Basic Memory Management

Introduction

Memory management involves:
• Defining system memory requirements
• Describing the available memory map to the linker
• Allocating code and data sections using the linker

The latter two, along with the C6000 memory architecture are covered in this chapter.

Defining memory requirements is very application specific and therefore, is outside the scope of this workshop. If you have any questions regarding this, please discuss these during a break with your instructor.

Learning Objectives

Outline

◆ C6416 Memory Architecture
◆ C6713 Memory Architecture
◆ Section → Memory Placement
Module Topics

Basic Memory Management

C6416 Memory Architecture
C6416 Internal Memory
C6416 External Memory
C6416 DSK Memory
What is a Memory Map?

C6713 Memory Architecture
C6713 Internal Memory
C6713 External Memory
C6713 DSK Memory

Section → Memory Placement
What is a Section?
Let’s Review the Compiler Section Names
Exercise - Section Placement
How Do You Place Sections into Memory Regions?
  1. Creating a New Memory Region (Using MEM)
  2. Placing Sections – MEM Manager Properties
  3. Running the Linker

Optional Discussion
‘0x Memory Scheme
‘1x Memory Scheme
C6416 Memory Architecture

C6416 Internal Memory

The C6416 internal memory map consists of two parts, Level 1 and Level 2.

Level 1 consists of two 16K-byte cache memories, one program, the other for data. Since these memories are only configurable as cache they do not show up in the memory map. (Cache is discussed further in an upcoming chapter.)

Level 2 memory consists of 1M bytes of RAM – and up to 256K bytes can be made cache. (If a segment is configured as cache, it doesn’t show up in the memory map.) This is a unified memory, that is, it can hold code or data.
C6416 External Memory

External memory is broken into 4 CE (chip enable) spaces: CE0, CE1, CE2, CE3, per External Memory Interface (EMIF), each up to 1Gbytes long. Each CE space can contain program or data memory using asynchronous or synchronous memories (more on this in the EMIF module).

- Each EMIF has four ranges
  - Program or Data
  - Named: CE0, CE1, CE2, CE3
- Remaining memory is unused

C64x Memory Details

- Each device is different
- Some have two EMIF's
  - EMIFA is 64-bits wide
  - EMIFB is 16-bits wide

<table>
<thead>
<tr>
<th>Devices</th>
<th>Internal (L2)</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6414</td>
<td>1MB</td>
<td>A: 1GB (64-bit) B: 256MB (16-bit)</td>
</tr>
<tr>
<td>C6415</td>
<td>256KB</td>
<td>1GB (64-bit)</td>
</tr>
<tr>
<td>C6416</td>
<td>256KB</td>
<td>256MB (32-bit)</td>
</tr>
<tr>
<td>DM642</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6411</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C6416 DSK Memory

Based on the C6416’s memory-map, how does the C6416 DSK use this map?

'D6416 DSK Block Diagram

- DSK uses both EMIFs (A and B)
- EMIFA
  - CE0 for SDRAM
  - CE2 and CE3 pinned-out to daughter card connector
- EMIFB
  - CE1 for Flash Memory and CPLD (switches, LED’s, etc.)
There are a few ways to view the memory architecture in your system. One is to use a block diagram approach (shown at the top of the slide below). Another way, which is often more convenient is to display the addresses and “contents” of the memories in a table format called a Memory Map.

**What is a Memory Map?**

A Memory Map is a table representation of memory...
C6713 Memory Architecture

The C6713's memory architecture is very similar to that of the C6416. We're going to highlight the differences here.

### C6713 Internal Memory

The C6713 has a two-level memory architecture just like the C6416. The Level 1 Caches are 4KB each (Program and Data). The Level 2 memory is 256KB, and up to ¼ of it can be made cache. You can actually add 16KB cache ways for up to a 4 way set-associative cache.
C6713 External Memory

The C6713 has one EMIF with four external ranges. Each range has a dedicated strobe (CEx). The memory addresses that fall outside of the ranges are unused.

