Memory Allocation Techniques in System with Dynamic Swapping of Application Codes

Oh, Hong Lye *SC Field Applications, Texas Instruments Singapore*

ABSTRACT

This application report illustrates how to manage the code/data in a system that performs dynamic swapping of routines on demand. This is process is achieved by following a simple setup procedure. This method is one of many ways to implement a dynamic swapping of applications.

The examples given in this report are simplified versions of a working code and are specific to C55x™ family of DSPs. Using these examples on platforms other than the C55x may require further modifications.

Contents

1 Introduction

When memory becomes a limitation in developing a system, external memory must be added. There are various forms of external memory:

- Volatile asynchronous (e.g. SRAM)
- Volatile synchronous (e.g. SDRAM)
- Non-volatile asynchronous (e.g. Flash), etc.

The speed of these types of memory will not be as fast as that of the internal memory of today's DSP, which typically runs at speed beyond 100MHz. In order to meet real-time requirement, critical codes must be swapped into the internal memory when required.

2 Basics

Before illustrating an example of dynamic swapping of applications, it is important to be aware of some basic information that explains how TI's Assemblers, C compilers, and linkers handle data/code.

2.1 Default Code/Data Sections

If a user does not specify any #pragma in the code, the C Compiler allocates the data/code in the following manner:

.text	=> Code
.bss	=> Global and Static Variables
.const	=> Global and Static constant variables
.switch	=> switch statement tables
.cinit	=> Tables for explicitly initialized global and static variables
.cio	=> CIO buffers (used in printf, scanf and other I/O functions)
.stack	=> Primary Stack memory
.sysstack	=> Secondary (System) Stack memory
.sysmem	=> Heap memory (for malloc allocation)

Another section, .data, is not used by the C Complier, but may be used by the Assembler to store initialized data.

For detailed information explaining the above sections, please refer to the *Optimizing C/C++ Compiler User Guide* (SPRU281C).

2.2 Assembler/Compiler directives

As stated earlier, a user can allocate data/code to user-defined sections. To allocate data/code, use the following examples:

- Assembler

  For initialized sections:

  .sect “user_sections”

  For un-initialized sections,

  .usect “user_section”, section_size

- C Compiler

  The C Compiler uses two pragmas for Code and Data sections.

  For example:

```
#pragma CODE_SECTION (func, "user_section”) ;
int func(int x, int y)
{
   …
}
#pragma DATA_SECTION (x, "user_section”) ;
int x[100];
```

  There are instances where data sections must be aligned to specific boundaries. Alignment can be achieved by using the DATA_ALIGN pragma.

  For example,

  To align x array declared above to 32-bit (double word) boundary, use:

```
#pragma DATA_ALIGN(x, 2);
```

2.3 Linker Directives

Two linker directives that are used to enable dynamic swapping are:

- UNION Directive

  UNION directive allows the user to create multiple sections with same run-time address. The example below illustrates how three user sections could be linked to run at the same start address during run-time, while having a different load address:

```
UNION: run = SARAM
{
.user_section1 : load = SDRAM
.user_section2 : load = SDRAM
.user_section3 : load = SDRAM
}
```

  The Linker does not accept load allocation for UNION e.g. UNION:

  `run = SARAM, load = SDRAM.`

- GROUP Directive

  GROUP directive is used to allocate several sections contiguously in memory:

```
GROUP: run = SARAM
{
user_section1    : load = SDRAM
user_section2    : load = SDRAM
}
```

  In above example, both load and run address of user_section1 and user_section2 are allocated contiguously in SDRAM and SARAM respectively.

For more information on the use of directives and their various variants, please refer to the *Assembly Tools User Guide* (SPRU280D).

3 Sample Program

In this section, a simple project is provided to aid the user with the concepts of dynamic code/data swapping.

