Modifying LDF files in CrossCore Embedded Studio

Introduction

When facing Linker errors, such as those described in the Common Errors section of this document, the solution often requires knowledge of modifying the LDF to control the link and exercise control of the linking process. While not exhaustive, this section will detail the essentials for modifying LDFs, as well as using the Project Options and Compiler Pragmas to assist the Linker.

Blackfin and the System Builder

In CrossCore Embedded Studio, both the Blackfin and SHARC processors use generated LDFs. The addition of an LDF is done either through the New Project Wizard, or by adding the "Startup Code/LDF" Add-In via the System Configuration dialog (which can be accessed by double-clicking the *.svc file in the Project Explorer Window). There are some features of this Add-In which need to be considered when trying to implement changes to the LDF.

The first consideration is that the LDF section of the Startup Code/LDF Add-In offers a number of options which can be changes to control the way the System Builder auto-generates the LDF. The benefit of this option is that it does not require as much knowledge of the LDF commands. The obvious drawback is that there is less flexibility offered by these project options than hand-modifying your LDF.

Modifying the MEMORY{…} Map

The LDF contains a MEMORY{…} block which defines the various memory ranges on the processor, so the Linker knows how to put together the executable. The default LDFs included in CrossCore Embedded Studio list, in comments, all of the valid memory ranges for the processor. Then within the MEMORY{…} block they define each memory range. E.g.

```
MEMORY
{
  MEM_L1_SCRATCH    { START(0xFFB00000) END(0xFFB00FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_CODE_CACHE { START(0xFFA10000) END(0xFFA13FFF) TYPE(RAM) WIDTH(8) }
  MEM_L1_CODE       { START(0xFFA00000) END(0xFFA0BFFF) TYPE(RAM) WIDTH(8) }
  ...
}
```

For Blackfin the memory map is straight forward, as there is only one valid WIDTH; 8. With the SHARC processors, however, a memory section can be a variety of widths; 8, 16, 32, 48 or 64 bits. When modifying, or defining new sections, you need to check that the memory range - START to END - is valid for the WIDTH specified. Also note that the WIDTH for memory connected to the external memory interface must be the physical width of that memory or bus. E.g. the 21369 External Memory interface is 32-bits, so the width *must* be 32.

Another consideration needs to be made here for System Builder generated LDFs. These will overwrite any changes you make to the memory map when they are regenerated. You can add new sections by inserting new definitions in the appropriate $VDSG section. Additionally, for the SDRAM memory range you can enable an option to use a Custom Partition for external memory via the LDF section of the Startup Code/LDF Add-In in the System Configuration dialog. This will wrap the SDRAM definitions in the MEMORY{…} block in $VDSG tags, allowing customization of the memory ranges.

Macros

The default LDFs make use of macros to provide a convenient way to reference groups of objects in the link, and control their placement as a group rather than having to individually reference each one. The two basic ones (though there are variations on these, depending on your target) are $OBJECTS and $LIBRARIES.

The $LIBRARIES macro provides a collective reference to all of the objects within the .dlb library files being linked into the project, while the $OBJECTS macro provides a collective reference to the .doj object files being produced by the Compile/Assemble processing of the source files in your project.

For the most part, these two macros will be all that is needed. However you may feel a need, for example, to single out objects/libraries as higher priority, or simply to distinguish them from the $OBJECTS and $LIBRARIES default collections. To do this you can simply define your own macros and tell them what objects/libraries you want them to encompass. E.g.:

```
$MY_OBJS_AND_LIBS = main.doj, afunction.doj, mylibrary.dlb;
```

This macro can later be referenced using an INPUT_SECTIONS command within the SECTIONS{...} block to place them in a specific memory section (more on this later):

```
aSection
{
  INPUT_SECTIONS($MY_OBJS_AND_LIBS(program))
} > MEM_L1_CODE
```

Modifying the SECTIONS{…} Block

Controlling the placement of any object ultimately comes down to the LDF configuration. The LDF is full of output section commands that tell the linker which objects (and what part of those objects) to group together for placement in a section of memory.

It is important to remember that all commands in the LDF are processed in the order they appear in the LDF. See 'Prioritising Output Section Commands' below for more information.

An output section command takes the following basic form:

```
output_section_name
{
  section_commands
  //e.g.:
  INPUT_SECTIONS($OBJECTS(input_section_name))
} > memory_segment
```

There are additional options that can control the memory type and initialization (zero-init, no-init, etc), but these are not within the scope of this document. The output section commands are discussed in detail in the Linker and Utilities Manual.

