Advanced Memory Management

Introduction

Advance memory management involves using memory efficiently. We will step through a number of options that can help you optimize your memory usage as well as your performance needs.

Outline

- Using Memory Efficiently
  - Keep it on-chip
  - Use multiple sections
  - Use local variables (stack)
  - Using dynamic memory (heap, BUF)
  - Overlay memory (load vs. run)
  - Use cache
- Summary
Chapter Topics

Advanced Memory Management ........................................................................................................10-1

Using Memory Efficiently ..............................................................................................................10-3
  Keep it On-Chip ..........................................................................................................................10-3
  Using Multiple Sections ..............................................................................................................10-5
    Custom Sections .......................................................................................................................10-6
    What is the “.far” Section? ......................................................................................................10-7
    Link Custom Sections .............................................................................................................10-8
  Using Local Variables ................................................................................................................10-11
    Everything you wanted or didn’t want to know about the stack ...........................................10-12
    Sidebar: How to PUSH and POP Registers ............................................................................10-13
  Using the Heap ..........................................................................................................................10-14
    Multiple Heaps .......................................................................................................................10-17
    Using MEM_alloc ...................................................................................................................10-19
    Using BUF ................................................................................................................................10-20
  Memory Overlays ........................................................................................................................10-22
    Implementing Overlays (code overlay example) .....................................................................10-23
    Overlay Summary .....................................................................................................................10-24
    Using Copy Tables ..................................................................................................................10-25
  Cache .........................................................................................................................................10-27
  Summary ......................................................................................................................................10-28
Using Memory Efficiently

Keep it On-Chip

One challenge for the system designer is to figure out where everything should be placed. Putting everything on-chip is the easiest way to maximize performance.

From earlier discussions in this chapter, remember that two sections hold most of our code and data. They are:

- `.text` - code and
- `.bss` - global and static variables.
Unfortunately, keeping everything on-chip is not always possible. Often code and data will require too much space and you are left with the decision of what should be kept on-chip and what can reside off-chip. Here are 5 other techniques to help you make the best use of on-chip memory and maximize performance.

<table>
<thead>
<tr>
<th>How to use Internal Memory Efficiently</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Keep it on-chip</td>
</tr>
<tr>
<td>2. Use multiple sections</td>
</tr>
<tr>
<td>3. Use local variables (stack)</td>
</tr>
<tr>
<td>4. Using dynamic memory (heap, BUF)</td>
</tr>
<tr>
<td>5. Overlay memory (load vs. run)</td>
</tr>
<tr>
<td>6. Use cache</td>
</tr>
</tbody>
</table>
Using Multiple Sections

If your code and data cannot all fit on-chip, create multiple sections.

If these sections are too big to fit on-chip, you will have to place them off-chip. But you may still want to put critical function and/or data on-chip.
**Custom Sections**

In order to use multiple sections, you’ll need a way to create them:

### Making Custom Code Sections

- **Create custom code section using**
  ```
  #pragma CODE_SECTION(dotp, “critical”);
  int dotp(a, x)
  ```

- **Use the compiler’s –mo option**
  - -mo creates a subsection for each function
  - Subsections are specified with “:”
    ```
    #pragma CODE_SECTION(dotp, “.text:_dotp”);
    ```

### Making Custom Data Sections

- **Make custom named data section**
  ```
  #pragma DATA_SECTION (x, “myVar”);
  #pragma DATA_SECTION (y, “myVar”);
  int x[32];
  short y;
  ```

You will have to create new sections to keep critical code and data on-chip and other code and data off-chip.

**Hint:** Here is a little rule of thumb: “Create a new section for any code or data that must be placed in a specific memory location.”
**What is the “.far” Section?**

Rather than type in the whole DATA_SECTION pragma, if all you want to do is create a second data section, you can use the `far` keyword. Shown below are three different ways to create a variable `m` in the `.far` section.