3.1 Software System Block Diagram

The diagram below illustrates the target system:

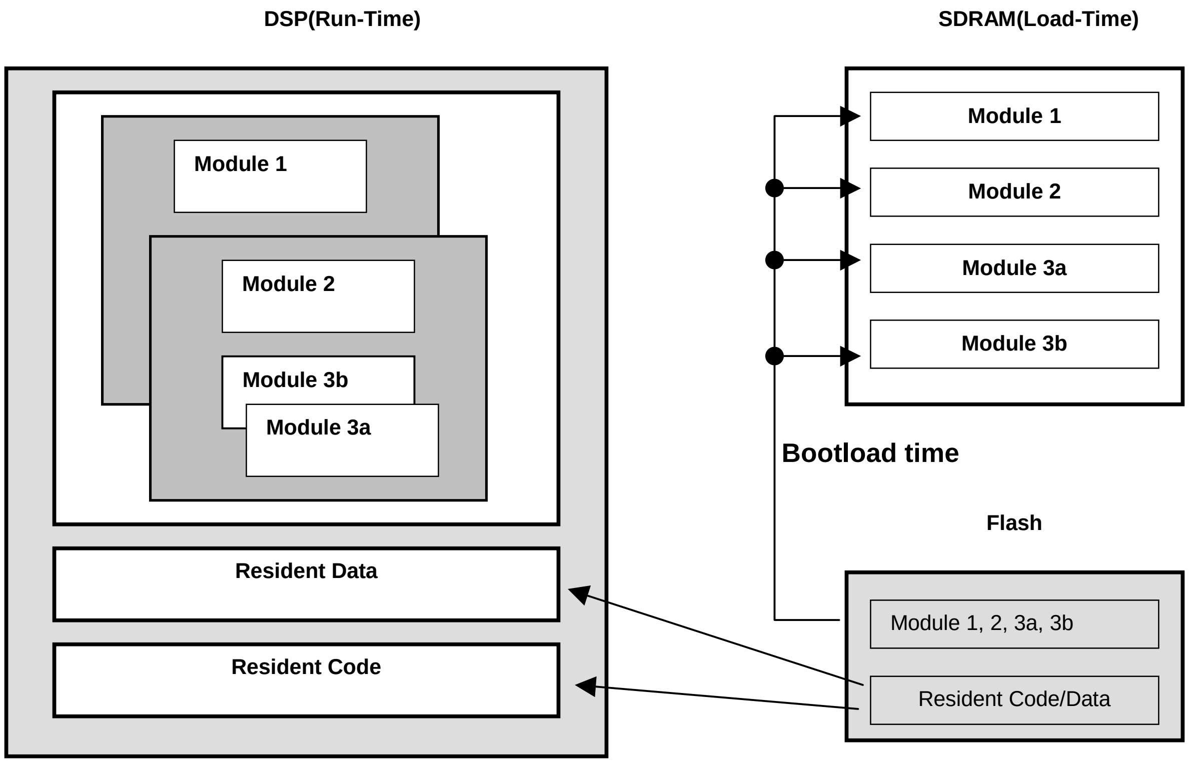

Figure 1. Software System Block Diagram

The system above illustrates routines that share run time addresses, or utilize the same physical memory within the DSP. The resident code is boot loaded from non-volatile memory, (i.e. Flash). The code used to manage this transfer resides in this section.

3.2 Considerations

- General

  Before you begin with the sample project, note which sections can be swapped in/out dynamically during run-time:

  – All code sections can be swapped.

  – For data sections, those that can be swapped are:

    .const

    .data

    Any user defined section containing data that are not initialized during code boot-up.

    .bss can be swapped provided there are no global or static variables that are declared with an initial value. (i.e. int x=3)

  – Sections that can't be swapped are:

    .cinit

    Any user defined sections containing global or static variables with an initial value.

  – Stack specific to a particular piece of code i.e. TSK stacks can be swapped together with the code, but the stack must be initialized to a default signature when working under DSPBIOS environment. Failure to initialize the stack may result in the DSPBIOS APIs, which make use of these signature stamps, to not function properly.

- .bss/.const

  Based on the above considerations, if .bss is to be swapped, modifications to the existing .code must be done. The objective is to have a .bss with no initialized variables. This is achieved by avoiding the use of initialized global variables. It is suggested to avoid using them altogether.

