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.

Understanding and Using Linker Description Files on SHARC® Processors

Contributed by Matt Walsh

Rev 2 – January 17, 2007

Introduction

Often, a programmer wants to control where a particular piece of code or data resides in a SHARC® processor's memory space. For example, on ADSP-2106x, ADSP-2116x, and ADSP-2137x processors, it is possible to execute from external memory; however, execution is performed at a reduced rate. In this case, the program will need to store and execute frequently used code internally and store rarely used code externally.

Regardless whether you want to relocate a C function or an assembly routine, the mechanism is the same. To map portions of code or data to specific memory sections, it is necessary to use the Linker Description File (referred to as the `.LDF` file). This EE-Note explains the functionality of the `.LDF` file and demonstrates its features through an example. The implementation details of specific applications that require relatively complex `.LDF` files (such as the external code-execution example previously mentioned) are beyond the scope of this document.

This EE-Note applies to all SHARC processors.

The first step toward gaining an understanding of the `.LDF` file is to understand the makeup of the files involved in building a processor executable (`.DXE`).

Source Files

Source files contain code written in C or assembly. The first step toward producing an executable is to compile and/or assemble these source files. The assembler outputs files called *object files*. (The compiler outputs assembly files that are then fed to the assembler.) The VisualDSP++® assembler produces object files that have a `.DOJ` extension.

Typically, these object files are output to the `~/YourProject/debug` directory.

Object Files

Object files produced by the compiler and assembler are divided into various *sections* (referred to as *object sections*). Each of these objects section holds a particular type of compiled source code. For example, an object section may hold program opcodes (48-bits wide) or data such as variables (16, 32, or 40 bits wide). Some object sections also hold information not pertinent to this discussion (since the information is not important to the user, such as debug-information, and so on). Each object section has a different name that is specified in the source code. Depending on whether the source is C or assembly, a different convention is used to specify the object section name.

In an assembly source, the code and/or data is placed below the `.SECTION` *`segment_name`*.

Refer to the following example.

```
.SECTION /dm seg_dmda
.VAR foo[3];

.SECTION /pm seg_pmco
r0 = 0x1234;
r1 = 0x4567;
r2 = r1 + r2;
```

In the example above, the `seg_dmda` object section would contain the `foo` array, and the three lines of code would be located in the `seg_pmco` object section.

In a C source file, the programmer would use the **`section`**`("`*`segment_name`*`")` directive, for example:

```
section("ext_data") int temp;
section("ext_code") void func1(void)
{
int x = 1;
}
void func2(void)
{
int i = 0;
}
```

When the C file above is compiled, the code generated from `func1` will be stored in its own separate object section of the of the `.DOJ` file named `ext_code`. The `temp` variable will reside separately in `ext_data`.

So what happens to `func2`? If an object section name is not specified, the compiler will use a default name. In this case, the object file would hold the code from `func2` in an object section for program code with a default object section name of `seg_pmco`.

For the list of default object section names, refer to the *VisualDSP++ C/C++ Compiler and Library Manual for SHARC Processors*.

There are no default object section names for assembly source files, and `.SECTION/` statements must be used.

The way that these object sections and object section names relate to code placement in memory will be illustrated later.

Executable Files

Once the source files have been assembled and/or compiled into their respective *object files*, the linker combines these object files into one integrated *executable*, which has a `.DXE` extension.

Similar to object files, an executable is split into different sections. All `.DXE` files produced by the VisualDSP++ linker adhere to the splitting rules as dictated by the ELF (Executable and Linking Format) file standard. These sections are referred to as *DXE sections*, and have *DXE section names*.

DXE section names are completely independent of object section names. They exist in different namespaces, which means that a *DXE section name* can be identical to an object section name. (Unfortunately, in most distributed examples, this is the case. Using the same name for different items, although valid, is poor programming style and makes the `.LDF` file look confusing.)

