A Primer on Linker Scripts and Command Files

George Mock

Texas Instruments

Embedded Systems Conference

August 2019

Who is George Mock

- Software Developer
- Texas Instruments
- 32 years
- All of them spent working on …
  - C/C++ Compilers
  - Assemblers
  - Linkers
  - Related tools
  - Directly and indirectly
- Moderate the C/C++ part of TI's customer forum

Agenda

- Gentle Introduction
- GCC
  - Basics
  - Examples
- TI
  - Basics: Differences vs GCC
  - Examples
- References

Introduce Linker Scripts

- Compiler turns source files into object files
- Linker takes object files and libraries and produces final executable
- How does linker know where memory is? Where does code and data go in memory?
- Linker Script!
- Two terms for the same thing
  - GCC: *Linker Script*
  - TI: *Linker Command File*

Linker Scripts Are Everywhere

```
C:\dir>ld --verbose
…
/* Default linker script, for normal executables */
…
SECTIONS
{
…
     /* ??? Why is .gcc_exc here?  */
     *(.gcc_exc)
```

What is a Linker Script?

- Text file, just like any other source file
- Another filename in the linker invocation
  - GCC: Usually specify with –T option
  - TI: Usually no option is used
- May contain anything linker accepts on the command line
  - Options
  - Filenames
  - Not covered in this presentation
- Tells linker layout of memory
- How to combine code and data together
- Where to put it in memory

Problem Statement

- You already have a linker script or command file
- It came with the development system you started on
- You need to change it to match your final system
- This presentation explains the linker script you have now, so you can change it
- It does not explain everything

Agenda

- Gentle Introduction
- **GCC**
  - **Basics**
  - Examples
- TI
  - Basics: Differences vs GCC
  - Examples
- References

MEMORY Command

```
MEMORY
{
    /* MCUSS-OCMC RAM RESERVED FOR MCUSS & SOC Boot  - 384KB          */
    MCU_RESVD   : ORIGIN = 0x000041C00000, LENGTH = 0x00060000
    /* MCUSS-OCMC RAM - 128KB                                         */
    OCMCRAM     : ORIGIN = 0x000041C60000, LENGTH = 0x00020000
    /* MSMC RAM INIT CODE (4 KB)                                  */
    BOOTVECTOR  : ORIGIN = 0x000070000100, LENGTH = 0x00001000 - 0x100
    /* MSMC RAM GENERAL USE */
    MSMC_SRAM   : ORIGIN = 0x000070001000, LENGTH = 0xEF000
…
```

- Assigns names to regions of memory
- The names are used later in the script

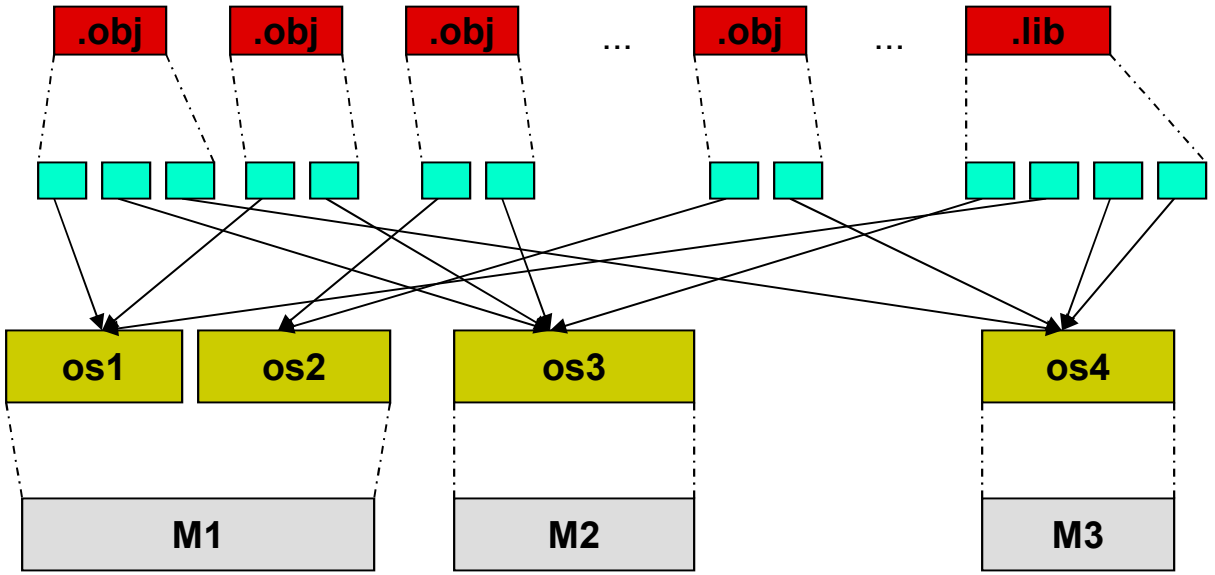
SECTIONS Command

- Does two things at once
  1. Forms output sections from input sections
  2. Allocates the output sections to memory

```
SECTIONS
{
    .text : { *(.text) } > FLASH   /* not explained yet! */
…
}
```