The output_section_name is just a name that helps the Linker create distinct groups of objects based on the following section_commands so that it can allocate them space, and later place the objects, in a memory_segment, as defined in the MEMORY{...} block of the LDF. This name is arbitrary, though it must be unique within the scope of its PROCESSOR{…} block within the LDF.

A basic example of a complete output section command would be as follows:

```
an_output_section
{
  INPUT_SECTION_ALIGN(4)
  INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
} > MEM_L1_CODE
```

In the above example we have an arbitrary (but unique) name for our output section, ‘an_output_section’, which maps to a memory section called ‘MEM_L1_CODE’ which is (we can assume for the sake of this example) defined in our MEMORY{...} map. Within the output section command we have some section commands:

Firstly we have an INPUT_SECTION_ALIGN(4) command, which tells this linker that all objects must be aligned to a 4-word boundary. E.g. if an object is 3words, the next word would be skipped, and the next object would start after that, meaning that everything takes up a multiple of 4-bytes.

Next up is the most important command for controlling the object placement, the INPUT_SECTIONS command. This is discussed in detail next but in simple terms the example above extracts the ‘program’ sections from any objects that are part of the $OBJECTS and $LIBRARIES macros, and links them into the specified memory section, MEM_L1_CODE.

INPUT_SECTIONS Commands and section/default_section Pragmas

All object files (.doj) contain ‘sections’. There are default section names for the different data types. E.g. data is placed in ‘data1’, code is placed in ‘program’, zero-initialized data is placed in ‘bsz’, etc. You can also define your own section names, which is extremely useful for controlling the placement of the objects.

The recommended way for doing this is through the use of the “section” and “default_section” pragmas in C/C++ source files, and the “.section” keyword in Assembly files.

For example, if your source file contains multiple functions and you want to separate one of your functions from the others in the default “program” section, so that later you have finer control over its placement, you can do so using these pragmas/keywords.

Pragmas only affect the Global Symbol that immediately follows their use (this is important to remember; you cannot control the placement of local, or “automatic”, variables within a function as these are allocated on the heap, rather than placed by the linker). This Global Symbol could be some global data, such as a buffer, or a function. For example:

```
#pragma section("MyData")
int myFirstLargeArray[10000];
int mySecondLargeArray[10000];

#pragma section("MyCode")
void myFunction
{
  ...
}
```

In the example above, "myFirstLargeArray" will be placed in section "MyData", while "mySecondLargeArray" will be placed in the default section (e.g. "data1") as the pragma's scope has ended. The function myFunction gets placed in the section "MyCode" by the next section pragma.

The 'default_section' pragma can be used to control the section name over a larger scope. The maximum this 'scope' can cover is until the end of the current source file, when the Compiler will revert to the default sections. For example:

```
#pragma default_section(DATA, "MyData")
int myFirstLargeArray[10000];
int mySecondLargeArray[10000];

#pragma default_section(DATA, "data1")
int myThirdLargeArray[10000];
```

In the case above, we change the section name for all "DATA" to "MyData". So, the first two arrays will be given this section name. We are resetting the default section for DATA back to "data1" for myThirdLargeArray, and it will remain that way until we either change it back/again using the default_section pragma, or until the end of the file.

After understanding how the Compiler/Assembler assigns sections to objects, and how the Linker processes the output section commands to extract these sections from objects before placing them into a memory section, the next stage is tying these together in order to control the placement of your objects.

The most basic way to place an object is using the section pragmas/keyword, so you can place an object using the existing commands in the LDF. Say you have an array you want to place in SDRAM on a BF537 target. Taking a look at the BF537 LDF you will see a command such as:

```
sdram0_bank1
{
  INPUT_SECTION_ALIGN(4)
  INPUT_SECTIONS( $OBJECTS(sdram0_data) $LIBRARIES(sdram0_data))
  ...
} > MEM_SDRAM0_BANK1
```

This places any objects with a section name “sdram0_data” into MEM_SDRAM0_BANK1, which is what we want to do. To place your array in this section you can use the section pragma as follows:

```
#pragma section(“sdram0_data”)
int myLargeArrayInSDRAM[10000];
```

You could also use the section pragma to define a custom section name, and then add your own INPUT_SECTIONS command in the LDF to place your objects. E.g. In your source file:

```
#pragma section(“myCustomData”)
int myLargeArrayInSDRAM[10000];
```

Then in your LDF, just add an INPUT_SECTIONS command to the existing sdram0_bank1 output section command:

```
sdram0_bank1
{
  INPUT_SECTION_ALIGN(4)
  INPUT_SECTIONS( $OBJECTS(sdram0_data) $LIBRARIES(sdram0_data))
  INPUT_SECTIONS( $OBJECTS(myCustomData))
  ...
} > MEM_SDRAM0_BANK1
```