The `.DXE` file is *not* loaded into the processor, and it is *not* burned into an EPROM. A `.DXE` file contains extra information (in addition to the raw code and data from the object files), which is used by down-stream tools such as the loader (to populate an EPROM) and the debugger (to simulate or emulate a processor) in locating code to the processor.

.LDF Files vs. the Other Files

The linker's job is reasonably straightforward. Based on commands in the `.LDF` file, the linker collects the object sections from the object files and combines them into a single executable

(.DXE) file using the memory model declared in the .LDF file.

Let's look at an example .LDF file to understand how this process works. We will examine an .LDF file used to link assembly-only source code. This is the simplest type of .LDF file, because C source code must be supported in the .LDF file with additional information about the C run-time header and other library information.

The following section of this document discusses test.ldf (Listing 1). Each section below includes a number that matches it to a corresponding portion of the .LDF file, identified with [#:] adjacent to the .LDF feature being described.

[1:] Memory{}

The first thing of interest in the .LDF file is the Memory{} command, which defines the memory model. It informs the linker as to the available *memory spaces* and gives them names called *memory space names*. It's important to understand that the memory space names declared here have absolutely no relation to *DXE* object *section names*. The next notable feature of the .LDF file is used to bind (link) all of these various sections and name spaces together.

[2:] Sections{}

As mentioned, the second and perhaps most important piece of the .LDF file is the Sections{} portion, where the linker does the real work. Based on three arguments to the Sections{} command, the linker takes object sections as inputs, places them in a *DXE section*, and then maps each *DXE section* to the specified memory space.

In test.ldf, for example, the first line in the Sections{} command takes the object section named seg_rth created in main.doj, places it in the DXE section named seg_rth, and maps that DXE section to the seg_rth memory space.

When the .DXE file created by the linker is loaded into the debugger or made ready for EPROM via the loader, each of these downstream utilities will know where the different DXE sections reside on-chip.

[3:] $OBJECTS and $COMMAND_LINE_OBJECTS

A minor thing to note is that we have used $OBJECTS interchangeably with $COMMAND_LINE_OBJECTS. This is because the $ symbol in the .LDF file is a macro definition. It works much like a #define statement in C.

These macro definitions, though unnecessary in this example, are quite useful when you have multiple object files. For instance, there are two options if you want to place all of the code portions (object sections) of multiple files into the same memory section. The first way is to explicitly list each object file, as follows:

```
seg_pmco
{
INPUT_SECTIONS(
main.doj(seg_pmco)
config.doj(seg_pmco)
dsp.doj(seg_pmco)
)
}> seg_pmco
```

The second way is to define a macro (as follows)

```
$OBJECTS = main.doj, config.doj,
dsp.doj;
```

and place the macro in the INPUT_SECTIONS{} portion of the .LDF file as follows:

```
seg_pmco
{
INPUT_SECTIONS(
$OBJECTS (seg_pmco)
)
} > seg_pmco
```

Both of these syntaxes are equivalent. They cause the linker to go through all of the object files listed, and if any object sections named seg_pmco are found, it puts them into the same DXE section (here "seg_pmco"), and then links that DXE section to the memory space called `seg_pmco`.

Each processor in a multiprocessor system would have its own `Processor` section (processor p0, p1, and so on). Each processor would also have its own `Sections{}` declaration.

[4:] LINK_AGAINST()

The `LINK_AGAINST()`command is used by the linker in a multiprocessor situation. It is used when there is a shared memory resource that holds code and/or data. The use of this command is beyond the scope of this document, but is discussed in *VisualDSP++ Linker and Utilities Manual.*

[5:] OUTPUT()

The final thing of interest in the `.LDF` file is the `OUTPUT()` command. This specifies the name of the produced `.DXE` file. If the `$COMMAND_LINE_OUTPUT_FILE` macro is used as an argument to the `OUTPUT()` command; the linker will name the `.DXE` file based on the VisualDSP++ project name**.**

