Software Coding Guidelines for ’C5x Developers

Application Report

Mansoor A. Chishtie
Digital Signal Processing Applications — Semiconductor Group

SPRA149
October 1994

Introduction

This report furnishes guidelines to DSP application software developers on how to organize and structure their software to facilitate its maintenance and ease its porting to any custom-defined DSP hardware platform. The model DSP platform used here is a PCMCIA-based ’C5x DSP card with an external connector for an analog interface. (For details on the card, see the preceding report, *The PCMCIA DSP Card: An All-in-One Communications System.)*

The guidelines in this report should be used in conjunction with the following documents:

- *TMS320 Fixed-Point DSP Assembly Language Tools User’s Guide*
- *TMS320C2x/C5x Optimizing C Compiler User’s Guide*

Hardware Platform Overview

A model DSP hardware platform that will be used as a test and demonstration bed for various DSP applications consists of a PCMCIA type II card with an embedded 25-ns ’C51 digital signal processor and memory. This card complies with the PCMCIA I/O card specifications. This card is capable of running in either standard or smart mode. In standard mode, the DSP is nonfunctional, and the card behaves like any other PCMCIA memory card. The host can switch the card into smart mode by writing a predetermined *signature* sequence to a memory location. In smart mode, the embedded DSP is active and executes code from the card memory. Memory available on the first version of this card is 192K words, mapped as multiple 64K pages in data and program spaces.

There are two standard methods for data transfer and command handshake between the host and the DSP: the shared PCMCIA memory and a pair of dual-ported memory-mapped registers. The shared PCMCIA memory, when properly initialized by a PCMCIA card controller, acts like extended memory to the PC memory map. This is the preferred way of transferring large blocks of code or data to and from the embedded DSP. Note that this mode of access may impose additional time constraints on the real-time execution of an application because the DSP halts while the PC is accessing the shared memory.

Both the host and the DSP can read or write to the dual-ported memory-mapped registers that provide the other host-DSP interface. Access to these registers does not affect the normal operation of the DSP or the host processor. Both sides can poll special bit flags or enable themselves to be interrupted whenever the other side accesses these registers. This register-based communication link is especially suited for sending commands and occasional data parameters to the other end. This feature should be fully utilized by applications to pass results back to the host and let the host apply real-time control functions (such as mode change, start, stop, etc.) to the applications.

For applications that require an analog interface to the outside world, a special connector is provided at the back end of the PCMCIA card; the connector can interface special peripherals to the DSP serial port or bit I/O. Additionally, digital data can be sent over the serial link from an external processor or controller. The connector also supports a TI JTAG emulator (XDS-510) that facilitates application software debug directly on the card.

This hardware platform overview is provided for illustration purposes only. The following discussion is equally applicable to any other ’C5x-based hardware platform.

Software Organization

It is strongly recommended that the following guidelines be observed to organize DSP application software. This will not only result in well-structured code, but it will also make the application easier to port to any other hardware platform.

Organize each software application as a collection of modules or files that belongs to one of the following categories:

- *Source Modules (*.c, *.asm):* C or assembly source code files should not define any global constants or macros.

- *Include Modules (*.h, *.inc):* All include files for C modules must use file extension *.h, and all such files for assembly modules must use extension *.inc. Include files should define all global constants, macros, or variable types. They should not allocate memory or define functions, because this prevents them from being included by multiple source files. All functions and variables that form part of the overall interface to a *.c or *.asm file should be declared in a *.h or *.inc file. This provides a convenient overview of the interface and allows the compiler or assembler to check for errors.

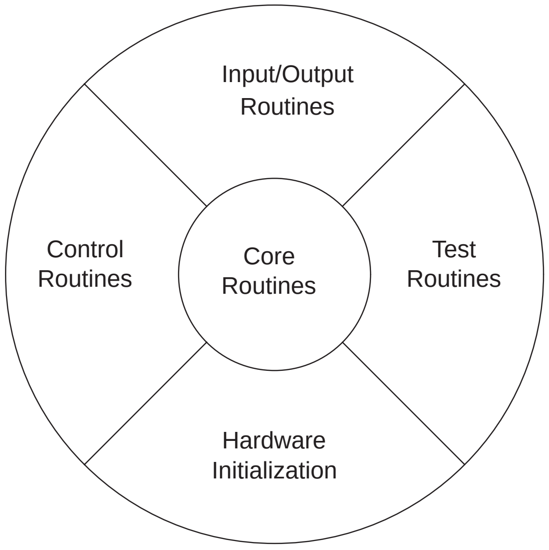
- *Linker Command File (*.cmd):* This command file is used by the TI COFF linker to link multiple modules into a single executable COFF output file.

- *Data Vectors (*.dat):* These files should contain only data to be used for tests or algorithms. There must not be any code in these data files. These files, if used, will probably be included or copied (.include or .copy directives) in other source files or assembled as stand-alone modules.

