C28x Compiler - Understanding Linking

From Texas Instruments Wiki

Contents

Introduction

This page describes the compiler section names and how to link them in your project. This article has been written with the C28x flash device in mind.

Other Resources

- TMS320C28x Optimizing C/C++ Compiler User's Guide (spru514)
- TMS320C28x Assembly Language Tools User's Guide (spru513)

The following application note describes in detail how to link code when running in RAM or Flash. It also includes information on including DSP/BIOS.

- Running an Application from Internal Flash Memory on the TMS320F28xx DSP (spra958)
- Associated code (http://www.ti.com/lit/zip/spra958)

The following is great general Code Generation Tools Information. Much of it applies to 28x:

- Compiler Wiki Area
- Code Gen Tools Tips and Tricks for Beginners Wiki article
- Code Generation Tools FAQ Wiki article

Understanding Compiler Sections

Every C program consists of different parts called sections. All default sections begin with a leading dot (.text, .ebss, etc). The compiler has section names for initalized as well as unitialized variables. The section names that will be generated by the following code are indicated in the comments:

```
//
// Global variables x & y ==> .ebss
// Initial values   2 & 7 ==> .cinit
//

int x = 2;
int y = 7;

void main()
{
  long z;     // Local variable => .stack
  z = x + y;  // Code => .text
}
```

The Compiler sections are documented in the TMS320C28x Optimizing C/C++ Compiler User's Guide (spru514).

Types of Compiler Sections

During development both initalized sections and uninitalized sections should be linked to SARAM. This makes debug faster since the flash will not need to be erased and programmed after each minor change.

Once the initial debug is complete and the code works, initialized sections can be moved to FLASH. Once this is done the code should be tested again.

Linking C28x Compiler Sections

Section Name	Description	Section Type	Link Location
.text	Executable code and constants	Initalized	Load into non-volatile memory: Flash Run time critical code in SARAM Note (1)
.cinit	Initalized global and static variables	Initalized	Non volatile memory: Flash
.econst	Constant data (e.g. const int k = 3;)	Initalized	Non volatile memory: Flash
.switch	Tables for switch statements	Initalized	Non volatile memory: Flash
.ebss	Global and static variables	Uninitalized	Volatile memory: SARAM
.stack	Stack space	Uninitalized	Volatile memory: SARAM The 28x Stack Pointer (SP) is 16-bits .stack must be in the low 64k words
.esysmem	Memory for malloc type functions	Uninitalized	Volatile memory: SARAM
.reset	Reset vector	Initalized	This section is not typically used by C28x devices and should be set to type=DSECT

Notes: (1) Time critical code will typically be linked with a load address in flash, and a run address in SARAM. The application will then copy it from flash to SARAM.

The .reset section

The compiler generates a section called .reset that holds nothing more than the address of the _c_init00 routine (the c-initilization routine). The purpose of this section is to provide a reset vector. For most 28x devices, however, the reset vector will be in the boot ROM of the device. For this reason, the .reset section can be set to "TYPE = DSECT" in the linker command file as described later in this article.

The exception to this rule is a 281x device with external interface. On this device zone 7 can be mapped such that the reset vector is in

external memory. In this case the .reset section might be useful.

Controlling the Linking Process

The linking step uses the memory description from the linker command file (.cmd). This file tells the linker how to place the software into the hardware. The format of the linker command file is described in the TMS320C28x Assembly Language Tools User's Guide (spru513).

It's suggested that you start with a linker file from a working example and then modify it for your needs. The linker command file has two basic parts.

Memory Map Description

This part of the .cmd file tells the linker what memory blocks are available.

- Memory block name
- The memory block location
- Size of the memory block

An example of the memory map description is shown below:

```
/* Memory Map Description
   Describe the memory blocks on the device */

MEMORY
{
   PAGE 0:      /* Program Space */
   H0SARAM:     org = 0x00A000,  len = 0x2000

   PAGE 1:       /* Data Space */
   M0SARAM:     org = 0x000000,  len = 0x400
   M1SARAM:     org = 0x000400,  len = 0x400
   L0SARAM:     org = 0x008000,  len = 0x1000
   L1SARAM:     org = 0x009000,  len = 0x1000

}
```

PAGE 0 or PAGE 1

PAGE 0 is typically used for program sections. For example .text or other named sections that contain code.

PAGE 1 is used for data sections. For example .stack.

In some legacy Flash programming tools, such as SDFlash, only sections on "PAGE 0" would be programmed into the flash. Consider the .const section which is data. If .const was allocated to PAGE 1, the SDFlash flash programmer will ignore it and not program the data into flash. Since this is initialized data that is constant you really want it programmed into flash. In some of our examples you will find .const allocated to PAGE 0 for this reason. Memory blocks on C28x are unified so it does not cause an issue.

