control signals (MS_x , $RDH\backslash L$, and $WRH\backslash L$) for external memory interface. The unpacked data and instructions are placed in the processor's 64-bit external memory as shown in Figure 2.

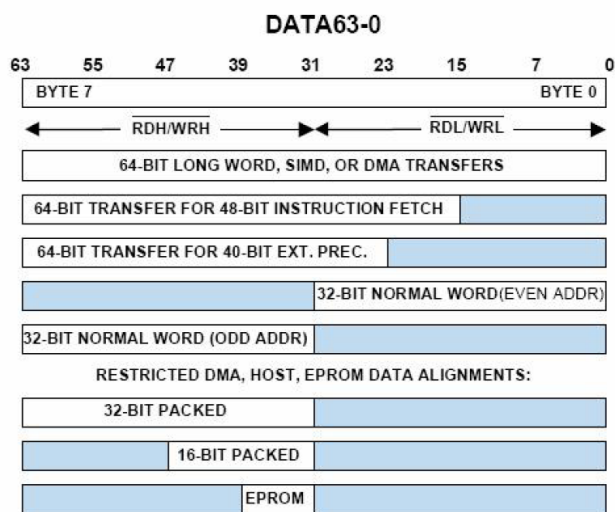


Figure 2. External Port Data Alignment

The external memory map is organized such that consecutive addresses access adjacent 32-bit memory locations. For unpacked data, the processor stores the data in both the odd and even addresses. For packed data, the processor uses only the odd addresses for storing the data. The overlay code is placed as data in the external memory. When the overlay code is placed in external memory, the emulator uses core writes to place the data; due to this, data is written into both the odd and even addresses of the external memory. In this condition, the overlay manager uses the DMA with no packing enabled to transfer the data from external memory to internal memory. If the overlay manager uses DMA with packing enabled to transfer the data from external memory to internal memory, the DMA will only read the data from the odd addresses because when packing is enabled in

DMA mode the ADSP-21160 uses only the $RDH\#$ and $WRH\#$ lines for data transfer. Due to this, data in even addresses will not be read into internal memory. The overlay manager calculates the 32-bit value corresponding to the 48-bit run-time address and uses it as the internal memory address for the DMA. Once the data is written into the internal memory, it is accessed as a 48-bit instruction by using the 48-bit run-time address. In a system that uses a single 32-bit external memory connected to the processor's higher-order address lines, the overlay code can be placed only in the odd addresses. For these systems, the overlay manager uses DMA with 32-bit to 48-bit packing enabled to transfer the data from external memory.

Packing Method in Visual DSP++

The Linker Description File's (.LDF) `PACKING ()` command specifies the order used by the linker to place bytes in memory. This ordering places data in memory in the sequence the processor uses as it transfers data. When mapping the overlays to external memory, the `PACKING ()` command must be used to place the overlay data properly in external memory. Use the following packing mode with the actual device:

```
PACKING (12 B3 B4 B5 B6 B0 B11 B12 B1
         B2 B0 B7 B8 B9 B10 B0)
```

VisualDSP++ Support

The VisualDSP++ linker automatically generates *overlay constants*, which configure the external port DMA parameters in the overlay manager. Each overlay has a word size, run-time address, and live address used by the overlay manager to determine where the overlay resides and where it is executed. The linker-generated constants (where N is the ID# of the overlay) are:

- `_ov_startaddress_N`
(live address space)
- `_ov_word_size_run_N`
- `_ov_word_size_live_N`

- `_ov_runtimestartaddress_N`
 (run address space)

The linker is also responsible for resolving the symbol addresses of overlay data and labels.

Overlay Manager with DMA No Packing - Example

This section shows a simple example program to illustrate the software overlay implementation on ADSP-21160 processors. The blink example program provided with this EE-Note uses the overlay manager, which initiates a DMA without packing to download the overlay function. It toggles three flag pins. This program uses two software overlay functions that reside in external memory and one function that resides in internal memory.