### Special Data Section: “.far”

- `.far` is a **pre-defined** section name
- Three cycle read (pointer must be set before read)
- Add variable to `.far` using:
  1. Use DATA_SECTION pragma
     ```c
     #pragma DATA_SECTION(m, "far")
     short m;
     ```
  2. Far compiler option
     ```shell
     -ml
     ```
  3. Far keyword:
     ```c
     far short m;
     ```

No matter how you create additional data sections, they will always be accessed using far addressing (MVKL/MVKH). Only .bss is ever accessed with the near addressing optimization (global Data Pointer).


**Link Custom Sections**

Recall that the CCS Memory Manager provided drop down boxes to aid with placing the compiler and Bios created sections. Unfortunately, there isn’t a way for TI to know what section names you might create, thus there are no drop-down boxes for custom section placement.

Rather, you must create your own linker command file, as shown below.

A few points:

1. Second, using the `SECTIONS` descriptor, list all the custom sections you have created and direct them into a MEM object. Each line “reads”:

   ```
   myVar : > SDRAM
   critical: > IRAM
   .text:_dotp: > IRAM
   ```

   To learn more about the `SECTIONS` directive, or linking in general, please refer to *TMS320C6000 Assembly Language Users Guide* (SPRU186).

2. You should not specify a section in both the Configuration Tool and your own linker command file.

3. You shouldn’t use the same label for a section name as you did for a label in your code. In other words, don’t put variable `y` into section “y”.
4. Specifying link order

If you have more than one linker command file, how do you specify the order they are executed?
If you are concerned that you might forget a custom-named section (or a team member might create one without telling you), the \-w linker option can warn you of unspecified sections:
Using Local Variables

Whenever a new function is encountered, its local variables are automatically created on the software stack. Upon exiting the function, they are deleted from the stack. While most folks today call them “local” variables, they often used to be called “auto” variables. (A fitting name in that they are automatically allocated and deallocated from memory as they’re needed.)

Linking the software stack (.stack) into on-chip memory – and using local variables – can be an excellent way to increase on-chip memory efficiency … and performance.
**Everything you wanted or didn’t want to know about the stack**

Why learn about the stack? It is important to learn about the stack so you can trace what the compiler is doing, write assembly ISRs (Interrupt Service Routines), and because engineers want to know or think they need to know about the stack. So, here it goes!

The C/C++ compiler uses a stack to:
- Save function return addresses
- Allocate local variables
- Pass arguments to functions
- Save temporary results

The run-time stack grows from the high addresses to the low addresses. The compiler uses the B15 register to manage this stack. B15 is the stack pointer (SP), which points to the next unused location on the stack.

The linker sets the stack size to a default of 1024 bytes. You can change the stack size at link time by using the –stack option with the linker command. The actual length and location of the stack is determined at link time. Your link command file can determine where the .stack section will reside. The stack pointer is initialized at system initialization.

If arguments are passed to a function, they are placed in registers or on the stack. Up to the first 10 arguments are passed in even number registers alternating between A registers and B registers starting with A4, B4, A6, B6, and so on. If the arguments are longs, doubles, or long doubles, they are placed in register pairs A5:A4, B5:B4, A7:A6, and so on.
Any remaining arguments are placed on the stack. The stack pointer (SP) points to the next free location. This is where the eleventh argument and so on would be placed. Arguments placed on the stack must be aligned to a value appropriate for their size. An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int. An argument that is a float is passed as double if it has no prototype declared. A structure argument is passed as the address of the structure. It is up to the called function to make a local copy.

**Sidebar: How to PUSH and POP Registers**

*How would you PUSH “A1” to the stack?*

```
STW  A1, *SP--[1]
```

*How about POPing A1?*

```
LDW  *++SP[1], A1
```

---

**Using the Stack in Asm**

*Using the Stack in Assembly*

*Example:*

```
; PUSH nine registers -- “A0” thru “A8”
SP .equ B15
STW  A0, *SP--[10] ;
STW  A1, *+SP[9]
STW  A2, *+SP[8]
STW  A3, *+SP[7]
STW  A4, *+SP[6]
STW  A5, *+SP[5]
STW  A6, *+SP[4]
STW  A7, *+SP[3]
STW  A8, *+SP[2]
```

*New SP* | 8Byte boundary
--- | ---
A8 | 8Byte boundary
A7 | 8Byte boundary
A6 | 8Byte boundary
A5 | 8Byte boundary
A4 | 8Byte boundary
A3 | 8Byte boundary
A2 | 8Byte boundary
A1 | 8Byte boundary
A0 | 8Byte boundary

- Only move SP to 8-byte boundaries
- Move SP (to create a local frame), then Use offset addressing to fill-in PUSHed values
- May leave a small “hole”, but alignment is critical
Using the Heap

When the term *dynamic memory* is used, though, most users are referring to the heap.

