



Implementing Software Overlays on ADSP-218x DSPs with VisualDSP++®

Contributed by Ramesh Babu and Aseem Vasudev Prabhugaonkar

Rev 1 – October 4, 2004

Introduction

This EE-Note discusses the implementation of software overlays on ADSP-218x DSPs. A simple code example, demonstrating the software overlay technique for ADSP-218x DSP, is discussed later in this document.

Early versions of the VisualDSP® toolset used a different scheme to support software overlays as discussed in the legacy EE-Note *Everything You Always Wanted to know about Overlays - *But were afraid to ask (EE-67)*. With VisualDSP++® 3.5 a new approach has been enabled.

What are Software Overlays?

Due to limited address space, many ADSP-218x derivatives provide multiple memory pages, called *hardware overlays*. The *software overlay* technique is a completely different scheme to populate the content of physical SRAM with different data or code on demand.

The software overlay technique is commonly used when a processor does not have sufficient internal memory to hold all of the application's data / 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 where 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*. It typically resides in a reserved space of the DSP's internal memory.

Software Overlays on ADSP-218x DSPs

Software overlays can be implemented using the ADSP-218x DSP's external byte-wide memory space. A byte-wide memory can be interfaced through the processor's BDMA port. The overlay code/data resides in the external byte-wide memory. The overlay manager is responsible for initializing byte DMA (BDMA) to transfer the data/code from external memory into internal memory when required.

VisualDSP++ Support

The VisualDSP++ linker automatically generates *overlay constants*, which configure the BDMA 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 addr. space)

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

Example Program

This section shows a simple example program to illustrate the software overlay implementation on ADSP-218x DSPs. The assembly program provided with this EE-Note toggles a flag pin at two different speeds. This program uses two software overlay functions which exist in the external byte memory.

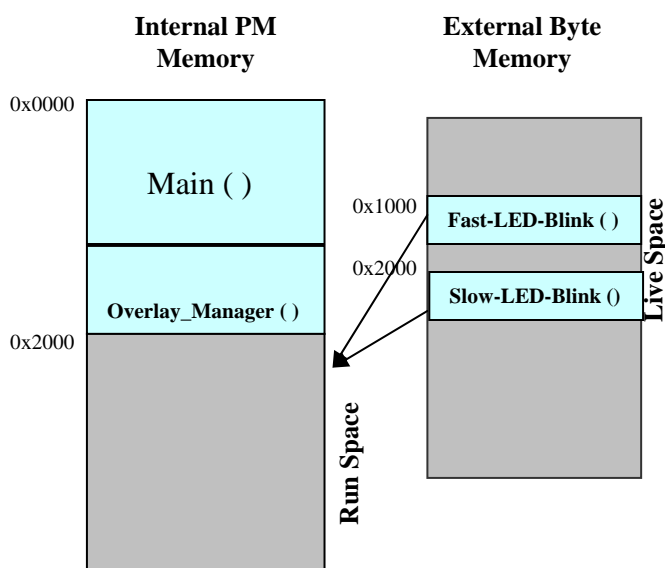


Figure 1. Simple Memory Overlay Example

The main program calls the two software overlay functions in an infinite loop. One of the software overlay functions (`Fast-LED-Blink()`) toggles the LED at a faster rate; the other (`Slow-LED-Blink()`) toggles the LED at a slower rate. Figure 1 illustrates where the overlay functions, overlay manager, and the main function are stored.

The `main()` and `Overlay_Manager()` functions are loaded into the internal RAM of the ADSP-218x during the booting process. The `Fast-LED-Blink()` and `Slow-LED-Blink()` overlay

functions are saved in external byte memory. During runtime, when an overlay functions is called, the function is loaded into internal memory and then executed. Accessing code and/or data overlays dynamically provides greater flexibility toward managing your DSP's internal memory.

Linker Definitions

Let's look into the declarations required in the Linker Description File (`.LDF`) for the example program explained above.

```
// "run" space for PMOVLAY pages
mem_pmovly {
    TYPE (PM RAM)
    START (0x002000)
    END (0x003fff)
    WIDTH (24)
}

// "live" space for PMOVLAY pages
mem_pmpage1 {
    TYPE (BM RAM)
    START (0x001000)
    END (0x001fff)
    WIDTH (8)
}
mem_pmpage2 {
    TYPE (BM RAM)
    START (0x002000)
    END (0x002fff)
    WIDTH (8)
}
```

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

The statements in Listing 1 in the `MEMORY { }` section of the `.LDF` file define the target memory (i.e., the run space and live space addresses of the overlay program code).

Live space, which is specified in the byte memory (BM), helps the linker generate the overlay constants.