- *Make File (*.mak, *.prj):* It is strongly recommended that you maintain a project make file that checks for any out-of-date target files and builds them automatically. Note that both Microsoft and Borland make-file utilities use mutually compatible file syntax.

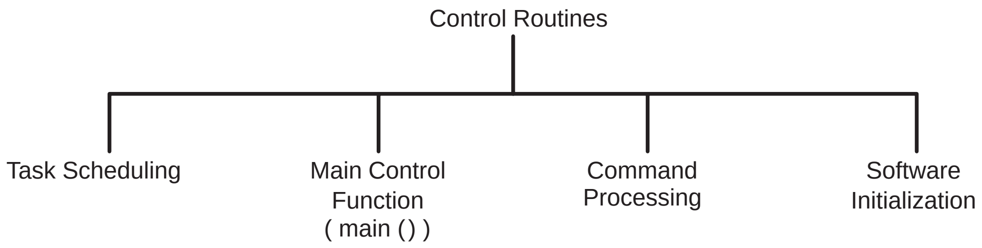
Organize source code files so that each file will fall under one of the categories shown in Figure 1:

Figure 1. Categories of Source Code Files

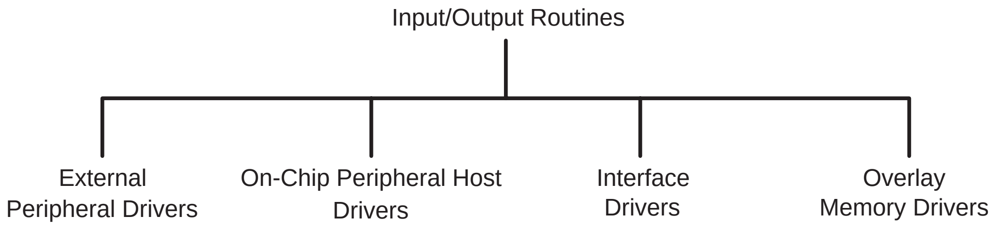


- *Core Routines:* Include all software modules that implement the core algorithm. These routines should be independent of hardware-specific implementations. The only target-specific information that these routines should contain is the knowledge of the target DSP processor, in case the modules are in assembly language. Developers of independent applications may want to group these routines into additional categories on the basis of their functionality.

- *Control Routines:* These routines consist of all software modules that implement control functions. These control functions may include a C-like main function for program flow control, task handling and scheduling functions, interrupt service routines that pass control to core routines, a command handler that interprets host commands, and routines that initialize variables and tables. Some of these modules may contain some hardware-specific information, but their primary task is to control the program flow. They must *not* handle any input/output functions or external peripheral accesses. Note that interrupt service routines (ISRs) that handle on-chip or external peripherals must not be grouped here. The intention is to keep any modifications to these routines at a minimum when the software is ported to a new platform.



- *Input/Output Routines:* These routines should handle all the input/output activities of the application, including accesses to any on-chip or external peripherals and I/O ports. As an example, DSP code that handles host communication protocol falls under this category. A serial port ISR and other functions that access an I/O-mapped external peripheral also belong to this category. It is recommended that each peripheral driver be arranged as one source file.



- *Hardware Initialization Routines:* In general, most nonhardware-specific initialization routines belong to the control routines category. However, since core routines must not have hardware-specific implementations, all functions that initialize external hardware such as external peripherals, host processor, etc., must be grouped separately. Note that these routines will differ from input/output routines in that they are invoked only once during system initialization.

- *Test Routines:* Application developers should provide a test procedure to verify functionality of their applications. This is especially important when an application is ported (or modified) to a different hardware platform. This test procedure can be in the form of a test program that calls different modules of an application separately to determine their integrity, or it can be in the form of input data vectors that can be processed by the application and output data vectors to be used for verification of the results.

Memory Organization

Proper memory organization is essential for application portability and maintenance. The following guidelines are mandatory:

- Addresses of data variables and tables should not be hard-coded. For example, you cannot use the .set directive to equate a label to an address. This is effectively a form of hard-coded memory allocation because variable addresses are determined during assembly time. The .usect, .sect, and other similar assembler directives should be used to allocate uninitialized and initialized variables. It is recommended that all variable definitions and allocations be done in separate files, as in the following examples:

```
Var_addr    .set   0800h                     **Hard-Coded Addr**
Var_addr    .usect ”Section_name”,1          **Addr Def. by Linker**
```

  If a peripheral is mapped to a unique address, then this mapping should be clearly identified in the linker command file.