A Note About Unified Memory

Memory blocks on 28x devices are typically unified. That means they can be accessed both in program space and data space. Refer to the data manual for your device. There will be a memory map that shows the physical memory blocks and where they are defined - program, data or both.

In the MEMORY description, these unified blocks can be placed either in page 0 (i.e. program) or page 1 (i.e. data). It is also possible to split a physical memory block so part of the block is used by page 0 and another part used by page 1.

Important Notes When Defining the MEMORY Section

- If code accesses data within the same physical memory, then performance will degrade due to resource conflicts. It is better

to place code and the data it accesses separate blocks.
- Take care to not define the same memory address range in both program and data space! Doing so may cause your data to corrupt program or vice versa.
- Wait states will degrade performance. Most SARAM is zero-wait on 28x devices. There are some blocks, however, that are not on 2833x devices. Always check the data manual to find the wait states for each physical block and whether it applies to program or data accesses.
- Flash sectors can be combined to form larger memory segments. The thing to keep in mind is the smallest amount that can be erased is a physical sector. This may affect how code is partitioned for field upgrades.
- Continuous SARAM blocks can be combined to form a larger memory segment.
- Devices with DMA: only some blocks will be accessible by the DMA. Refer to the data manual.
- Devices with CLA: only some blocks will be accessible by the CLA. Refer to the data manual.

Sections Description

This part of the linker command file tells the linker:

- Which software sections should be allocated to which memory regions
- Allows you to allocate to memory on a per-file basis
- Allows for separate load and run locations

An example sections description is shown below:

```
/* Sections Description
   Tell the linker where to allocate sections */

SECTIONS
{
   .text:      >  H0SARAM      PAGE 0
   .ebss:      >  M0SARAM      PAGE 1
   .cinit:     >  H0SARAM      PAGE 0
   .stack:     >  M1SARAM      PAGE 1
   .reset:     >  H0SARAM      PAGE 0, TYPE = DSECT
}
```

If you do not specify a generated section then the liker will try to place it for you. This may result in it being placed somewhere you don't want it to be. For example, the .ebss section may end up in flash which will not work! To receive a warning message when the linker assigns unspecified sections use the -w linker switch.

Frequently Asked Questions

Q: Why do I need a linker command file? Why doesn't the compiler take care of memory management?

For simple examples it may not matter much what memory is used. But in embedded processing it quickly does become important.

Here is a list of some things to consider when placing code/data:

- Speed: RAM is faster than Flash. Flash is faster than XINTF.
- Does it need to be stored in non-volatile memory like Flash?
- Is it time critical? Example: a time critical ISR stored in flash and copied to RAM.
- Some RAM may be accessible by a DMA module while other RAM is not.
- Use separate physical RAM blocks to avoid resource conflicts. On C2000 RAM blocks are single access (SARAM).
- Is a resource on external memory?
- Should it be in memory with a cache or prefetch to improve performance?
- Does the memory have wait states in program space, data space, or both?
- Is it something that needs to be protected by the Code Security Module? Not all memory is CSM protected.

The linker command file is how you describe your device to the tools and tell it where you want different sections to go.

If you don't assign a particular section to a memory region then the tools will indeed try to place it in any available memory. This

may not be desired for the above reasons. Because of this I suggest always linking with the -w flag so the tools will tell you it is assigning a section to memory without your influence.

Q: The linker says "placement fails for object" but the available memory is larger than the section

This is most likely happening because the section is "blocked" by the compiler. This means it needs to either fit completely within a page or begin at a page boundary. Meeting that requirement does not allow it to fit within the space available in the memory block. The compiler does this to optimize the data page (DP) register load. For example, the compiler can safely load DP for an array once and access all its elements (or the first 64-words) without having to reload DP. If blocking is not enabled the compiler has to load DP for accessing each element leading to much more inefficiencies in terms of code size and performance.

The alternative would be to generate a DP load before every direct memory access. Obviously this could have a serious negative impact on code size. There is an option though that will cause DP loads on every access, and disable blocking. If you add the -md switch to your project it will disable blocking but impact code size and performance - this is not recommended.

You can confirm that a particular section is "blocked" by looking at the .bss and .usect directives in the assembly listing file and checking that the blocking flag is set. The blocking flag is documented in the C28x Assembly Language Tools Users Guide under "Uninitialized Sections".

Q: The linker says "placement fails for object '.text' ". How can I make more memory available for .text?

There are a few options - here are three:

Option 1

The linker command file (.cmd) specifies how memory will be allocated for the project. You can try modifying the .cmd file to allow for a larger .text section. For example, you can try combining contiguous RAM blocks or flash sectors to make a larger block for code.