Alternatively, you can enter the explicit filename: `OUTPUT(myfile.dxe)`.

```
ARCHITECTURE(ADSP-21368)
SEARCH_DIR( $ADI_DSP/213xx/lib )

[3:] $OBJECTS = $COMMAND_LINE_OBJECTS;

[1:] MEMORY
{
  seg_rth   { TYPE(PM RAM) START(0x00090000) END(0x000900ff) WIDTH(48)  }
  seg_init  { TYPE(PM RAM) START(0x00090100) END(0x000901ff) WIDTH(48)  }
  seg_int_code { TYPE(PM RAM) START(0x00090200) END(0x000902CF) WIDTH(48)  }
  seg_pmco  { TYPE(PM RAM) START(0x000902D0) END(0x00093FFF) WIDTH(48)  }
  seg_pmda  { TYPE(PM RAM) START(0x000C0000) END(0x000c1FFF) WIDTH(32)  }
  seg_dmda  { TYPE(DM RAM) START(0x000b8000) END(0x000bbfff) WIDTH(32)  }
  seg_stak  { TYPE(DM RAM) START(0x000bc000) END(0x000bdfff) WIDTH(32)  }
  seg_heap  { TYPE(DM RAM) START(0x000e0000) END(0x000e1fff) WIDTH(32)  }
  seg_sram  { TYPE(DM RAM) START(0x00200000) END(0x0027ffff) WIDTH(8)  }
}
PROCESSOR p0
{
   [4:] LINK_AGAINST( $COMMAND_LINE_OUTPUT_DIRECTORY\P1.dxe)
   [5:] OUTPUT($COMMAND_LINE_OUTPUT_DIRECTORY\P0.dxe )

   [2:]  SECTIONS
     {
        // .text output section
        seg_rth {
               INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
        } > seg_rth

        seg_init {
            ldf_seginit_space = . ;
            INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
        } > seg_init

        seg_int_code {
            INPUT_SECTIONS( $OBJECTS(seg_int_code) $LIBRARIES(seg_int_code))
```

```
        } > seg_int_code

        seg_pmco {
            INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
        } > seg_pmco

        seg_pmda {
            INPUT_SECTIONS( $OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
        } > seg_pmda

        .bss ZERO_INIT {
           INPUT_SECTIONS( $OBJECTS(.bss) $LIBRARIES(.bss))
        } > seg_dmda

        seg_dmda {
            INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
        } > seg_dmda

        Stackseg {
            // allocate a stack for the application
            ldf_stack_space = .;
            ldf_stack_length = MEMORY_SIZEOF(seg_stak);
        } > seg_stak

        heap {
            // allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_length = MEMORY_SIZEOF(seg_heap);
            ldf_heap_end = ldf_heap_space + ldf_heap_length - 1;
        } > seg_heap

        seg_sram {
            INPUT_SECTIONS($OBJECTS(seg_sram) $LIBRARIES(seg_sram))
            PACKING(5   B0 B0 B0 B4 B0
                        B0 B0 B0 B3 B0
                        B0 B0 B0 B2 B0
                        B0 B0 B0 B1 B0)
        } > seg_sram
    }
}
```

Listing 1. test.ldf

Expert Linker Utility

The Expert Linker utility available with the latest VisualDSP++ tools simplifies the process of the `.LDF` file generation. The Expert Linker is a graphical tool that simplifies complex tasks such as memory-map manipulation, code and data placement, overlay and shared memory creation, and C stack/heap adjustment. It enables new users to take immediate advantage of the powerful `.LDF` format flexibility by providing visualization capability.

Expert Linker allows users to:

- Define a target processor's memory map
- Place a project's object sections into that memory map
- View how much of the stack or heap has been used after running the DSP program

The `.LDF` file for the user application can be created by selecting the following menu item from VisualDSP++:

Tools -> Expert Linker -> Create LDF

Figure 1 shows the `.LDF` file generated using the Expert Linker.