- Unless otherwise stated, all other examples are inside a SECTIONS command

SECTIONS Command Diagram



Object Files & Libs

Input Sections

Output Sections

Memory Regions

SECTIONS Command Glossary of Terms

- Object File
  - Collection of input sections
  - May be presented directly, or come from a library
- Input Section
  - One section from one object file
  - Code or data
  - Data: Initialized or uninitialized
- Output Section
  - Collection of one or more input sections
  - Formed by the SECTIONS command
- Memory Region
  - Range of memory specified in the MEMORY command

Data Sections: Initialized vs Uninitialized

```
const int cv = 42;   // const initialized variable
      int iv = 43;   // initialized variable
      int uv;        // uninitialized variable
```

Variable	Section Type	Startup Code	Memory
cv	initialized	nothing	RO or RW
iv	uninitialized	initializes	RW
uv	uninitialized	zero fills	RW

- RO: Read only memory, typically flash
- RW: Read write memory, typically RAM

Section Naming Conventions

- Strictly speaking, a section name says nothing about the contents
- By convention, these names imply these contents

Name	Initialized	Contents
.text	Yes	Executable code
.data	Yes	Initialized data, changes during execution
.bss	No	Global variables, zero filled
.rodata	Yes	Initialized data, always constant

Debug Sections

- Names similar to …
  - .debug_*something*
  - .stabs_*something*
  - .comment
  - .note
- Not loaded to the target
- Used by the debugger
- Ignore them
- If your linker script mentions them, leave them alone

Form Output Sections

- Many shortcuts and wildcards are supported
- This example uses none of them

```
output_section_name :
{
   file1.o(.text)
   file2.o(.text)
   file3.o(.text)
} > FLASH
```

- Name: **`output_section_name`**
  - Does not have to begin with a dot "."
- Contains three input sections named **`.text`** from those specific object files
- Allocated to the **`FLASH`** memory region

Form Output Sections

- The previous example does not scale to a system with many files
- This example uses one shortcut

```
output_section_name :
{
   *(.text)
} > FLASH
```

- Same as the previous example, except …
- Contains the **`.text`** input section from all the object files

Form Output Sections

- The previous example is correct, but not typical
- This example is typical …

```
.text :
{
   *(.text)
} > FLASH
```

- Same as the previous example, except …
- The name of the output section changed from **`output_section_name`** to **`.text`**
- Even though the names are the same, do not overlook the distinction between input sections and the output section which contains them

Form Output Sections – TI only

- TI linker command files support a further shortcut

```
.text > FLASH
```

- No different from the previous example

SECTIONS Command Program Counter

- Within SECTIONS, a PC is maintained
- Thus, a MEMORY command is not strictly necessary
- This is a complete linker script

```
SECTIONS
{
     . = 0x10000;              /* Set PC = 0x10000                          */
     .text : { *(.text) }
     . = 0x8000000;            /* Set PC = 0x8000000                        */
     .data : { *(.data) }
     .bss  : { *(.bss)  }      /* .bss starts at 0x8000000 + sizeof(.data)  */
}
```

- Typical of scripts for hosted systems
- Most (probably all) scripts for embedded systems use the MEMORY command

REGION_ALIAS

- Defines another name for a memory region

```
REGION_ALIAS(“NEW_REGION_NAME”, EXISTING_REGION_NAME);
```

- Supports separate specification of MEMORY and SECTIONS commands

```
REGION_ALIAS("REGION_TEXT", DDR_0);
REGION_ALIAS("REGION_BSS", MSMC_SRAM_H);
REGION_ALIAS("REGION_DATA", DDR_0);
…
```

- Written outside of MEMORY and SECTIONS commands

REGION_ALIAS Example

```
MEMORY {
    DDR_0 (RWX) : ORIGIN = 0x80000000, LENGTH = 0x10000000
    …
}

REGION_ALIAS("REGION_TEXT", DDR_0);
…

SECTIONS {
…
    .text : {
        …
    } > REGION_TEXT
```

- HW expert writes MEMORY and REGION_ALIAS
- SW expert writes SECTIONS in terms of REGION_ALIAS

Agenda

- Gentle Introduction
- **GCC**
  - Basics
  - **Examples**
- TI
  - Basics: Differences vs GCC
  - Examples
- References

Example 1: .rodata

```
.rodata : {
    *(.rodata)
    *(.rodata*)                    /* explained next slide */
} > REGION_TEXT AT> REGION_TEXT
```