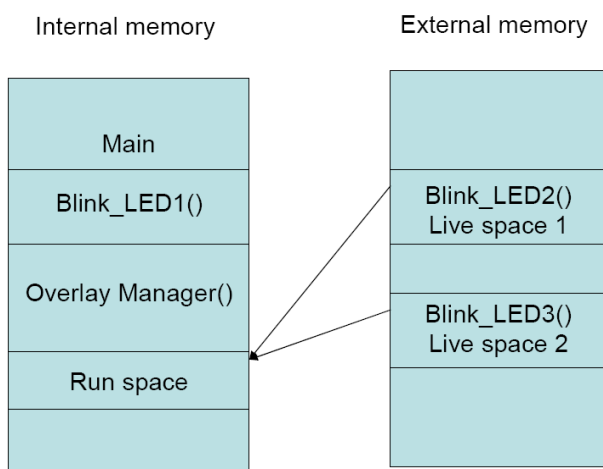


Figure 3. Simple Memory Overlay Example

The main program calls the internal memory function and the two software overlay functions in an infinite loop. Each function toggles a different flag for a small duration. Figure 3 illustrates where the overlay functions, overlay manager, and the main function are stored in the memory. The `main()`, `Blink_LED1()`, and `Overlay_Manager()` functions are loaded into the internal RAM of the ADSP-21160 initially. The `Blink_LED2()` and `Blink_LED3()` overlay functions are loaded in external memory. During

runtime, when the overlay function is called, the function is loaded into internal memory and then executed.

Linker Definitions

This section discusses the declarations required in the Linker Description File (.LDF) for the example program explained above.

```
// "run" space for Overlay functions
seg_pmco_ovl {
    TYPE(PM RAM)
    START(0x00047288)
    END(0x00049aa9)
    WIDTH(48)
}
// "live" space for Overlay functions
ovl_ext {
    TYPE(DM RAM)
    START(0x02800000)
    END(0x0281FFFF)
    WIDTH(32)
}
```

Listing 1. Run and Live Space Declaration in .LDF File

The statements in Listing 1 under the `MEMORY { }` command of the .LDF file define the target memory (i.e., the run space and live space addresses of the overlay program code). The live space, which is specified in the external memory, helps the linker generate the overlay constants. The statements specified in Listing 2, under `PROCESSOR { }` command of the .LDF file maps the overlay code and data to the external memory of the system. In Listing 2, the overlay commands instructs the linker that a specific section (e.g., `seg_pmco`) from a specified input file (`Blink1.doj`) is to be used as an input for this overlay segment (`seg_pmco_ovl`). For .LDF file syntax and definitions, refer to the *VisualDSP++ Linker and Utilities Manual*.

```
seg_ovl {
    OVERLAY_INPUT{
        OVERLAY_OUTPUT($COMMAND_LINE_OUTPU
            T_DIRECTORY\OVLY_1.ovl)
        ALGORITHM(ALL_FIT)
```

```

INPUT_SECTIONS(Blink1.doj
               (seg_pmco))

PACKING(12 B3 B4 B5 B6 B0 B11
        B12 B1 B2 B0 B7 B8 B9 B10 B0)
}>ovl_ext

OVERLAY_INPUT{
OVERLAY_OUTPUT($COMMAND_LINE_OUTPU
               T_DIRECTORY\OVLY_2.ovl)

ALGORITHM(ALL_FIT)

INPUT_SECTIONS(Blink2.doj
               (seg_pmco))

PACKING(12 B3 B4 B5 B6 B0 B11
        B12 B1 B2 B0 B7 B8 B9 B10 B0)
}>ovl_ext
}>seg_pmco_ovl

```

Listing 2. Specifying Overlay Sections in the .LDF File

The Linker Description File (.LDF) in an overlay project has the procedure linkage table (PLIT) command. The PLIT is a jump table constructed by the linker in root memory. Each call to an overlay section is replaced by a call to the PLIT. The PLIT {} commands provide a template by which the linker generates distinct assembly code for each overlay section. Listing 3 shows an example PLIT section that would be defined in the .LDF file. This section is defined only once in the .LDF file. However, the linker generates separate PLIT code for each overlay function call (OVLY_1 and OVLY_2). In other words, the PLIT {} command in an .LDF file inserts assembly instructions that handle calls to functions in overlays. The template in Listing 3 from which the linker generates assembly code when a symbol resolves to a function in overlay memory. Typically, the code handles a call to a function in overlay memory by calling the overlay memory manager. Since each call to an overlay function is replaced by a call to the PLIT, you must place the jump instruction to the user-defined overlay manager code in the PLIT section.

```

PLIT {
    R0 = PLIT_SYMBOL_OVERLAYID;

/* Assigns "run" address of resolved
symbol to R1.. */

    R1 = PLIT_SYMBOL_ADDRESS;

    // Execute overlay manager

    jump _OverlayManager;
}