In addition to using a stack, C compilers provide another block of memory that can be user-allocated during program execution (i.e. at runtime). It is sometimes called System Memory (.sysmem), or more commonly, the *heap*. 

---

### Dynamic Memory

**Using Memory Efficiently**

3. **Local Variables**
   - If stack is located on-chip, all functions can use it

4. **Use the Heap**
   - Common memory reuse within C language
   - A Heap (i.e. system memory) allocate, then free chunks of memory from a common system block
Here is an example using dynamic memory; in fact, it provides a good comparison between using traditional static variable definitions and their dynamic counterparts.

```
#define SIZE 32
int x[SIZE]; /*allocate*/
int a[SIZE];
x={...}; /*initialize*/
a={...};
filter(...); /*execute*/
```

```
#define SIZE 32
x=malloc(SIZE);
a=malloc(SIZE);
x={...};
a={...};
filter(...);
```

```
Free
```

```
free(a);
free(x);
```

◆ High-performance DSP users have traditionally used static embedded systems
◆ As DSPs and compilers have improved, the benefits of dynamic systems often allow enhanced flexibility (more threads) at lower costs

malloc() is a standard C language function that allocates space from the heap and returns an address to that space.

The big advantage of dynamic allocation is that you can free it, then re-use that memory for something else later in your program. This is not possible using static allocations of memory (where the linker allocates memory once-and-for-all during program build).
*** this page is __________ ***
**Multiple Heaps**

Assuming you have infinite memory (like most introduction to C classes assume), one heap should be enough. In the real world, though, you may want more than one. For example, what if you want both an off-chip and an on-chip heap.

Just as we discussed earlier with *Multiple Sections* for code and data, multiple heaps allows you to target critical elements on-chip, while less critical (or larger ones) can be allocated off-chip.
While standard C compilers do not provide multiple heap capability, TI’s DSP/Bios tools do. When creating MEM objects, you have the option to create a heap in that memory space. Just indicate you want a heap (with a checkmark) and set the size. From henceforth, you can refer to this specific heap by its MEM object name.

Alternatively, if you don’t want to use the MEM object name to refer to a heap you can define a separate identification label.
**Using MEM_alloc**

Q: If standard C doesn’t provide multi-heap capabilities, how would the standard C functions like malloc() know which heap to use?

A: They can’t know.

Solution: Use the DSP/BIOS MEM_alloc() function as opposed to malloc().

<table>
<thead>
<tr>
<th>Standard C syntax</th>
<th>Using MEM functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define SIZE 32</td>
<td>#define SIZE 32</td>
</tr>
<tr>
<td>x = malloc(SIZE);</td>
<td>x = MEM_alloc(IRAM, SIZE, ALIGN);</td>
</tr>
<tr>
<td>a = malloc(SIZE);</td>
<td>a = MEM_alloc(SDRAM, SIZE, ALIGN);</td>
</tr>
<tr>
<td>x = {...};</td>
<td>x = {...};</td>
</tr>
<tr>
<td>a = {...};</td>
<td>a = {...};</td>
</tr>
<tr>
<td>filter(...);</td>
<td>filter(...);</td>
</tr>
<tr>
<td>free(a);</td>
<td>MEM_free(SDRAM, a, SIZE);</td>
</tr>
<tr>
<td>free(x);</td>
<td>MEM_free(IRAM, x, SIZE);</td>
</tr>
</tbody>
</table>

As you can see, there is also MEM_free() to replace free(). Additional substations can be found in the DSP/BIOS library.
Using BUF

While using dynamic memory via the heap is advantageous from a memory reuse perspective, it does have its drawbacks.