In a more advanced scenario, you may have a requirement that your objects are mapped separately from any others, perhaps to control the priority with which they are placed in memory. In this scenario you simply need to use the custom section name as above, and then add your own output section command to the LDF:

```
my_sdram_data
{
  INPUT_SECTION_ALIGN(4)
  INPUT_SECTIONS( $OBJECTS(myCustomData))
  ...
} > MEM_SDRAM0_BANK1
```

INPUT_SECTIONS commands can be used to directly reference an object in the link, to control the placement of the sections within. For example, assume we have a “main.c” file containing various “program” and “data1” sections, which creates an object called main.doj. We can place the sections within this object wherever we want by using INPUT_SECTIONS commands:

```
L1_code
{
  INPUT_SECTIONS(main.doj(program))
} >MEM_L1_CODE
```

```
L1_data_a
{
  INPUT_SECTIONS(main.doj(data1))
}
```

Similarly, you can place a section, such as “program” from all objects within a library by referring to the library only. This will extract the “program” section from every object in the library and place it in this section.

```
INPUT_SECTIONS(libc532y.dlb(program))
```

With more advanced INPUT_SECTIONS commands you can even address an object from within a library, and place the sections from within that object. This can be done with a command of the form:

```
INPUT_SECTIONS( library [ object.doj (section) ] )
```

Note: The spacing of these brackets is very important, and will lead to a Linker error if used incorrectly.

So, to place the “program” sections of the sprintf32 object from within the libc532y.dlb you would use the following command

```
INPUT_SECTIONS( libc532y.dlb [ sprintf32.doj (program) ] )
```

Prioritising Output Section Commands

The LDF, as already mentioned, is processed by the Linker in order. That is, whatever command appears first, get processed first. Whether it is one output section command followed by another, or even the objects specified in the INPUT_SECTIONS command, they will be processed in order.

What this means is, in a typical example where an INPUT_SECTIONS command references both the $OBJECTS and the $LIBRARIES macros, the $OBJECTS get priority as they appear first:

```
INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program))
```

When modifying the default LDF, you need to be aware of possible commands that will map your objects somewhere you don’t want them to be, and ensure that your custom commands precede them.

The default flow of the LDF is that it fills L1 memory, then L2, then SDRAM. So, anything that doesn’t fit in L1 will overflow to L2, and so on into SDRAM. Because all objects are included in the $OBJECTS macro, this means that you need to heed the fact that there may be a $OBJECTS(program) command placing objects in L1 memory that you want placed in SDRAM.

If you simply add an INPUT_SECTIONS command to the default SDRAM mappings in the LDF to try and place the "program" section of an object into SDRAM, there is a good chance that the one of the default INPUT_SECTIONS($OBJECTS(program) $LIBRARIES(program)) commands will get there first, and your command will not make any difference.

This can be easily avoided by using custom output section commands at the start of the SECTIONS{...} block of the LDF - before any of the default LDF commands - so that your custom commands get priority. For generated LDFs, use the first set of $VDSG tags in the SECTIONS{…} block.

RESOLVE LDF Command

The LDF offers another command that can be used to control the placement of symbols; the RESOLVE() command.

The RESOLVE command allows you to specify a symbol name, and a 'resolver'; typically an address. This is used in the default Blackfin LDFs to place the 'start' symbol at the first address of L1 Code memory as follows:

```
RESOLVE(start, 0xFFA00000)
```

The same principle can be applied to tell the Linker to try to place certain symbols, like a specific function, at an exact address. It is important to be aware of the required alignment, and ensure you do not place any code/data at a misaligned address as this could result in exceptions in the application.

Mutiple DXEs/output files for multicore targets

Instructing the Linker to produce multiple DXEs is, at the most basic level, a matter of modifying the LDF using the above techniques to control which objects are linked in for each DXE. So, producing multiple DXEs requires that you define a PROCESSOR{…} block for each DXE you want to create.

To create multiple DXEs from a single project (though this project's Project Group may include projects which produce library files), you need to ensure that the LDF has a PROCESSOR{…} block for each DXE. Within this PROCESSOR{…} block you need to define an output filename using the OUTPUT(*filename.dxe*) command.

Finally, all you need to do is create a SECTIONS{…} block, and make use of custom macros, INPUT_SECTIONS commands, etc, to control what objects from within the project's linking scope (this includes libraries being linked in, etc) should be linked into each of the DXEs. The BF561 EZ-KIT Lite and BF609 EZ-KIT Lite projects are great references for this, as these are currently our only multiprocessor targets for CrossCore Embedded Studio.