```

Listing 3. Simple .LDF File PLIT Entry Example

Each overlay module declared in the .LDF file has a unique copy of the PLIT entry defined in the .LDF file. The example program has two code overlays, and a simple PLIT is declared in the .LDF file shown in Listing 3. The corresponding PLIT table for the two overlay functions would look like Listing 4.

```

.section/code .plt;
plt_1__._OVL1_Blink_LED2:
R0 = 1;
R1 = _OVL1_Blink_LED2;
jump _OverlayManager;

plt_2__._OVL2_Blink_LED3:
R0 = 2;
R1 = _OVL2_Blink_LED3;
jump _OverlayManager;

```

Listing 4. Example PLIT Table

The R0 register is loaded with the overlay ID, and the R1 register is loaded with the run address used by the overlay manager.

Overlay Manager with DMA Without Packing

The overlay manager is responsible for transferring data/code from the live space to the run space. The linker generates various overlay constants, such as live addresses, run addresses, live word sizes for each live address, and run word sizes. The linker-generated constants must be declared as external constants as shown in Listing 5 in the overlay manager function. Initially, the overlay manager checks whether the

overlay function is available in the memory using the overlay ID. If the function is not yet loaded in internal memory, it loads the overlay function into internal memory using DMA transfer. The overlay manager calculates the equivalent 32-bit address for the 48-bit run address space. Listing 6 shows the code that calculates the 32-bit equivalent address of the 48-bit run address space. Then the overlay manager configures the external port DMA parameters using the linker-generated overlay constants, and the external port DMA transfer without packing is initiated. Figure 4 shows a simple overlay manager that uses DMA without packing. This overlay manager is used in the system where two 32-bit external memory are connected to the odd and even addresses of the processor.

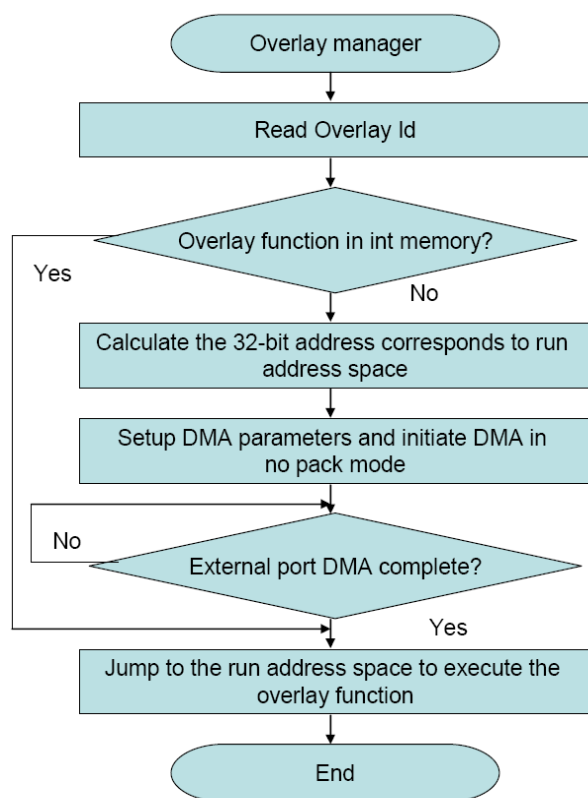


Figure 4. Flowchart: Overlay Manager With DMA Without Packing

```

/* 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_runtimestartaddress_1;
.extern _ov_runtimestartaddress_2;
.extern _ov_startaddress_1;
.extern _ov_startaddress_2;
.extern _ov_word_size_live_1;
.extern _ov_word_size_live_2;
  
```

Listing 5. Declaration of Linker-Generated Constants

```

i8 = runAddress;
R0=pm(m8,i8);
// Address translation
R1 = 0x40000 ;
R1 = R0 - R1 ;
R0 = LSHIFT R1 BY -1 ;
R1 = R0 + R1 ;
R0 = 0x40000 ;
R0 = R0 + R1 ;
R0 = R0 - 1 ;
R0 = R0 - 1 ;

dm(II10)=R0; // Internal address
              for DMA
  