Heap drawbacks:
- Allocation calls (i.e. malloc) are non-deterministic. That is, each time they are called they make take longer or shorter to complete.
- The allocation functions are non-reentrant. For example, if malloc() is called while a malloc() is already running (say, it was called in a hardware interrupt service routine), the system may break.
- Heap allocations are prone to memory fragmentation if many malloc's and free's are called.

BUF solves these problems by letting users create pools of buffers that can then be allocated, used, and set free.

BUFF Concepts

- Buffer pools contain a specified number of equal size buffers
- Any number of pools can be created
- Buffers are allocated from a pool and freed back when no longer needed
- Buffers can be shared between applications
- Buffer pool API are faster and smaller than malloc-type operations
- In addition, BUF_alloc and BUF_free are deterministic (unlike malloc)
- BUF API have no reentrancy or fragmentation issues
Creating a BUF
1. right click on BUF mgr
2. select “insert BUF”
3. right click on new BUF
4. select “rename”
5. type BUF name
6. right click on new BUF
7. select “properties”
8. indicate desired
   • Memory segment
   • Number of buffers
   • Size of buffers
   • Alignment of buffers
   • Gray boxes indicate effective pool and buffer sizes
Memory Overlays

Another traditional method of maximizing use of on-chip memory is to overlay code and data. (You could even substitute the term overlap for overlay.) While each exists on its own externally, they run from the same overlayed locations, internally.

With overlays, each code or data item must reside in its own starting location. The TI tools call this its load location, because this is what is downloaded to the system (when using the CCS Load Program menu item, or when you download to an EPROM via an EPROM programmer).

During program execution, your code must copy the overlayed data or code elements into their run location. This is where the program expects the information to reside when it is used (i.e. when the overlayed function is called, or the overlayed data elements are accessed). The linker resolves all your code/data labels (i.e.symbols) to the runtime addresses.

How do you implement overlays, follow these 3 steps …
**Implementing Overlays (code overlay example)**

1. **Create a section for each item you want to overlay.**

   For example, if you wanted two functions to be overlayed, create them with their own sections.

   ```c
   #pragma CODE_SECTION(fir, "FIR");
   int fir(short *a, ...)
   #pragma CODE_SECTION(iir, "myIIR");
   int iir(short *a, ...)
   ```

   We arbitrarily chose the section names .fir and myIIR.

2. **Create your own linker command file (as discussed earlier for Multiple Sections).**

   Earlier we put something like this into our SECTIONS part of the linker command file.

   ```
   .bss :> IRAM
   ```

   This could be re-written as:

   ```
   .bss:  load = IRAM, run = IRAM
   ```

   In the case of our overlayed functions, though, we don’t want them to be loaded-to and run-from the same locations in memory, therefore, we might try something like:

   ```
   .fir:   load = EPROM, run = IRAM
   myIIR:  load = EPROM, run = IRAM
   ```

   In this case, they are both loaded into EPROM and Run from IRAM. The problem is that the linker assigns different **run** addresses for both functions. But, we wanted them to share (i.e. overlap) their run addresses. How can we make this happen?

   Use the linker’s UNION command. The union concept is similar to that of creating union types in the C language. In our case, we want to tell the linker to put the run addresses of the two functions in union.

   ```
   UNION run= IRAM
   {
     .fir:   load = EPROM
     myIIR:  load = EPROM
   }
   ```

   This then, allocates separate load addresses for each function, while providing a single run address for both functions.

   **Note:** To set separate load and run addresses for pre-defined BIOS and Compiler sections, there is an additional tabbed page in the CCS Config Tools Memory Section Manager dialog.
3. **Last, but not least, you must copy the code from its original location to its runtime location.** Before you run each function you must force the code (or data, in a data overlay) to be copied from its load addresses to its run addresses. When using the Copy Table feature of the linker, copying code from its original location is quite easy.

```c
#include <cpy_tbl.h>
extern far COPY_TABLE fir_copy_table;
extern far COPY_TABLE iir_copy_table;
extern void fir(void);
extern void iir(void);
main()
{   copy_in(&fir_copy_table);
    fir();
    ...

    copy_in(& iir_copy_table);
    iir();
    ...
}
```

The `copy_in()` function is a simple wrapper around the compiler’s `mem_copy()` function. It reads the table description created by the “table” feature of the linker and uses it to perform a `mem_copy()`.