- Four External ranges
  - Program or Data
  - 128 Mbytes each
  - Named: CE0, CE1, CE2, CE3
- Remaining memory is unused

How does this apply to the DSK?
C6713 DSK Memory

Here is a block diagram of the memory (internal and external) that is available on the C6713 DSK.

So what does the Memory Map look like?

One of the biggest differences between the two chips is that the C6713 only has one EMIF. The FLASH on the C6713 DSK is also 256KB, as opposed to 512KB on the C6416 DSK.
Here is the memory map for the C6713 DSK. This shows the total available memory that a C6713 has, and how that memory was used on the DSK.
Section → Memory Placement

What is a Section?

Looking at a C program, you'll notice it contains both code and different kinds of data (global, local, etc.).

Every C program consists of different parts called Sections

All default section names begin with "."

In the TI code-generation tools (as with any toolset based on the COFF – Common Object File Format), these various parts of a program are called Sections. Breaking the program code and data into various sections provides flexibility since it allows you to place code sections in ROM and variables in RAM. The preceding diagram illustrated five sections:

- Global Variables
- Initial Values for global variables
- Local Variables (i.e. the stack)
- Code (the actual instructions)
- Standard I/O functions

Though, that’s not all the sections broken out by the C6000’s compiler …
Let’s Review the Compiler Section Names

Following is a list of the sections that are created by the compiler. Along with their description, we provide the Section Name defined by the compiler.

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Description</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>Code</td>
<td>initialized</td>
</tr>
<tr>
<td>.switch</td>
<td>Tables for switch instructions</td>
<td>initialized</td>
</tr>
<tr>
<td>.const</td>
<td>Global and static string literals</td>
<td>initialized</td>
</tr>
<tr>
<td>.cinit</td>
<td>Initial values for global/static vars</td>
<td>initialized</td>
</tr>
<tr>
<td>.pinit</td>
<td>Initial values for C++ constructors</td>
<td>initialized</td>
</tr>
<tr>
<td>.bss</td>
<td>Global and static variables</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.far</td>
<td>Global and static variables</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.stack</td>
<td>Stack (local variables)</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.sysmem</td>
<td>Memory for malloc fcns (heap)</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.cio</td>
<td>Buffers for stdio functions</td>
<td>uninitialized</td>
</tr>
</tbody>
</table>

If you think some of these names are a bit esoteric, we agree with you. (.code might have made more sense than .text, but we have to live with the names they chose.)

You must link (place) these sections to the appropriate memory areas as provided above. In simplest terms, *initialized* might be thought of as ROM-type memory and *uninitialized* as RAM-type memory.
Exercise - Section Placement

Where would you anticipate these sections should be placed into memory? Try your hand at placing five sections and tell us why you would locate them there.

<table>
<thead>
<tr>
<th>Section</th>
<th>Location</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.cinit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.bss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.cio</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hint: Think about what type of memory each one should reside in – ROM or RAM.
Solution? There are actually many solutions to this problem, depending on your system’s needs. If you are contemplating booting your system from reset, then your answers may be very different from a non-booted system. Here’s what we came up with:

<table>
<thead>
<tr>
<th>Section</th>
<th>Location</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>FLASH</td>
<td>Must exist after reset</td>
</tr>
<tr>
<td>.cinit</td>
<td>FLASH</td>
<td>Must exist after reset</td>
</tr>
<tr>
<td>.bss</td>
<td>Internal</td>
<td>Must be in RAM memory</td>
</tr>
<tr>
<td>.stack</td>
<td>Internal</td>
<td>Must be in RAM memory</td>
</tr>
<tr>
<td>.cio</td>
<td>SDRAM</td>
<td>Needs RAM, speed not critical</td>
</tr>
</tbody>
</table>

Also, consider a bootable system. Some sections may initially be “loaded” into EPROM but “run” out of internal memory. How are these sections handled? If you thought of this, great. We’ll tackle how to do this later.
How Do You Place Sections into Memory Regions?