- Output section named **`.rodata`**
- Contains all input sections named **`.rodata`**
- Explaining **`*(.rodata*)`** requires an entire slide
- Allocated to **`REGION_TEXT`**
  - This instance of **`AT>`** has no effect
  - More detail to come

Sections per Entity and Garbage Collection

- Compile with **`-ffunction-sections -fdata-sections`**
- Link with **`-Wl,--gc-sections`**
- Each function and global data item in separate input section
- Input section is named after the entity
  - Examples: **`.text.function_name`**, **`.rodata.const_array_name`**
- Garbage collects (removes) functions never called and data items never used
  - If nothing gets garbage collected, total program size is bigger

```
        *(.rodata*)                    /* explained THIS slide */
```

- Collects all the input sections that start with the name **`.rodata`**
  - Such as **`.rodata.const_array_name`**

Specific Allocation of One Input Section

```
special_output_section : {
    *(.rodata.const_array_name)
} > SPECIAL_MEMORY_REGION
```

- Output section named **`special_output_section`**
- Contains one input section named **`.rodata.const_array_name`**
- Allocated to **`SPECIAL_MEMORY_REGION`**

Example 2: .data

```
.data : ALIGN(8) {
     __data_load__ = LOADADDR (.data);
     __data_start__ = .;
     *(.data)
     *(.data*)
     . = ALIGN(8);
     __data_end__ = .;
 } > REGION_DATA AT> REGION_TEXT
```

- Output section named **`.data`**
- Contains all input sections named **`.data`**, and start with **`.data`**
- Explain all the rest in the next few slides

Different Load and Run Address

```
    } > REGION_DATA AT> REGION_TEXT
```

- All output sections have two allocations: run and load
  - GCC docs use the terms VMA (run) and LMA (load)
  - VMA: Virtual Memory Address
  - LMA: Load Memory Address
- Default: run == load
- Specify different load address with **AT>** syntax
- Typical use case: load in flash, run in RAM
- A copy from flash to RAM must occur early in execution, usually as part of system startup

Symbols for Run and Load Addresses

```
.data : ALIGN(8) {
     __data_load__ = LOADADDR (.data);
     __data_start__ = .;
     *(.data)
     *(.data*)
     . = ALIGN(8);
     __data_end__ = .;
 } > REGION_DATA AT> REGION_TEXT
```

- These symbols are used to implement the copy from load to run
- Copy length: **`__data_end__ - __data_start__`**

Alignment

```
.data : ALIGN(8) {
     __data_load__ = LOADADDR (.data);
     __data_start__ = .;
     *(.data)
     *(.data*)
     . = ALIGN(8);
     __data_end__ = .;
 } > REGION_DATA AT> REGION_TEXT
```

- First **`ALIGN(8)`** aligns output section to an 8 byte boundary
- Last **`ALIGN(8)`** insures output section length is a multiple of 8 bytes
  - If a gap is created, it is filled with 0

Example 3: .text

```
.text : {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    *(.text.*)
    . = ALIGN(0x8);
    KEEP (*(.ctors))
    . = ALIGN(0x8);
    KEEP (*(.dtors))
    . = ALIGN(0x8);
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
    *(.init)
    *(.fini*)
} > REGION_TEXT AT> REGION_TEXT
```

- Output section named **`.text`**
- Contains all input sections named **`.text`**, and start with **`.text`**
- Plus other input sections
- Despite use of **`AT>`**, run and load allocation is the same REGION_TEXT
- This script consistently uses **`AT>`** for every allocation

Symbols for Input Files

```
.text : {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    *(.text.*)
    . = ALIGN(0x8);
    KEEP (*(.ctors))
    . = ALIGN(0x8);
    KEEP (*(.dtors))
    . = ALIGN(0x8);
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
    *(.init)
    *(.fini*)
} > REGION_TEXT AT> REGION_TEXT
```

- Creates a symbol for each input file
- Named after the file
- Program probably does not use these symbols, but I did not verify that
- Does not increase code size
- Increases the number of symbols, which may slow load time and debugging

C++ Sections

```
.text : {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    *(.text.*)
    . = ALIGN(0x8);
    KEEP (*(.ctors))
    . = ALIGN(0x8);
    KEEP (*(.dtors))
    . = ALIGN(0x8);
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
    *(.init)
    *(.fini*)
} > REGION_TEXT AT> REGION_TEXT
```

- **KEEP** disables garbage collection of these input sections
- These sections are related to constructors and destructors for C++ objects with global or static scope
- Startup code constructs these objects before main starts
- Cleanup code destructs these objects after main ends