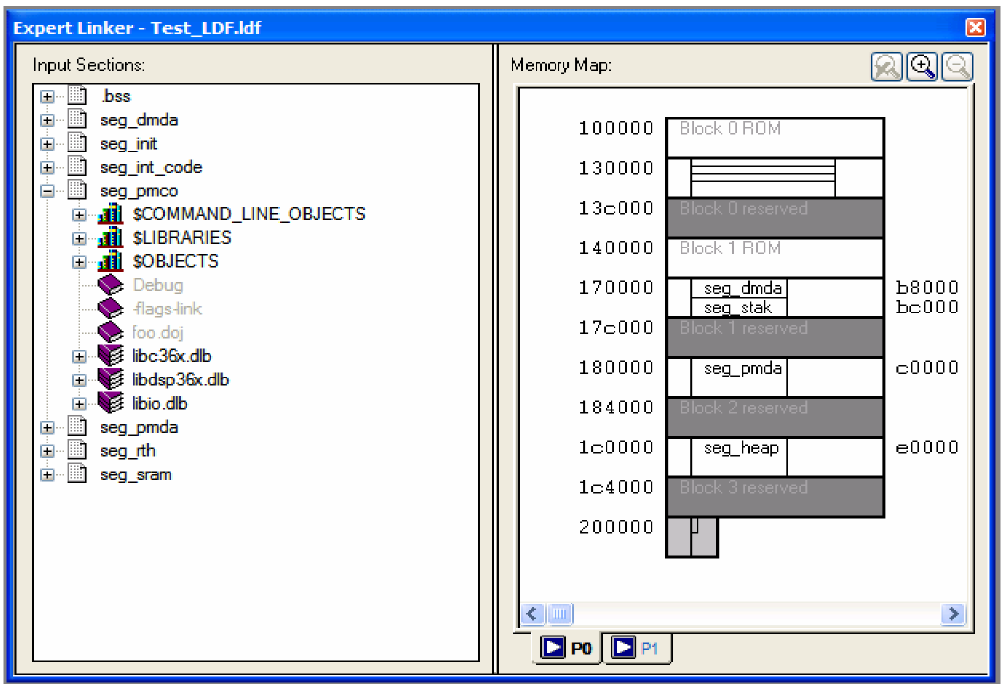

Figure 1. Expert Linker GUI.

Use the Expert Linker to drag and drop the object sections from the input section to the output memory section. After running the application, it shows the amount of the heap memory and stack memory utilized by the user application. This information can be used to adjust the heap and stack memory size. The unused memory sections can be used for placing code or data, providing efficient use of the processor's memory. For details on Expert Linker features, refer to the *VisualDSP++ Linker and Utilities manual* and to online Help.

The Expert Linker supports `.LDF` file generation for both single-processor and multiprocessor systems. See *Using the Expert Linker for Multiprocessor LDFs (EE-202)* for details.

Summary

This EE-Note discusses the various sections and commands used in the `.LDF` file. It provides an overview of the Expert Linker utility in VisualDSP++. It also explains how code and data can be placed in different memory sections of the user application.

The example code associated with this EE-Note demonstrates how code can be located in specific memory sections either on a per-object file basis or via the `SECTION ()` directive shown above.

References

[1] *VisualDSP++ 4.5 Linker and Utilities Manual,* Revision 2.0, April 2006. Analog Devices, Inc.

[2] *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for SHARC Processors,* Revision 6.0, April 2006. Analog Devices, Inc.

[3] *Using the Expert Linker for Multiprocessor LDFs (EE-202),* Revision 2.0, September 2004, Analog Devices Inc.

Document History

Revision	**Description**
Rev 2 – January 17, 2007 by C. Prabakaran and Jeyanthi Jegadeesan	Generalized the EE-Note for all SHARC Processors.
Rev 1 – January 5, 2002 by Matt Walsh	Initial Release.