From a performance standpoint, though, you are better off using the DMA or EDMA hardware peripherals. These hardware peripherals can be easily used to copy these tables by using the `DAT_copy()` function from TI’s Chip Support Library (CSL).

**Overlay Summary**

- First, create a section for each function
- In your own linker cmd file:
  - `load`: where the fxn resides at reset
  - `run`: tells linker its runtime location
  - `UNION` forces both functions to be runtime linked to the same memory addresses (ie. overlayed)
- You must move it with CPU or DMA

```c
myCode.C

```

```c
#pragma CODE_SECTION(fir, “.FIR”);
int fir(short *a, ...)

#pragma CODE_SECTION(iir, “myIIR”);
int iir(short *a, ...)
```

```c
myLnk.CMD

#Pragma CODE_SECTION(fir, “.FIR”);
int fir(short *a, ...)

#Pragma CODE_SECTION(iir, “myIIR”);
int iir(short *a, ...)

SECTIONS
{   .bss:  IRAM /*load & run*/

UNION  run = IRAM
{   .FIR : load = EPROM
    myIIR: load = EPROM
}
```
**Using Copy Tables**

An easy way to generate the addresses required for overlays is to use copy tables.

```c
#include <cpy_tbl.h>
extern far COPY_TABLE fir_copy_table;
extern far COPY_TABLE iir_copy_table;
extern void fir(void);
extern void iir(void);
main()
{  copy_in(&fir_copy_table);
   fir();
   ...
   copy_in(&iir_copy_table);
   iir();
   ...
}
```

* copy_in() provides a simple wrapper around mem_copy().
* Better yet, use the DMA hardware to copy the sections; specifically, the DAT_copy() function.
Copy Table Header File

/**************************************************************************/
/* cpy_tbl.h */
/* Specification of copy table data structures which can be automatically */
/* generated by the linker (using the table() operator in the LCF). */
/**************************************************************************/
/* Copy Record Data Structure */
/**************************************************************************/
typedef struct copy_record
{  unsigned int load_addr;
    unsigned int run_addr;
    unsigned int size;
} COPY_RECORD;
/**************************************************************************/
/* Copy Table Data Structure */
/**************************************************************************/
typedef struct copy_table
{  unsigned short rec_size;
    unsigned short num_recs;
    COPY_RECORD recs[1];
} COPY_TABLE;
/**************************************************************************/
/* Prototype for general purpose copy routine. */
/**************************************************************************/
extern void copy_in(COPY_TABLE *tp);

Overlays can be very useful, but they’re also tedious to setup. Isn’t there an easier way to get the advantages of overlays? …
Cache

Data and program caching provides the benefits of memory overlays, without all the hassles.

Since modern C6000 devices have both data and program cache hardware, this is the easiest method of overlaying memory (and hence, most commonly used).

Rather than discuss cache in detail here, the next chapter is dedicated to this topic.
Summary

You may notice the order in the summary is a bit different from that which we just discussed the topics. While introducing them to you, we wanted to build the concepts piece-by-piece. In real life, though, as you design your system you will probably want to employ them in the following order.

Summary: Using Memory Efficiently

- You may want to work through your memory allocations in the following order:
  1. Keep it all on-chip
  2. Use Cache (more in Ch 15)
  3. Use local variables (stack on-chip)
  4. Using dynamic memory (heap, BUF)
  5. Make your own sections (pragma’s)
  6. Overlay memory (load vs. run)

- While this tradeoff is highly application dependent, this is a good place to start

For example,

1. If you can get everything on-chip, you’re done.

2. If it won’t all fit, you might try enabling the cache. If your system meets its real-time deadlines, you’re now done.

3. In most cases, you’ve probably already used local variables whenever possible. So this one is probably a ‘given’.

4. If you’ve enabled the cache and still need to tweak the system for performance, you might try to using dynamic memory
   … or one of the remaining options.

The advantage to the top 4 methods is that they can all be done from within your C code. The remaining two require a custom linker command file (or modification of your .cmd file). (Not difficult, but one more thing to manage.)