- No assumption should be made about the type of COFF loader available to a host. In many cases, the host would not have access to a smart loader that can autoinitialize global variables during loading (similar to the –c option in the COFF linker). In other cases, an application can be preloaded in nonvolatile memory so that a loader is unnecessary. Therefore, an application should initialize all data variables during system initialization. One side effect of this restriction is that no initialized data can exist in data memory; all initialized tables and variables must be in program memory. They can be later copied to data memory, if necessary, by the software initialization module. This, however, implies that the total code size of an application will become larger than necessary. If the program size is getting unreasonably large because of this restriction, you can choose to ignore this restriction if your system loader can initialize data memory directly. In this case, all initialized data sections must be clearly identified in the linker command file.

- Avoid any restrictions on placement of variables and tables in memory, if possible. Occasionally, an application may require that restrictions be imposed on where a table can be placed in memory. This may happen because 1) a particular DSP feature (for example, bit-reversed addressing) demands it, or 2) it makes an algorithm implementation easier. Any such restriction should be clearly defined in the COFF linker command file in the form of extended comments.

- Global variables and local variables should be defined in separate sections. However, memory can be reused, and local variables of independent functions can occupy the same physical memory space when you use the GROUP and UNION linker directives (see the appendix for a sample linker command file).

- All code and data sections should be mapped to physical addresses during link time. In other words, the *linker command file should be the only module in which absolute addresses are defined.*

- If your application uses overlays or multiple memory pages, you should use the TI COFF linker syntax to define these overlays (see the appendix for an example linker command file). Additionally, you should write a driver module to be a part of the input/output routines that will handle the custom-defined memory overlay/page control implementation. This driver module should comply with the following restrictions:
  - The module must be located in on-chip memory. This restriction is intended to guarantee that the DSP will not be accessing off-chip memory when bank-switching occurs.
  - Due to pipelining of instructions by the DSP, the next three instructions following a bank-switch instruction can still access the previous bank. To avoid this, you must make sure that the three instructions immediately following a bank-switch must not access the address range that corresponds to the switched memory bank. Note that if this driver module is called as a subroutine, then a return (RET) instruction immediately after the bank-switch will guarantee that the switch has occurred before the DSP fetches instructions from the new bank:

```
Bank_Switch: ; bank switch routine
   ...
out *,PA0     ; switch in new memory bank
ret           ; return to new bank
```

Programming Guidelines

- Many DSP applications use mixed-mode (C and assembly) programming techniques to compromise between the need for efficient code and ease of programming. However, in some cases, an application may completely be written in DSP assembly language. In such cases, it is highly recommended that at least a dummy C main() function be written that simply transfers control to an assembly function. In this way, a basic C environment is automatically set up by main(), which leads to easier integration of any C functions in the future. If main() is the only C function in an application, then the rest of the functions need not adhere to C calling conventions.

- Many mixed-mode applications strictly follow the C convention for function calls, parameter passing, and variable allocation. However, you may need to avoid these constraints to efficiently implement some assembly-level functions. All such exceptions must be clearly identified and described in corresponding documentation. In some cases, when an assembly language function is called only by other assembly functions, context is not maintained across the function calls. These functions, although legal, must be clearly identified as non-C-callable functions to avoid any future maintenance problems.

- Self-modifying code should not be written. Such code is commonly used in interrupt vector tables (IVT), where one ISR can be patched for another during runtime. You can avoid this by using a software semaphore in ISR or by using an LAMM/BACC sequence to replace a more conventional B address sequence in IVT. The following interrupt vector table code example illustrates the use of an LAMM/BACC instruction to fetch the address of an ISR from a data memory location (in data page 0):

```
INT1:  lamm   INT1_Addr
       bacc
```

- For relocatable sections of code, do not use the .asect directive. Instead, use the runtime and load-time address options of the TI linker. This emphasizes our strategy of not allowing absolute addresses in assembly modules. Note that the .asect directive requires an absolute address to be specified as a parameter.

- Avoid using numerical constants as instruction parameters. Code listings are more readable when constants are replaced by meaningful labels. You can do this with the .set directive, as shown in the following example:

  replace:

```
add       #07FFFh
```

  with:

```
One_Q15   .set   07FFFh
add       #One_Q15
```

Source Code Documentation

- All source modules, whether in assembly or C, must maintain a modification history table that lists the date and time of each modification in chronological order, the person who made the change, and a brief description of the change.

- Line-by-line comments are highly recommended, especially for assembly language modules. All functions in a module, whether assembly or C, must clearly describe the implementation-specific details of the function.

- All functions should be preceded by a function header that gives the function description, input and output parameter lists, global variables used, a list of nested function calls, a list of functions that can call this function, and entry/exit conditions. Note that entry and exit conditions are especially important for assembly functions because processor context is not often maintained across function calls.

