Introduction

This article explains the code which typically appears in most TI linker command files. It does not explain everything. It does not apply to GCC linker scripts.

Problem Statement

You already have a linker command file. It works well on the development system (kit, launchpad, EVM, etc.) you started on. But now it is time to get everything running on the final production system. Among other things, you need to change the linker command file to model the configuration of memory on the production system. This article helps you do that by explaining the linker command file you have now.

Article Overview

The Basics section is for everyone. The code described in the Basics section appears in every linker command file. The Barely Beyond Basics section is where you can be more selective. That is where you find the description of code which appears in some, but not all, linker command files.

Basics

Linker command files can contain anything that might appear on the command line: options, object file names, library names. Global symbols can be created. But none of that is described in this article. This section focuses on the MEMORY directive and especially the SECTIONS directive. Those directives appear in every linker command file.

The MEMORY Directive

The purpose of the MEMORY directive is to assign names to ranges of memory. These memory range names are used in the SECTIONS directive. Here is part of the MEMORY directive from a typical MSP430 system ...

```
MEMORY
{
    SFR                     : origin = 0x0000, length = 0x0010
    PERIPHERALS_8BIT        : origin = 0x0010, length = 0x00F0
    PERIPHERALS_16BIT       : origin = 0x0100, length = 0x0100
    RAM                     : origin = 0x1C00, length = 0x0FFE
    INFOA                   : origin = 0x1980, length = 0x0080
    INFOB                   : origin = 0x1900, length = 0x0080
    INFOC                   : origin = 0x1880, length = 0x0080
    INFOD                   : origin = 0x1800, length = 0x0080
    FLASH                   : origin = 0x8000, length = 0x7F80
    INT00                   : origin = 0xFF80, length = 0x0002
    /* ... and so on */
}
```

The line that begins with RAM defines a memory range named RAM. It starts at address 0x1C00 and has a length of 0xFFE.

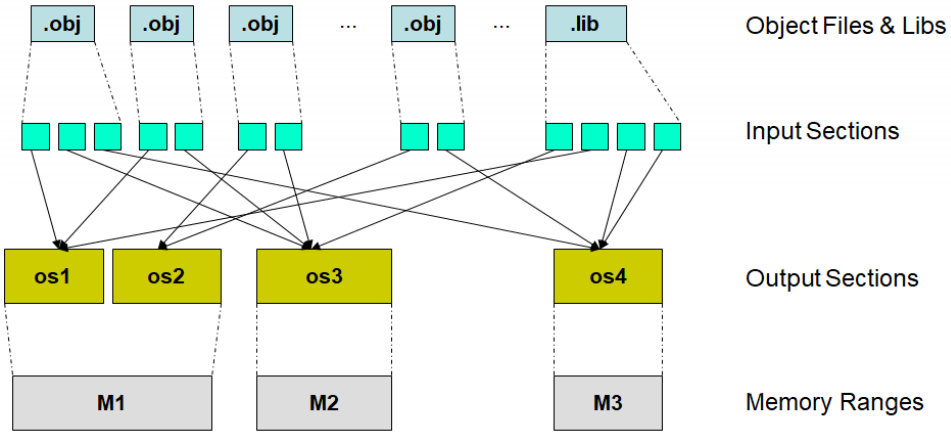
The SECTIONS Directive

The SECTIONS directive contains most of the interesting code. The key thing to understand is that the SECTIONS directives does two things at once.

- It forms output sections from input sections
- It allocates those output sections to memory

Diagram

Here is a graphic way of showing how the SECTIONS directive works.



Glossary

Describing the SECTIONS directive requires an understanding of these terms.

- Object file - For the purposes of this article, an object file is a collection of input sections. An object file can be presented directly to the linker (via the command line or in a command file), or it can come from a library.
- Input Section - One section from one object file. An input section can

be initialized or uninitialized. It can contain code or data.
- Output Section - A collection of one or more input sections. Output sections are formed according to the SECTIONS directive, with extremely rare exceptions not described in this article.
- Memory Range - A range of system memory specified in the MEMORY directive

Section Naming Conventions

In theory, you can't know anything about the contents of an input section based on the name alone. Nonetheless, input sections with these names usually have these contents:

Name	Initialized	Notes
.text	yes	executable code
.bss	no	global variables
.cinit	yes	tables which initialize global variables
.data (EABI)	yes and no	initialized coming out of the assembler; changed to uninitialized by the linker
.data (COFF ABI)	yes	initialized data
.stack	no	system stack
.heap or .sysmem	no	malloc heap
.const	yes	initialized global variables
.switch	yes	jump tables for certain switch statements
.init_array or .pinit	yes	Table of C++ constructors called at startup
.cio	no	Buffer for stdio functions

Sections names often begin with a '.', but that is not required.