Now that we have defined these sections and where we want them to go, how do you create the memory areas that they are linked to and how do you actually link them there?

Placing Sections In Memory

0000_0000
8000_0000
9000_0000

.s. t. ex

.bss

.cini t

.cio

.stack

1MB Internal
16MB SDRAM
4MB FLASH

◆ How do you define the memory areas (e.g. FLASH, SDRAM) ?
◆ How do you place the sections into these memory areas ?

Linking code is a three step process:

1. Defining the various regions of memory (on-chip RAM vs. EPROM vs. SDRAM, etc.)
2. Describing what sections go into which memory regions
3. Running the linker with “build” or “rebuild”
1. Creating a New Memory Region (Using MEM)

First, to create a specific memory area, open up the .CDB file, right-click on the Memory Section Manager and select “Insert MEM”. Give this area a unique name and then specify its base and length. Once created, you can place sections into it (shown in the next step).

Using the Memory Section Manager

- MEM Manager allows you to create memory areas & place sections
- To Create a New Memory Area:
  - Right-click on MEM and select Insert Mem
  - Fill in base/len, etc.

How do you place sections into these memory areas?

Note: The heap part of this dialog box is discussed later.
2. Placing Sections – MEM Manager Properties

The configuration tool makes it easy to place sections. The predefined compiler sections that were described earlier each have their own drop-down menu to select one of the memory regions you defined (in step 1).

MEM Manager Properties

◆ To Place a Section Into a Memory Area...
1. Right-click on MEM Section Manager
2. Select the appropriate tab (e.g. Compiler)
3. Select the memory area for each section

What about the BIOS Sections?
There are 3 tabbed pages of pre-defined section names:
(1) BIOS Data Sections
(2) BIOS Code Sections
(3) Compiler sections

Placing BIOS Sections

- BIOS creates both Data and Code sections
- User needs to place these into appropriate memory region

What gets created after you make these selections?

We haven’t had the opportunity to describe all the BIOS-related sections. Please refer to the online help for a description of each.

At times you will need to define and place your own user-defined sections, this is discussed later in the chapter.
Initialized Sections

Earlier we discussed putting some sections into initialized (ROM) memory. When debugging our code with CCS, though, we haven’t been putting these sections into ROM. How can the system work?

The key lies in the difference between ROM and initialized memory. ROM memory is a form of initialized memory. After power-up ROM still contains its values – in other words it’s initialized after power-up.

Therefore, for our system to work, the initialized sections must “exist” before we start running our code. In production we can program EPROM’s or Flash memory ahead of time. Or, maybe a host downloads the initialized code and data before releasing the processor from reset.

Initialized Memory

- CCS loader copies the following sections into volatile memory:
  - .text
  - .switch
  - .cinit
  - .pinit
  - .const
  - .bios
  - .sysinit
  - .gbInit
  - .trcdata
  - .hwi_vec
  - .rtdx_text

When using the CCS loader (File:Load Program…), CCS automatically copies each of the initialized sections (.text, .switch, .cinit, .pinit, .const, etc.) into volatile memory on the chosen target.

Later in the workshop we will examine more advanced ways to locate initialized sections of code and data. We even will get a chance to burn them into a Flash memory and re-locate them at runtime. But for now, we won’t try anything that fancy.
3. Running the Linker

Creating the Linker Command File (via .CDB)

When you have finished creating memory regions and allocating sections into these memory areas (i.e. when you save the .CDB file), the CCS configuration tool creates five files. One of the files is BIOS’s cfg.cmd file — a linker command file.

This file contains two main parts, MEMORY and SECTIONS. (Though, if you open and examine it, it’s not quite as nicely laid out as shown above.)

Later in the workshop we’ll explore linker command files in greater detail. In fact, you will get to build a custom linker command file in one of the lab exercises.
Running the Linker

The linker’s main purpose is to **link** together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code.

![Diagram of GUI’s Linker Command File]

- **Do not modify** appcfg.cmd – your changes will be overwritten during “Build” (or “Rebuild”).