Appendix: A Sample Linker Command File for the ’C5x Card

The following linker command file is listed here to illustrate how to use the TI COFF linker syntax to define overlays and multiple code/data pages for the ’C5x PCMCIA card version 1.0. This command file would require minimum modifications to adapt to any ’C5x application running on this PCMCIA card.

```
f1.obj f2.obj f3.obj f4.obj f5.obj f6.obj
-o f.out
-m f.map
/***********************************************************************
PCMCIA ’C5x Card Memory Map: version 1.0
At reset, page 0 is in the ’C5x program space and page 1 is in data space.
If page 2 is enabled, it is dual-mapped in both program and data spaces. Each
application must carefully divide page 2 into two or more sections, and each
section must be considered as either program or data, but not both. In the
following example, the PRAM2 section is mapped as program, and the RAMEXT2
section is mapped as data, but this can be modified by an application.
The RAMSA and RAMDA memory blocks (in both page 0 and page 1) are defined as
overlays. This means that runtime addresses of multiple code and data
sections can be bound to these overlay sections. Note, however, that you must
copy any initialized section to an overlay area before it can be used.
All pages are 64K words in length.
          Program                     Data                       Program/Data
 0000 -------------         0000 -------------           0000 -------------
        |           |             | XXXXXXXXXX |                |           |
        |           |        0060 |-----------|                 |           |
        |   PRAM    |             |   RAMB2   |                 |   PRAM2   |
        |           |        007F |-----------|                 |           |
        |           |             | XXXXXXXXXX |                |           |
 2000   |-----------|        0100 |-----------|                 |           |
        |   RAMSA   |             |   RAMDA   |                 |           |
 2400   |-----------|        0500 |-----------|          2000   |-----------|
        |           |             | XXXXXXXXXX |                |           |
        |  RAMEXT   |        0800 |-----------|                 |  RAMTEXT2 |
        |           |             |   RAMSA   |                 |           |
 FE00   |-----------|        0C00 |-----------|                 |           |
        |   RAMDA   |             |   RAMEXT  |                 |           |
 FFFF -------------         FFFF -------------           FFFF -------------
           Page 0                     Page 1                      Page 2
***********************************************************************/
MEMORY
{
    page 0 : /* Program Only */
        PRAM  : origin = 00000h,  length = 02000h
        RAMSA : origin = 02000h,  length = 00400h  /* Overlay Section */
        RAMEXT: origin = 02400h,  length = 0DA00h
        RAMDA : origin = 0FE00h,  length = 00200h  /* Overlay Section */
    page 1 : /* Data Only */
```

```
        RAMB2 : origin = 00060h,  length = 00020h
        RAMDA : origin = 00100h,  length = 00400h  /* Overlay Section */
        RAMSA : origin = 00800h,  length = 00400h  /* Overlay Section */
        RAMEXT: origin = 00C00h,  length = 0F400h
    page 2 : /* Dual-Mapped in Program and Data */
        PRAM2  :origin = 00000h,  length = 02000h  /* Contains Code  */
        RAMEXT2:origin = 02000h,  length = 0E000h  /* Contains Data  */
}
SECTIONS
{
      PROG1:  load = PRAM     page 0
      {
              f1.obj(.text)
      }
      PROG2:  load = PRAM2    page 2
      {
              f2.obj(.text)
      }
      DATA1:  load = RAMEXT   page 1
      {
              f1.obj(.data)
      }
      DATA2:  load = RAMEXT2  page 2
      {
              f2.obj(.data)
      }
      UNION : run  = RAMSA    page 0          /* Overlay Section:     */
      {                                       /* f3 and f4 functions  */
              .text1 : load = RAMEXT page 0   /* will be copied and   */
              {                               /* run from RAMSA page0 */
                      f3.obj(.text)
              }
              .text2 : load = RAMEXT page 0
              {
                      f4.obj(.text)
              }
      }
      UNION : run  = RAMDA    page 0          /* Overlay Section:     */
      {                                       /* f5 and f6 functions  */
```

```
                .text3 : load = RAMEXT page 0   /* will be copied and   */
                {                               /* run from RAMDA page0 */
                        f5.obj(.text)
                }
                .text4 : load = PRAM    page 0
                {
                        f6.obj(.text)
                }
        }
        UNION : run  = RAMSA    page 1          /* Overlay Section:     */
        {                                       /* local variables of   */
                .bss1  :                        /* f3 and f4 functions  */
                {                               /* overlay each other   */
                        f3.obj(.bss)            /* in RAMSA page 1      */
                }
                .bss2  :
                {
                        f4.obj(.bss)
                }
        }
        UNION : run  = RAMDA    page 1          /* Overlay Section:     */
        {                                       /* local variables of   */
                .bss3  :                        /* f5 and f6 functions  */
                {                               /* overlay each other   */
                        f5.obj(.bss)            /* in RAMDA page 1      */
                }
                .bss4  :
                {
                        f6.obj(.bss)
                }
        }
}
```