# Engineer To Engineer Note EE-69

Notes on using Analog Devices' DSP's, audio, & video components from the Computer Products Division
Phone: (800) ANALOG-D , FAX: (781) 461-3010, EMAIL: dsp.support@analog.com

## Understanding and Using Linker Description Files (LDFs)

There are a number of cases in which a programmer would like to control where a particular piece of code or data resides in a 2106x's memory space.

For example, on the 21065L it is possible to execute from external memory, however this execution is performed at a reduced rate. In this case, the programmer will want to store and execute frequently used code internally and rarely used code externally.

Regardless of if one wishes to relocate a C-function or an assembly routine, the mechanism is the same. To map portions of code or data to specific memory segments, it is necessary to use the Linker Description File (referred to as the LDF).

This application note will explain the functionality of the LDF and demonstrate it's 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.

---

The first step towards gaining an understanding of the LDF is to first understand the makeup of the files involved in building a DSP executable.

### - Source Files -

At the beginning of it all, we have the source files. These files contain code written in either C or Assembly. The first step towards producing an executable is to compile and/or assemble these sources. The assembler outputs files called "object files". (The compiler outputs assembly files that are then fed to the assembler.) VisualDSP produces object files that have the extension `.DOJ`. (Note: these object files are typically output to the `~/YourProject/debug` directory.)

### - Object Files -

The object files produced by the compiler and assembler are divided up into various "sections" or "segments" (referred to as *Object Segments*). Each of these *Object Segments* is used to hold a particular type of compiled source code. For example, such an *Object Segment* may hold program op-codes (48-bits wide) or data such as variables (16, 32 or 40-bits wide). Some of these *Object Segments* also hold information not pertinent to this discussion (since the information isn't important to the user - such as debug-information, etc.).

Each of these different *Object Segments* has a different name that is specified in the source code. Depending on whether the source is C or assembly, there is a different convention for specifying this *Object Segment Name*.

In an **assembly** source, the code and/or data is placed between the "`.segment SegName`" and the "`.endseg`" statements will then be found in the *Object Segment* called "`SegName`". Here is an example:

```
.segment /dm asmdata
      .var foo[3];
.endseg

.segment /pm asmcode
      r0 = 0x1234;
      r1 = 0x4567;
      r2 = r1 + r2;
.endsegment
```

Here, the *Object Segment* "`asmdata`" would contain the array foo, and the 3-lines of code would be located in the *Object Segment* "`asmcode`".

In a **C** source the programmer would use the `segment("seg_name")` directive. Here is an example:

```
segment("ext_data") int temp;
segment("extern") void func1(void)
{
      int x = 1;
}

void func2(void)
{
      int i = 0;
}
```

When this C-file is compiled, the code generated from `func1` will be stored in its own separate *Object Segment* of the of the .DOJ file named "extern". The variable "temp" will reside seperately in the "`ext_data`".

So what happens to `func2`? If an *Object segment Name* isn't specified, the compiler will use a set of default names. So in this case, the object file would hold the code from `func2` in an *Object Segment* for program code which happens to have a default *Object Segment Name* of "`seg_pmco`".

Page 2-60 of the "C Compiler Guide & Reference for the ADSP-2106x Family DSPs" lists the default *Object Segment Names*.

(Note: there are no default *Object Segment Names* for assembly source files, and the `.segment/ .endseg` statements MUST be used.)

How these *Object Segments* and *Object Segment Names* relate to code placement in memory will be illustrated later.

**- Executable Files -**

Once all of the source files have been assembled and/or compiled into their respective *object files*, it is then the job of the linker to combine all of these *object files* into one integrated *executable* which has the extension .DXE.

Just like the *object file*, the *executable* is broken up into different "segments". (As dictated by the ELF [Executable and Linking Format] file standard that's conformed to by the .DXEs produced by Visual DSP.) These segments are referred to as *DXE Segments*, and as you may suspect, the segments have *DXE Segment Names*.

It is absolutely crucial to understand that these ***DXE*** *Segment Names* are completely independent of the ***Object*** *Segment Names*. They exist in different name-spaces, which means that you can actually have a ***DXE*** *Segment Name* that is exactly the same as an ***Object*** *Segment Name*. (Unfortunately, in most distributed examples, this is the case. This use of the same name for different items, although legal, is poor programming style and makes for a confusing LDF).

It is also important to understand the function of the .DXE. The .DXE is ***not*** what is loaded into the DSP, and it is ***not*** what is burned into an EPROM. This is because the DXE contains "extra" information (in addition to the raw code and data from the object files.) This data is used by "down stream tools" such as the Loader (used to create an EPROM) and the Debugger (used for simulation/emulation), to properly locate code in the DSP.

## So How Does the LDF Relate to All of These Different Files?

The linker's job in all of this is reasonably straightforward. Based on commands in the LDF file, the linker takes all of the *Object Segments* from all the object files, and using the memory model declared in the LDF file, combines them all into a single executable .DXE file.