  If initialized global/static variables cannot be avoided, place these variables in user defined sections using pragma directives (C-compiler), or sect directives (Assemblers) described in section 2.2. For data such as look-up tables, etc, declare them as .const data type (the value doesn't change). This insures that they are located to a .const section that can be swapped (initialization is done within the code execution and not during C environment initialization).

- .cinit

  This section cannot be swapped because it is required during C environment initialization. From a different perspective, this section need not be swapped since it is only used during boot-up. After which, the space it occupies could be re-used. If you are using less expensive memory i.e. SDRAM, consider using .cinit in its place. However, performance impact is reduced, since this is only performed once during C code boot-up.

3.3 Obtaining the Load Address

Obtaining load address of libraries or C modules can be problematic. Load addresses are obtained using ".label" directive as directed in the *TMS320C55xx Assembly Language Tools User's Guide* (SPRU280D). For modules where .label directive cannot be easily inserted, workarounds may be necessary.

One solution is to create a separate .asm file for each module/group you intend to swap. The .label directive can be inserted into this .asm file. This helps track the load address of the module/group concerned. The format of data and code varies slightly, as shown below:

```
========================= Start of <mod1_scode.asm> ========================

.def    _mod1_code_load;

        .sect ".mod1_code_start"

        ret

        .label _mod1_code_load


========================= End of <mod1_scode.asm> ========================



========================= Start of <mod1_sdata.asm> ========================


        .def   _mod1_data_load;

        .sect ".mod1_data_start"

        .word  0

        .label _mod1_data_load;


========================= End of <mod1_sdata.asm> =========================
```

The op code and data used are arbitrary. Any code or data can be used.

> **Note:** When using C55x, it is suggested that an op code with an even number of bytes be used. This prevents problems with re-aligning section boundaries during development.

Using this workaround allows the use of any module, without the need to track their load addresses. Section 3.4 discusses the use of this workaround in linker command files.

3.4 Linker Command File

The linker command file to implement the system illustrated in section 3.1 is shown below:

```
========================== Start of Linker Command File =========================

MEMORY
{
       DARAM : origin = 00100h, length = 0FF00h
       SARAM : origin = 20000h, length = 20000h
       SDRAM : origin = 40000h, length = 20000h
}

SECTIONS
{
       /* Data */
       UNION:
       {
              /* Module1 */
              GROUP:
              {
                     mod1_data:
                     {
                            \project\obj\mod1_sdata.obj (.mod1_data_start)
                            _mod1_data_run = .;
                            \project\lib\mod1.lib (.data, .const, .bss)
                            _mod1_data_length = . - _mod1_data_run;
                     } load = (SDRAM align 4)

                     .uninitialised_user_sect

              }

              /* Module2 and 3 */
              GROUP:
              {
                     mod2_data:
                     {
                            \project\obj\mod2_sdata.obj (.mod2_data_start)
                            _mod2_data_run = .;
```

```
                          \project\lib\mod2.lib (.data, .const, .bss)
                          _mod2_data_length = . - _mod2_data_run;
                    } load = (SDRAM align 4)

                    UNION:
                    {
                          mod3a_data:
                          {
                          \project\obj\mod3a_sdata.obj (.mod3a_data_start)
                          _mod3a_data_run = .;
                          \project\lib\mod3a.lib (.data, .const, .bss)
                          _mod3a_data_length = . - _mod3a_data_run;
                    } load = (SDRAM align 4)

                    mod3b_data:
                    {
                          \project\obj\mod3b_sdata.obj (.mod3b_data_start)
                          _mod3b_data_run = .;
                          \project\lib\mod3b.lib (.data, .const, .bss)
                          _mod3b_data_length = . - _mod3b_data_run;
                    } load = (SDRAM align 4)

              }
       }
}> DARAM /* run-time allocation */ /* DATA UNION */

/* CODE */
UNION:
{
       /* Module1 */
       mod1_code:
       {
             \project\obj\mod1_scode.obj (.mod1_code_start)
             _mod1_code_run = .;
             \project\lib\mod1.lib (.text, .user_code_sect1)
             _mod1_code_length = . - _mod1_code_run;
       } load = (SDRAM align 2)
       /* Module2 and 3 */
       GROUP:
       {
              mod2_code:
              {
                     \project\obj\mod2_scode.obj (.mod2_code_start)
                     _mod2_code_run = .;
                     \project\lib\mod2.lib (.text)
                     _mod2_code_length = . - _mod2_code_run;
                     } load = (SDRAM align 2)

                     UNION:
                     {
                            mod3a_code:
                            {
                                   \project\obj\mod3a_code.obj (.mod3a_code_start)
                                   _mod3a_code_run = .;
                                   \project\lib\mod3a.lib (.text)
                                   _mod3a_code_length = . - _mod3a_code_run;
                            } load = (SDRAM align 2)

                            mod3b_code:
                            {
                                   \project\obj\mod3b_scode.obj (.mod3b_code_start)
                                   _mod3b_code_run = .;
                                   \project\lib\mod3b.lib (.text)
                                   _mod3b_code_length = . - _mod3b_code_run;
                            } load = (SDRAM align 2)

                     }
              }
       }> SARAM /* CODE UNION */

       /* Resident code/data */
       .resident_code :> SARAM
       .resident_data  :> DARAM

}/*SECTIONS*/
```

```
========================= End of Linker Command File =======================
```