```

Listing 6. Code to Calculate the 32-bit Equivalent Address

In this example, the linker constants are placed in an array so that the overlay manager can use the appropriate constants based on the overlay ID. The required data can be extracted from these arrays to set up the external port DMA transfer (Listing 7).

```

dm(II10)=R0; /*Set internal address*/
i0=liveWordSize; /*Number of words
                  stored in internal memory */

R0=1;
dm(EM10)=R0; /*External modifier */
i8=liveWordSize; /* Number of words
                  stored in external memory */

R0=1;
dm(IM10)=R0; /*internal modifier */
  
```



```

R0=dm(m0,i0);
dm(C10)=R0; /* Set internal count */
R0=pm(m8,i8);
dm(EC10)=R0; /* Set external count */
R0=0x40401;
dm(DMAC10)=R0; /*Set DMA control
                register */

```

Listing 7. Setting up the DMA Parameters

Overlay Manager with DMA Packing

An overlay manager that uses DMA with packing is used for systems that have a 32-bit external memory connected to the higher-order address lines of the processor. The overlay manager configures the external port DMA parameters using the linker-generated overlay constants and the external port DMA transfer with 32-bit to 48-bit packing is initiated. Figure 5 shows the overlay manager that uses DMA in 32-bit to 48-bit packing mode. This EE-Note provides another example code that uses the overlay manager to initiate a DMA with 32-bit to 48-bit packing to download the overlay function. It toggles the two flag pins of the processors in an infinite loop. This program uses one software overlay function that resides in external memory and one function that resides in internal memory. The linker definitions discussed in the previous example code also applies to this application.

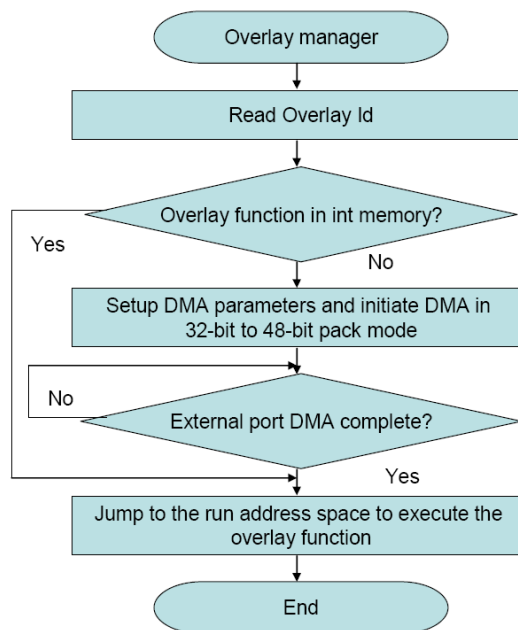


Figure 5. Flowchart: Overlay Manager with DMA Packing

This example code cannot be tested directly on the EZ-KIT Lite board using the emulator. The emulator uses core writes to place the data; due to this, the data will be written into both the odd and even addresses of the external memory. The boot kernel code available with this EE-Note is modified to place the overlay data in the external memory in higher-order addresses only. DMA with 32-bit to 48-bit data packing is used to transfer the data from the higher-order external memory locations to the internal memory.