The statements in Listing 2, which are specified in `PROCESSOR { }` section of the `.LDF` file, map code and data to the physical memory of a processor in a DSP system. In Listing 2, the commands tell the linker that a specific section (e.g., `pm_ovlay_1`) from a specified input file

(Fast-LED-Blink.doj) is to be used as an input for this overlay segment (mem_pmpage1). For .LDF file syntax and definitions, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* [2].

```
sec_pmpage {
  OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(pmpage1.ovl)
    INPUT_SECTIONS(
      Fast-LED-Blink.doj(
        pm_ovlay_1
      )
    )
  } > mem_pmpage1

  OVERLAY_INPUT {
    ALGORITHM(ALL_FIT)
    OVERLAY_OUTPUT(pmpage2.ovl)
    INPUT_SECTIONS(
      Slow-LED-Blink.doj(
        pm_ovlay_2
      )
    )
  } > mem_pmpage2
} > mem_pmovly
```

Listing 2. Specifying Overlay Sections in the .LDF

The Linker Description File (.LDF) in an overlay project has a section called a procedure linkage table (PLIT). The PLIT is a jump table in root memory constructed by the linker.

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 (pm_ovlay_1 and pm_ovlay_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 informs the linker which instructions to put into each PLIT entry. 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 {
  AY0 = PLIT_SYMBOL_OVERLAYID;
  JUMP Overlay_Manager;
}
```

Listing 3. Simple LDF 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.

```
plt_1__Fast_LED_Blink:
  AY0 = 0x0001;
  JUMP Overlay_Manager;

plt_2__Slow_LED_Blink:
  AY0 = 0x0002;
  JUMP Overlay_Manager;
```

Listing 4. Example PLIT Table

The AY0 register is loaded with the overlay ID, which is used by the overlay manager to determine the live address, run address, and word size of the overlay function.

Simple Overlay Manager

The overlay manager is responsible for transferring the 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. The overlay manager configures the BDMA parameters using the linker-generated overlay constants and then the BDMA transfer is initiated.

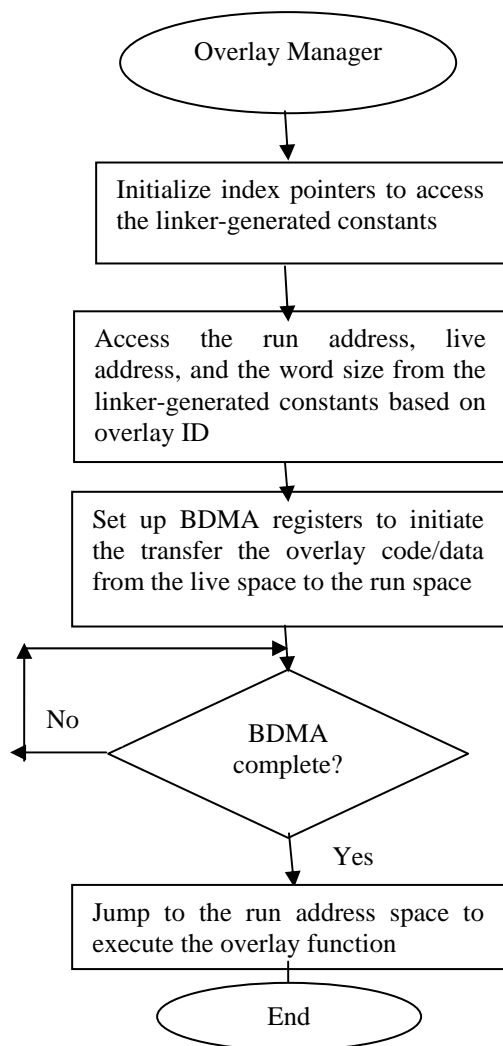


Figure 2. Overlay Manager Flowchart

```

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

.SECTION/DM data1;
.EXTERN _ov_word_size_run_1;
.EXTERN _ov_word_size_run_2;
.EXTERN _ov_word_size_live_1;
.EXTERN _ov_word_size_live_2;
.EXTERN _ov_startaddress_1;
.EXTERN _ov_startaddress_2;
.EXTERN _ov_runtimestartaddress_1;
.EXTERN _ov_runtimestartaddress_2;
  
```

Listing 5. Declaration of Linker-generated Constants

In this example, the linker constants are placed in an array (Listing 6) so that the overlay manager can use the appropriate constants based on the overlay ID.

```

.VAR liveAddresses[2] =
    _ov_startaddress_1,
    _ov_startaddress_2;

.VAR runAddresses[2] =
    _ov_runtimestartaddress_1,
    _ov_runtimestartaddress_2;

.VAR runWordSize[2]=
    _ov_word_size_run_1,
    _ov_word_size_run_2;

.VAR liveWordSize[2] =
    _ov_word_size_live_1,
    _ov_word_size_live_2;
  
```

Listing 6. Buffer Declaration using Linker-generated Constants

You can extract the required data from these arrays to set up the BDMA transfer (Listing 8).