Option 2

You can "split" the .text section among multiple memory regions using the following syntax:

```
  .text             : >> FLASHA | FLASHC | FLASHD,     PAGE = 0
```

Option 3

You can tell the linker to allocate as a whole into the first memory range in which it fits completely, by doing this:

```
 .text             : > FLASHA | FLASHC | FLASHD,     PAGE = 0
```

For any of these options, keep in mind that all blocks may not be equal in functionality - For example:

- On the Piccolo devices with Control Law Accelerator only some memory blocks can be used by the CLA.
- On devices with the DMA not all memory can be accessed by the DMA.
- Most SARAM blocks are single access (zero wait state) but on some devices there is a wait state in program memory. For example on 2833x the memories that the DMA can use are zero wait in data space, but 1 wait in program space. This makes them more suited towards pure data accesses.

Q: In the linker command file, can I combine continuous flash blocks into one section?

Yes - Flash sectors can be combined in the MEMORY section.

On the flash side, the thing to keep in mind is a **sector is the smallest amount of flash that can be erased at a time.** Thus you may want to partition your code such that code you upgrade (or don't upgrade) aligns to a sector.

Option 1

Combine two memory regions, in this case two sectors, into one:

Two sectors:

```
MEMORY
```

```
{
   //
   // Individual sectors E and F called out in the MEMORY description
   //
   ...
   FLASHF      : origin = 0x310000, length = 0x008000     /* on-chip FLASH */
   FLASHE      : origin = 0x318000, length = 0x008000     /* on-chip FLASH */
   ...
}
```

One combined memory region made of two sectors:

```
MEMORY
{
   //
   // Sectors E and F merged into one in the MEMORY description
   //
   ...
   FLASHF_E       : origin = 0x310000, length = 0x010000     /* on-chip FLASH F & FLASH E */
   ...
}
```

Option 2

Another option is to split the sections across multiple memory segments. This is described in the TMS320C28x Assembly Language Tools User's Guide (spru513).

```
SECTIONS
{
   .text: { *(.text) } >> FLASHE| FLASHH
}
```

Q: In the linker command file, I combine contiguous SARAM blocks into one section?

Yes, you can do this by combining memory blocks or splitting the section across memory blocks (described above for flash).

The thing to remember is the SARAM blocks are single access. So it is best to partition code into a different physical SARAM block than data that code is accessing. Keeping this partition will will improve performance.

Q: My project is linking successfully. How do I find details about my project's memory utilization?

After a project links successfully, a .map file can be generated which provides information about memory utilization and symbol placement. Please see the TMS320C28x Assembly Language Tools User's Guide for more information.

Q: I have a DSP/BIOS project. What should I know about linking?

Please refer to this application note:

- Running an Application from Internal Flash Memory on the TMS320F28xx DSP (spra958)
- Associated code (http://www.ti.com/lit/zip/spra958)

*For technical support on the C2000 please post your questions on The C2000 Forum (http://e2e.ti.com/support/microcontrollers/tms320c2000_32-bit_real-time_mcus/f/171.aspx). Please post only comments about the article **C28x Compiler - Understanding Linking** here.*

Links

Amplifiers & Linear (http://www.ti.com/lsds/ti/analog/amplifier_and_linear.page)
Audio (http://www.ti.com/lsds/ti/analog/audio/audio_overview.page)
Broadband RF/IF & Digital Radio (http://www.ti.com/lsds/ti/analog/rfif.page)
Clocks & Timers (http://www.ti.com/lsds/ti/analog/clocksandtimers/clocks_and_timers.page)
Data Converters (http://www.ti.com/lsds/ti/analog/dataconverters/data_converter.page)

DLP & MEMS (http://www.ti.com/lsds/ti/analog/mems/mems.page)
High-Reliability (http://www.ti.com/lsds/ti/analog/high_reliability.page)
Interface (http://www.ti.com/lsds/ti/analog/interface/interface.page)
Logic (http://www.ti.com/lsds/ti/logic/home_overview.page)
Power Management (http://www.ti.com/lsds/ti/analog/powermanagement/power_portal.page)

Processors (http://www.ti.com/lsds/ti/dsp/embedded_processor.page)

- ARM Processors (http://www.ti.com/lsds/ti/dsp/arm.page)
- Digital Signal Processors (DSP) (http://www.ti.com/lsds/ti/dsp/home.page)
- Microcontrollers (MCU) (http://www.ti.com/lsds/ti/microcontroller/home.page)
- OMAP Applications Processors (http://www.ti.com/lsds/ti/omap-applications-processors/the-omap-experience.page)

Switches & Multiplexers (http://www.ti.com/lsds/ti/analog/switches_and_multiplexers.page)
Temperature Sensors & Control ICs (http://www.ti.com/lsds/ti/analog/temperature_sensor.page)
Wireless Connectivity (http://focus.ti.com/wireless/docs/wirelessoverview.tsp?familyId=2003&sectionId=646&tabId=2735)

Categories: C2000 | Compiler | Development Tools for C2000