The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

**Note:** If the graphic above wasn’t clear enough, the linker gets run automatically when you BUILD or REBUILD your project.
Optional Discussion

Entire C6000 Family Memory Description

'0x Memory Scheme

This block diagram represents the maximum allowable memory for the 'C6x0x devices ...
'0x Memory Scheme

- All '0x devices share same external memory map
- CE0,2,3: 16M Bytes; allows SDRAM, SBSRAM and Async
- CET: 4M Bytes; allows SBSRAM and Async only

<table>
<thead>
<tr>
<th>Devices</th>
<th>Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6201</td>
<td>P = 64 KB, D = 64 KB</td>
</tr>
<tr>
<td>C6204</td>
<td>P = 8 M, D = 8 M</td>
</tr>
<tr>
<td>C6205</td>
<td>P = 16 M, D = 16 M</td>
</tr>
<tr>
<td>C6201</td>
<td>P = 256 KB, D = 128 KB</td>
</tr>
<tr>
<td>C6202</td>
<td>P = 1 M, D = 1 M</td>
</tr>
<tr>
<td>C6203</td>
<td>P = 384 KB, D = 512 KB</td>
</tr>
</tbody>
</table>

Technical Training Organization
TT O
'0x alternate Memory Map

**MAP 0**

- 0000_0000: 16M x 8 External
- 0100_0000: 4M x 8 External
- 0140_0000: Internal Program
- 0180_0000: On-chip Peripherals
- 0200_0000: 16M x 8 External
- 0300_0000: 16M x 8 External
- 8000_0000: Internal Data
- FFFF_FFFF: Internal Data

**MAP 1**

- 0000_0000: Internal Program
- 0040_0000: Internal Program
- 0140_0000: 4M x 8 External
- 0180_0000: 4M x 8 External

- **Map 1** moves internal program to location zero
- **Used for boot-loading**
- **No memory lost, only rearranged**
- **Easy, drop-down selection between Map 0/1 with Config Tool**
‘1x Memory Scheme

‘1x Internal Memory

- Level 1 Memory
  - Always cache (not in map)
  - L1P (prog), L1D (data)
- Level 2 Memory (L2)
  - Program or Data
  - Four blocks
  - Each block - Cache or RAM

<table>
<thead>
<tr>
<th>Devices</th>
<th>Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6211</td>
<td>L1P = 4 KB</td>
</tr>
<tr>
<td>C6711</td>
<td>L1D = 4 KB</td>
</tr>
<tr>
<td>C6712</td>
<td>L2* = 64 KB</td>
</tr>
<tr>
<td>C6713</td>
<td></td>
</tr>
<tr>
<td>C6414</td>
<td>L1P = 16 KB</td>
</tr>
<tr>
<td>C6415</td>
<td>L1D = 16 KB</td>
</tr>
<tr>
<td>C6416</td>
<td>L2 = 1 MB</td>
</tr>
</tbody>
</table>

* C6713: L2 = 256KB
'1x External Memory

- All external ranges
  - Program or Data
  - Sync & Async memories
  - Each EMIF has 4 ranges
  - C64x has two EMIF's

<table>
<thead>
<tr>
<th>Devices</th>
<th>EMIF (A) size of range</th>
<th>EMIFB size of range</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6211</td>
<td>128M Bytes (32-bits wide)</td>
<td>N/A</td>
</tr>
<tr>
<td>C6711</td>
<td>64M Bytes (16-bits wide)</td>
<td>N/A</td>
</tr>
<tr>
<td>C6712</td>
<td>64M Bytes (16-bits wide)</td>
<td>N/A</td>
</tr>
<tr>
<td>C6414</td>
<td>256M Bytes (64-bits wide)</td>
<td>64M Bytes (16-bits wide)</td>
</tr>
<tr>
<td>C6415</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6416</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Level 2 Internal Memory

- External (A0)
- External (A1)
- External (A2)
- External (A3)
- External (B0)
- External (B1)
- External (B2)
- External (B3)