```

// Get Pointer to access the constants
AR = AY0 -1;
// Save the pointer in Modifier Reg.
M0 = AR;
M3 = 0;

// Set array pointers to access
// Linker generated constants
I0 = liveAddresses;
I2 = runAddresses;
I3 = runWordSize;
modify(I0,M0);
modify(I2,M0);
modify(I3,M0);
  
```

Listing 7. Setting DAG Pointers to Access Appropriate Overlay Constants in Arrays

```

/* Load the BDMA registers and trigger
   the Byte DMA transfer
*/
AX0=DM(I0,M0);
DM(BDMA_External_Address)=AX0;

AX0=DM(I2,M3);
DM(BDMA_Internal_Address)=AX0;

AX0=0x000;
DM(BDMA_Control)=AX0;
  
```

```
AX0 = DM(I3,M0);
DM(BDMA_Word_Count)=AX0;
```

Listing 8. Initiating BDMA Transfer Example

Figure 2 shows a simple overlay manager.

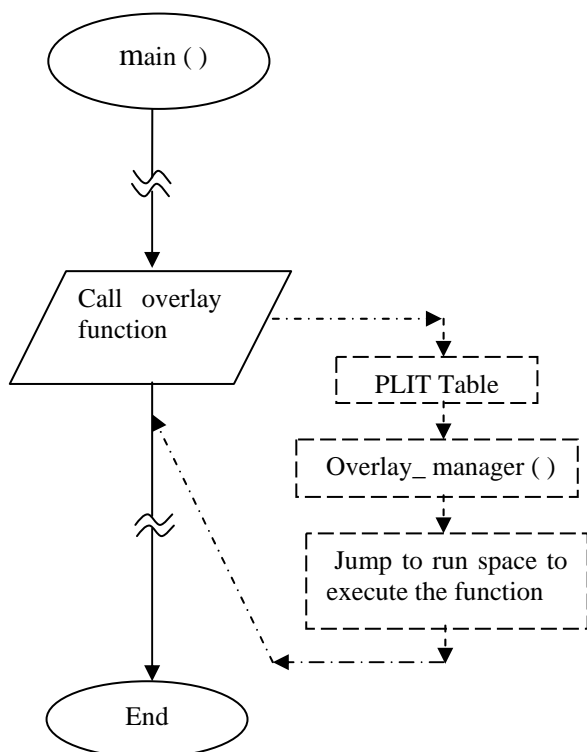


Figure 3. Overlay Function Execution Flowchart

In the main code, the overlay functions are called similar to ordinary (non-overlay) functions. 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 Fast-LED-Blink;
```

is replaced automatically by:

```
call plt_1__.Fast_LED_Blink;
```

Looking back at the code:

```
plt_1__Fast_LED_Blink:
    AY0 = 0x0001;
    JUMP Overlay_Manager;
```

In the `AY0=0x0001` instruction, the content of the `AY0` register in the `plt_1__Fast_LED_Blink` table is used as a pointer to access the linker-generated data in the overlay manager. The next instruction (`jump Overlay_Manager ()`) transfers program control to the overlay manager.

As explained earlier, the overlay manager code initiates a BDMA transfer of the `Fast-LED-Blink` overlay function from external memory to internal memory.

Finally, the overlay manager executes a `JUMP` instruction to transfer program control to the `Fast-LED-Blink()` overlay function in run space.

The overlay manager can also use a software stack to save the contents of registers used in the overlay manager function.

The loader file (`.BNM`) created using the VisualDSP++ can be burned into Flash/EEPROM memory directly. The `.BNM` file will have the memory image of both the bootable and non-bootable parts of the code. During booting, the page loader instructions load the non-overlay code only; they do not load the overlay functions. During booting, only the `main()` and `overlay_manager()` functions of the example are loaded into the DSP. However, the overlay functions mapped to `0x1000` and `0x2000` in byte memory are not loaded into the ADSP-218x during booting. During runtime, when overlay functions are called, they are transferred into the run space and executed.

The overlay manager discussed in this section is very simple. For complete code, refer to the source code provided with this EE-Note.

References:

- [1] *ADSP-218x DSP Hardware Reference*. First Edition, February 2001. Analog Devices, Inc.
- [2] *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*. Rev 1.0, October 2003. Analog Devices, Inc.
- [3] *Using Software Overlays with the ADSP-219x VisualDSP 2.0++ (EE-152)*. February 2002. Analog Devices, Inc.

Document History

Revision	Description
<i>Rev 1 – October 04, 2004 by Ramesh Babu</i>	Initial Release