Some points to remember:

- To avoid potential issues with byte/word boundary confusion, always align code sections to even byte boundaries. This is necessary because the C55x C-Compiler deals in word level and it is often tedious handling code sections with odd-byte boundaries.
- For data sections, it is advisable to align code sections to 4 byte boundaries as a solution for cases where there's a double word variable at the front of the data section.
- Notice that un-initialized sections are included into the data section meant for swapping. These sections are not required. They exist for maintenance purposes and/or making full use of spare memory (the amount of memory a UNION will occupy is equal to its largest member). i.e., there is a ".uninitialised_user_sect" declared in the first group of the first UNION, just after mod1_data section.
- The Data and Code are separated into two big UNIONs. This is not required, but they can be placed in the same UNION if it is required.

  Note: Placing both code and data together in a user specified section will result in a linker error.

3.5 Swapping Code

This section gives a simple example on how code swapping could be done for module 1. The same code can be applied to swap the other modules in the system.

```
========================= Start of Sample Code ===============================
extern void *mod1_code_length;
extern void *mod1_code_load;
extern void *mod1_code_run;
extern void *mod1_data_length;
extern void *mod1_data_load;
extern void *mod1_data_run;

int swap_code()
{
  int *src_ptr, *dest_ptr;
  int *dsrc_ptr, *ddest_ptr;
  unsigned int count, dcount;

  /* Swap Module 1 Code section in */
  src_ptr  = (int *)((unsigned long)mod1_code_load>>1);
  dest_ptr = (int *)((unsigned long)mod1_code_run >>1);
  count = ((unsigned int)mod1_length+1)>>1;
  memcpy(dest_ptr, src_ptr, count);

  /* Swap Module 1 Data section in */
  dsrc_ptr  = (int *)((unsigned long) mod1_data_load);
  ddest_ptr = (int *)((unsigned long)mod1_data_run);
  dcount = ((unsigned int)mod1_data_length+1)>>1;
  memcpy(ddest_ptr, dsrc_ptr, dcount);
}
========================== End of Sample Code ===============================
```

A few points to remember:

- The data type for the labels created in linker command files is declared as (void *).
- The address and length information of code sections are in bytes. (length+1)>>1 is used to compute the number of words to transfer (C-compiler doesn't work in bytes).
- Note that memcpy() can only transfer up to a maximum of 64K words. Beyond that, the user must create a separate routine.
- The address information of data sections is in bytes, while the length computed is shown in words.

4 Conclusion

Through usage of the UNION and GROUP linker directives, it is possible to create a structure that allows dynamic swapping of applications during run-time. Having this capability assures that the product can support many features without depleting limited internal memory too soon.

A successful implementation of such a system may also depend on the way the individual modules are written. The modules should use as few global and static variables as possible. They should be programmed on a predefined memory and MIPS budget so that they can be easily integrated easily without the need to repeatedly swap modules.