You may see variations on these names like .ebss, or .fardata. Such sections are very nearly the same as the one described in this table.

Syntax Note

All of the examples in this part of the article occur within a SECTIONS directive.

```
SECTIONS
{
   /* all examples appear here */
}
```

Form Output Sections

One of the most confusing aspects of SECTIONS directive code is all the syntax shortcuts. To demystify all of that, here is an example which takes no shortcuts.

```
    output_section_name         /* Name the output section        */
    {
       file1.obj(.text)         /* List the input sections        */
       file2.obj(.text)
       file3.obj(.text)
    } > FLASH                   /* Allocate to FLASH memory range */
```

This creates an output section named output_section_name. It is composed of 3 input sections: .text from file1.obj, .text from file2.obj, and .text from file3.obj. It is allocated to the FLASH memory range.

Clearly, this syntax does not scale well to a system with many object files. So, here is one possible shortcut.

```
    output_section_name         /* Name the output section           */
    {
        /* Shortcut syntax for all input sections named .text        */
        *(.text)
    } > FLASH                   /* Allocate to FLASH memory range    */
```

This does the same thing as the previous example, with one difference. The previous example used exactly 3 input sections, and they are explicitly specified for both object file name and input section name. This example uses all the input sections named .text. To be precise, it uses all the input sections named .text which are not part of any other output section.

Because even that is not short enough, the shortcut in this example builds on the previous one.

```
    .text > FLASH
```

This example has only one difference from the previous example: the name of the output section has changed from output_section_name to .text. Note how the output section name and the input section names are all the same: .text. Despite that similarity, it is important to not overlook the distinction between input sections and the output section which contains them.

Here is another shortcut example:

```
    .text : {} > FLASH
```

This example is no different than the previous one. It is shown here because that syntax pattern is common in many linker command files.

You can mix these shortcuts together. For example:

```
    output_section_name
    {
        first.obj(.text)        /* This code must be first */
        *(.text)
    } > FLASH
```

This creates an output section named output_section_name. The first input section is the .text section from first.obj. The remaining input sections are all the .text sections from all the other object files. It is allocated to the FLASH memory range.

Allocate Output Sections to Memory

This syntax in the above examples ...

```
    ... > FLASH
```

allocates the output section to memory. The FLASH memory range is used in this specific case.

You may also see ...

```
    ... > 0x20000000
```

Allocations to a hard-coded address are always done before allocations to a named memory range. Your linker command file may take advantage of that ordering difference.

Another technique to watch for ...

```
#define BASE 0x20000000

/* many lines later */

    ... > BASE
```

This looks the same as an allocation to a named memory range, but it is actually an allocation to a hard-coded address.

Barely Beyond Basics

Starting at this point, the article becomes less comprehensive, and more selective. Look at the examples which accompany each section. If code like that appears in your linker command file, then that section describes it. Otherwise, you can ignore it.

Unless otherwise shown or stated, all of the examples occur within a SECTIONS directive.

Additions and Changes

If you see code in your linker command file which is not described in this article, please post to the E2E compiler forum (https://e2e.ti.com/support/development_tools/compiler/f/343) about it. The forum reply will usually result in an addition or change to this article.

First Output Section in a Memory Range

Suppose you see code similar to ...

```
#define BASE 0x00200000

MEMORY
{
    FLASH : origin = BASE, length = 0x0001FFD4
    ...
}

SECTIONS
{
    .intvecs > BASE    /* only section allocated to BASE */
    .text    > FLASH
    .const   > FLASH
    ...
}
```

The net effect of this code is that .intvecs is the first output section in the FLASH memory range. The remaining output sections also go in FLASH, but can be allocated in any order.