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

The remainder of this application-note will discuss **Code Listing 1** - `test.ldf`. Each numbered section below describes a feature in the LDF file. The section number corresponds to a number in the LDF file that we've added (using the [#:] format) beside the LDF feature in question.

1) The first thing of interest in the LDF is the `Memory{}` command. This defines the memory model. It tells the linker what *Memory Spaces* are available and gives them names called *Memory Space*

*Names*. ("Great! an*other* name-space!", right?) Again, it's important to understand that the *Memory Space Names* declared here have absolutely no relation to either the ***DXE*** *Segment Names* or the ***Object*** *Segment Names*. The next notable feature of the LDF is used to bind, or "link", all of these various segments and name-spaces together.

2) As just mentioned, the second and perhaps most important piece of the LDF file is the "`Sections()`" portion. It is here where the linker does the real work. Based on three arguments to the `Sections()` command, the Linker takes ***Object*** *Segments* as inputs, places them in a ***DXE*** *Segment*, and then maps each of those ***DXE*** *Segments* to the specified *Memory Space*.

In `test.ldf`, for example, the first line in the `Sections()` command below takes the ***Object*** *Segment* named "`isr_tbl`" that we created in `main.doj`, places it in the ***DXE*** *Segment* which we've named "`dxe_isr`", and finally maps that DXE Segment to the "`mem_isr`" *Memory Space*. So when the .DXE that the Linker creates is loaded into the debugger or made ready for EPROM (via the Loader), each of these "down-stream" utilities will know where the different *DXE Segments* should reside on-chip.

3) A minor thing to take note of is the fact that we've used "`$OBJECT1`" interchangeably with "`main.doj`". This is because the "`$`" symbol in the LDF file is a macro definition. It works much like the "`#define`" statement does in C.

These macro definitions, though unnecessary in this example, are quite useful when one has multiple object files. For instance, if one would like to place all of the **code** portions *(Object Segments)* of multiple files in the same memory segment, there are two options. The first way is to go through and explicitly list each object file:

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

The second way is to define a macro as follows:

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

and then in the `Sections()`portion of the LDF :

```
dxe_pmco{ INPUT_SECTIONS($ALL(seg_pmco)) } > mem_pmco
```

Both of these syntaxes are exactly equivalent. The would cause the linker to go through all of the object files listed, and if it found any *Object Segments* named "`seg_pmco`", it would put them in the same *DXE Segment* (here "dxe_pmco"), and then link that *DXE Segment* to the *Memory Space* called "`mem_pmco`".

(Note: each processor in a MP system would have it's own "`Processor`" section - Processor p0, p1, etc. Each processor would then have its own "`Sections()`" declaration.)

```
ARCHITECTURE(ADSP-21065L)
SEARCH_DIR( $ADI_DSP\21k\lib )
[3:]$OBJECT1 = main.doj;
[1:]MEMORY
  {     mem_isr { TYPE(PM RAM) START(0x00008000) END(0x000080ff) WIDTH(48) }
       mem_pmco { TYPE(PM RAM) START(0x00008100) END(0x000087ff) WIDTH(48) }
       mem_pmda { TYPE(PM RAM) START(0x00009000) END(0x00009fff) WIDTH(32) }
       mem_dmda { TYPE(DM RAM) START(0x0000c000) END(0x0000dfff) WIDTH(32) }
  }
PROCESSOR p0
  {
[4:] LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
[5:] OUTPUT( $COMMAND_LINE_OUTPUT_FILE )
[2:] SECTIONS
    {   dxe_isr { INPUT_SECTIONS($OBJECT1(isr_tbl))  }  > mem_isr
       dxe_pmco { INPUT_SECTIONS($OBJECT1(seg_pmco)) }  > mem_pmco
       dxe_pmda { INPUT_SECTIONS(main.doj(seg_pmda)) }  > mem_pmda
       dxe_dmda { INPUT_SECTIONS(main.doj(seg_dmda)) }  > mem_dmda
    }
  }
```

Code Listing. 1 – test.ldf

4) The `LINK_AGAINST()`command is used by the linker in Multi-Processor situations. 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 on page 5-31 of the *VisualDSP User's Guide and Reference.*

5) The final thing of interest in the LDF is the `OUTPUT()` command. This simply specifies the name of the .DXE file produced. If the macro `$COMMAND_LINE_OUTPUT_FILE` is used as an argument to the `OUTPUT()` command; then the linker will name the .DXE based on the project name from the IDE. Alternatively you can simply enter the explicit filename: `OUTPUT(myfile.dxe).`

**- Attachment -**

The attached zip file contains an example that uses C-source files. It has three functions located in two files and demonstrates how code can be located in specific memory segments either on a per-object file basis or via the `SEGMENT()` directive shown above.

For more information contact Analog Devices' DSP Applications Group:

http://www.analog.com/dsp/EZ/

dsp.support@analog.com

1-800-Analog-D.