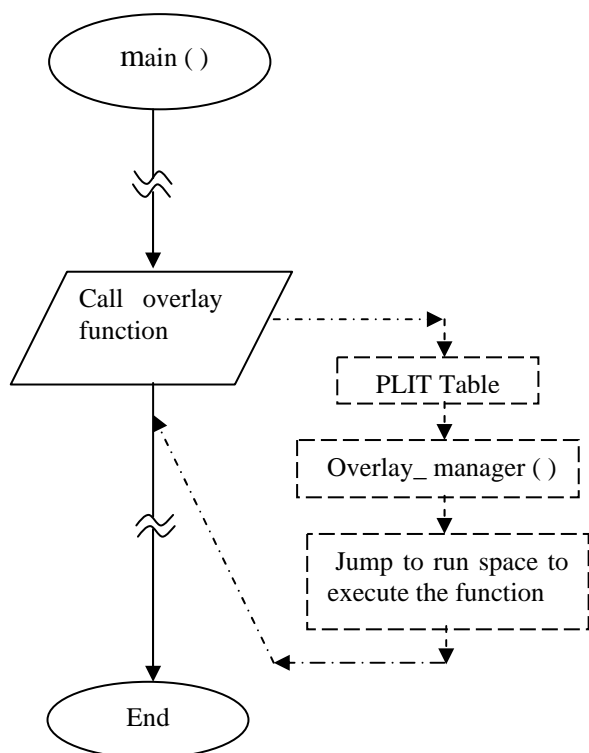


Figure 6. Overlay Function Execution Flowchart

In the main code, the overlay functions are called similar to ordinary (non-overlay) functions. Figure 6 shows the flowchart for the execution of an overlay function. For the overlay functions, the linker replaces the actual overlay function call with a call to the PLIT code generated for that particular overlay function.

For example, the instruction:

```
call OVL1_Blink_LED2;
```

is replaced automatically by:

```
call plt_1__._OVL1_Blink_LED2;
```

Looking back at the code:

```
plt_1__._OVL1_Blink_LED2:
    R0 = 1;
    R1 = _OVL1_Blink_LED2;
    Jump_OverlayManager;
```

In the `R0=0x0001` instruction, the content of the R0 register in the `plt_1__._OVL1_Blink_LED2` table is used as a pointer to access the linker-

generated data in the overlay manager. The instruction `(jump Overlay_Manager())` transfers program control to the overlay manager. As explained earlier, the overlay manager code initiates an external port DMA transfer of the `Blink_LED2()` overlay function from external memory to internal memory. Finally, the overlay manager executes a `JUMP` instruction to transfer program control to the `Blink_LED2()` overlay function in run space. The overlay manager also uses a software stack to save the contents of registers used in the overlay manager function.

Both overlay example codes are validated by booting from flash. The loader file (`.ldr`) attached with the example code can be burned into flash/EEPROM memory directly. The `.ldr` file will have the memory image of both the bootable and non-bootable parts of the code. During booting, the boot kernel instructions load the non-overlay code only; they do not load the overlay functions. The boot kernel loads the overlay code as data in the external memory using core writes. In the given example code during booting only the `main()`, `Blink_LED1()`, and `overlay_manager()` functions are loaded into the internal memory. The overlay functions mapped from `0x2800000` in external memory are not loaded into the internal memory. During runtime, when overlay functions are called, they are transferred into the run space and executed.

Testing Overlay Example Codes

Test the blink example code available for the overlay manager without packing using the emulator:

1. Load the `blink_code_overlay.dxe` file available with the EE-Note into internal memory.
2. Place a breakpoint after the `WAIT` register initialization and run until the breakpoint occurs.

3. In VisualDSP++, choose Settings > Preferences to open the Preferences dialog box and disable the Reset targets before load and Run to main after load options.
4. Reload the `blink_code_overlay.dxe` file.
5. Run the application.

The blink code example available for the overlay manager with DMA packing cannot be tested using the emulator. The `blink_code.ldr` file available with this EE-Note can be programmed into the flash using the Flash Programmer utility and tested.

Summary

The overlay manager codes discussed in this section are very simple. They use the DMA with no packing enabled or with 32-bit to 48-bit packing mode enabled to transfer overlay data from external memory to internal memory. The overlay codes are validated by booting from the flash. The default boot kernel is modified to work with the overlay manager example codes.

Source code for the overlay managers are provided with this EE-Note.

References

- [1] *Using Memory Overlays (EE-66)*. Analog Devices, Inc.
- [2] *ADSP-21160 SHARC DSP Hardware Reference*. Rev 3.0, November 2003. Analog Devices, Inc.
- [3] *VisualDSP++ 4.0 Linker and Utilities Manual*. Rev 1.0, January 2005. Analog Devices, Inc.
- [4] *Implementing Software Overlays on ADSP-218x DSPs with VisualDSP++ (EE-249)*. October 2004. Analog Devices, Inc.

Document History

Revision	Description
<i>Rev 1 – March 09, 2006 by Jeyanthi Jegadeesan</i>	Initial Release