The #define BASE is an example of using the C-like preprocessor feature of the linker. It used to establish the beginning of the FLASH memory range. It also used to allocate .intvecs to that specific address. Allocations to a specific address are always done before allocations to a named memory range.

Allocate to Multiple Memory Ranges

Consider this example ...

```
    .text > FLASH0 | FLASH1
```

That means the .text output section is allocated to the memory range FLASH0 or FLASH1. FLASH0 is tried first. If it cannot contain all of .text, then FLASH1 is tried. Note .text is **not** split. The entire output section is placed in

either FLASH0 or FLASH1.

Split an Output Section Across Multiple Memory Ranges

Consider this example ...

```
    .text : >> RAMM0 | RAML0 | RAML1
```

That means .text is split across those memory ranges. Note the `>>` syntax. If all of .text does not fit in RAMM0, then it is split, and the rest goes into the remaining memory ranges. The split occurs on input section boundaries. An input section is never split. The memory ranges are used in that order.

Pages of Memory

Memory pages are used only in C28xx linker command files.

A memory page is specified in the MEMORY directive like this ...

```
MEMORY
{
    PAGE 0 :
        RAMM0   : origin = ...
        RAML0L1 : origin = ...

    PAGE 1 :
        RAMM1   : origin = ...
        RAML2   : origin = ...
}
```

Each page of memory is completely independent. Between pages you can reuse memory range names and memory addresses. The following example is completely legal, but a very bad idea ...

```
/* DO NOT DO THIS!!! */
MEMORY
{
    PAGE 0 :
        MEM_RANGE : origin = 0x100, length = 0x100
    PAGE 1 :
        MEM_RANGE : origin = 0x100, length = 0x100
}
```

The C28xx devices are descended from a long line of C2xxx devices which started in the 1980's. Those early devices had separate memory buses for code and data. These buses were connected to physically separate memory blocks. Thus, it actually was possible for a specific address on PAGE 0 to have different contents than that same address on PAGE 1. In theory, this same separate connection of memory buses is possible on C28xx devices, though it

is a rarely (bordering on never) used feature. If there is the slightest doubt, check the documentation specific to your device.

Even though nearly all C28xx devices have all the memory buses connected to all the memory, this tradition of using memory pages persists. If your linker command file uses PAGE 0 and PAGE 1, it is best to continue using it that way. Just be aware of the pitfalls illustrated by the last example.

Syntax Hint

Anywhere you can write MEMORY_RANGE_NAME you can also write MEMORY_RANGE_NAME PAGE 0. Ostensibly, this is because it is possible to have the same memory range name on multiple pages. Because that's bad programming practice, writing the page number is really about keeping everything clear and easy to maintain. If you combine a memory range name and page that don't exist, the linker will tell you.

Nullify an Output Section

If you see something like this ...

```
   .reset : > RESET, PAGE = 0, TYPE = DSECT    /* not used */
```

The syntax `TYPE = DSECT` makes this output section a dummy section. A dummy section takes up no space in memory, and is not present in the output file. The net effect is that all of the input sections named .reset are silently thrown away. That's OK in this specific case. But it is **not** OK for all cases. Suppose code in another section calls a function in .reset, or yet another section uses data in .reset. The linker silently swallows those references to .reset. You probably prefer a diagnostic.

For more on DSECT and other special sections like it, please see the article Linker Special Section Types (http://processors.wiki.ti.com/index.php/Linker_Special_Section_Types).

Refer to ROM Code or Data

This code ...

```
FPUmathTables : > FPUTABLES, PAGE = 0, TYPE = NOLOAD
```

is how to refer to a section that is already present in the system. That section is usually supplied in ROM or flash memory. The syntax `TYPE = NOLOAD` imparts the special noload properties. A noload section does take up space in memory, but it is not present in the output file. In practice, there are no references from inside a noload section to anything outside. But other sections outside the noload section can refer to code and data inside the noload section. In this specific case some floating point unit tables must be

supplied in ROM, and other code is using those tables.

For more on NOLOAD and other special sections like it, please see the article Linker Special Section Types (http://processors.wiki.ti.com/index.php/Linker_Special_Section_Types).