C++ Startup Symbols

```
.text : {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    *(.text.*)
    . = ALIGN(0x8);
    KEEP (*(.ctors))
    . = ALIGN(0x8);
    KEEP (*(.dtors))
    . = ALIGN(0x8);
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
    *(.init)
    *(.fini*)
} > REGION_TEXT AT> REGION_TEXT
```

- **`.init_array`** contains pointers to functions called during startup
- These symbols mark the start and end of **`.init_array`**

More Startup Code

```
.text : {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    *(.text.*)
    . = ALIGN(0x8);
    KEEP (*(.ctors))
    . = ALIGN(0x8);
    KEEP (*(.dtors))
    . = ALIGN(0x8);
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
    *(.init)
    *(.fini*)
} > REGION_TEXT AT> REGION_TEXT
```

- More code related to startup
- Not explained in this presentation

Odd Syntax

```
.text : {
    CREATE_OBJECT_SYMBOLS
    *(.text)
    *(.text.*)
    . = ALIGN(0x8);
    KEEP (*(.ctors))
    . = ALIGN(0x8);
    KEEP (*(.dtors))
    . = ALIGN(0x8);
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
    *(.init)
    *(.fini*)
} > REGION_TEXT AT> REGION_TEXT
```

- Why the inconsistent use of the trailing asterisk?
- I suspect it's wrong, but did not investigate

More Symbols for .text

```
.text : {
…
} > REGION_TEXT AT> REGION_TEXT

PROVIDE (__etext = .);
PROVIDE (_etext = .);
PROVIDE (etext = .);
```

- Defined outside of output section, thus uses the SECTIONS command PC
- Marks the end of **`.text`**
- **`PROVIDE`** is similar to weak
  - With regard to the named symbol
  - Can be overridden by a definition in the program
  - If never referenced, not created
- Different variants of the symbol name **`etext`** have appeared over the years
  - So provide all of them

Agenda

- Gentle Introduction
- GCC
  - Basics
  - Examples
- **TI**
  - **Basics: Differences vs GCC**
  - Examples
- References

Terminology: GCC vs TI

GCC	TI
Linker script	Linker command file
MEMORY command	MEMORY directive
SECTIONS command	SECTIONS directive
Memory region	Memory range

Other Differences

- SECTIONS command PC (program counter)
  - GCC: yes
  - TI: no
- REGION_ALIAS
  - GCC: yes
  - TI: no
- C preprocessor statements like #include, #define, etc.
  - GCC: no
  - TI: yes

Syntax Difference Regarding Colon

- GCC requires this colon

```
.text : { /* input sections here */ } > FLASH
```

- TI does not

```
.text { /* input sections here */ } > FLASH
```

TI Shortcut Repeated

```
    .text > FLASH
```

- Output section named **`.text`**
- Contains all the input sections named **`.text`**
- Allocated to **`FLASH`**

Agenda

- Gentle Introduction
- GCC
  - Basics
  - Examples
- **TI**
  - Basics: Differences vs GCC
  - **Examples**
- References

First Output Section in a Memory Range

```
#define BASE 0x00200000

MEMORY {
  FLASH : o = BASE, l = 0xFFD4
…
}

SECTIONS {
    /* only one to use BASE */
    .intvecs > BASE
    .text    > FLASH
    .const   > FLASH
…
}
```

- **`.intvecs`** is the first output section allocated to **`FLASH`**
- All other output sections are in any order
- Allocations to a specific address are always done before allocations to a named memory range

Allocate to Multiple Memory Ranges

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

- Output section named **`.text`**
- Contains all the input sections named **`.text`**
- Allocated to the first memory range which can completely contain it

Split an Output Section Across Multiple Memory Ranges

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

- Note **>>** instead of **>**
- Output section .text is split across those memory ranges
- Split occurs on input section boundaries
  - Thus a split never occurs in the middle of a function, array, etc.
- Memory ranges are used in that order

Group Output Sections Together

- Use Case: Some output sections need to be together in order
- A first attempt might be

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

- All those sections go in RAM, but in any order
  - Other output sections can come in between them
- Use GROUP instead

Group Output Sections Together

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

- Output sections are PUTBUFFER, PUTWRITEIDX, and GETREADIDX
- Allocated to CTOMRAM memory range in that order
- Colon is optional
- Memory range name may be written after the closing brace instead
- Violates the unwritten convention that section names are written in all lower case

Agenda

- Gentle Introduction
- GCC
  - Basics
  - Examples
- TI
  - Basics: Differences vs GCC
  - Examples
- **References**

References

- This presentation is based on this article http://software-dl.ti.com/ccs/esd/documents/sdto_cgt_Linker-Command-File-Primer.html (link)
- GCC Linker Manual https://sourceware.org/binutils/docs/ld/ (link)
- All manuals for TI Compilers, Assemblers, Linkers, etc. http://www.ti.com/tool/TI-CGT#technicaldocuments (link)

Questions?