Load at One Address, Run from a Different Address

Consider this example:

```
ramfuncs : LOAD = FLASHD,
           RUN = RAML0,
           LOAD_START(_RamfuncsLoadStart),
           LOAD_END(_RamfuncsLoadEnd),
           RUN_START(_RamfuncsRunStart)
```

This creates an output section named ramfuncs. It is composed of all the input sections also named ramfuncs. It has two different allocations. It allocated to FLASHD for loading, and allocated to RAML0 for running. This output section is placed in the output file so that, when the program is loaded (which is probably implemented by programming it into flash memory), it is in the FLASHD memory range. Sometime during system execution, before anything in ramfuncs is used, the application copies it from FLASHD to RAML0. Note this copy is not done automatically. Explicit steps, not discussed here, must be taken in the application code. Any other section that uses ramfuncs (either calls its functions or refers to its data) acts as if ramfuncs is already in RAML0. The LOAD_START etc. statements establish symbols that are used to implement the copy. The value of the symbol _RamfuncsLoadStart is the starting load address. Likewise, _RamfuncsLoadEnd has the ending load address, and _RamfuncsRunStart has the starting run address.

Allocate a Single Input Section from a Library

Consider this example:

```
IQmathTables3 : > IQTABLES3
{
    IQmath.lib<IQNasinTable.obj> (IQmathTablesRam)
}
```

This forms an output section named IQmathTables3. It contains one input section named IQmathTablesRam. That input section comes from the object file IQNasinTable.obj, which is a member of the library IQmath.lib. This output section is allocated to the IQTABLES3 memory range.

A variation can be used to allocate all the sections from a library.

```
sinetext : > DDR2
{
     --library=Sinewave_lib.lib(.text)
}
```

This forms an output section named sinetext. It contains all the .text input sections from the files the linker used from the library Sinewave_lib.lib. It is allocated to the DDR2 memory range. Note the linker does not bring in all the files from Sinewave_lib.lib. It only brings in the files which are needed to satisfy open references from other object modules. Examples of these other modules include files from the main application code, and object files already included from other libraries. The `--library=` syntax tells the linker this file is not in the current directory, and to look for it in the directories collected in the library search path. The `--library=` syntax is not required in the previous example, because the use of angle brackets `<>` has the same effect as --library.

Group Output Sections Together

Suppose you need some output sections to be next to each other in order. You might write ...

```
/* This does NOT work */
output_section_1 > RAM
output_section_2 > RAM
output_section_3 > RAM
```

The linker places all of those output sections in the RAM memory range. But it can place them in any order, and yet other output sections can come in between them. Use the GROUP directive to allocate output sections together in a certain order. Here is an actual example ...

```
GROUP : > CTOMRAM
{
    PUTBUFFER
    PUTWRITEIDX
    GETREADIDX
}
```

The output sections are PUTBUFFER, PUTWRITEIDX, and GETREADIDX. They are allocated to CMTORM as a group, in that exact order. Note how the individual output sections do **not** have any memory allocation specification.

These output section names are in all capital letters. It is an unwritten convention in linker command files that only memory range names like CTOMRAM are written in all capital letters. But you need be prepared for violations of that convention.

Memory Attributes

The MEMORY directive might have lines like these ...

```
MEMORY
{
    ...
    FLASH1 (RX) : origin = 0x00204000, length = 0x1C000
    FLASH2 (RX) : origin = 0x00260000, length = 0x1FFD0
    CSM_RSVD_Z2 : origin = 0x0027FFD0, length = 0x000C
    CSM_ECSL_Z2 : origin = 0x0027FFDC, length = 0x0024
    C0 (RWX)    : origin = 0x20000000, length = 0x2000
    ...
}
```

Note the `(RX)` on two lines and the `(RWX)` on another. That syntax specifies the attributes of those memory ranges. Each letter represents one memory attribute attribute.

- **R**: Can be read
- **W**: Can be written
- **X**: Can contain executable code
- **I**: Can be initialized

By default, a memory range has all four attributes.

The documented purpose of these attributes is to support, in the SECTIONS directive, section allocation by memory attribute. There are no examples of that here, because no TI supplied linker command files use this feature. In this case this syntax is used only as a way to document what kinds of sections usually go in that memory range.