Application Report
SPRA640A -May 2001

Programming and Debugging Tips for DSP/BIOS

Don S. Gillespie — *Software Development Systems*

ABSTRACT

DSP/BIOS provides the developers of DSP applications with a comprehensive set of tools for design and development of real time embedded applications. Due to the complexity and constraints of the target DSPs, there are many potential pitfalls for the developer, even with DSP/BIOS and Code Composer Studio™.

This note serves as a troubleshooting guide for developers in this environment. It discusses areas where potential problems can arise, strategies to deal with these issues, and helpful debugging tools within DSP/BIOS and Code Composer Studio. The document is organized with each main section devoted to an individual debugging or programming issue, without regard to specific hardware. Within each section are subsections which present the processor–specific and/or board-specific considerations of these issues.

The information was compiled from the Engineering staff at Texas Instruments who are experienced in the DSP/BIOS environment. The note focuses on the C6000™ and C5000™ processor families, and information is provided that is specific to each family. Example code is provided where appropriate.

Contents

List of Figures

1 Code Composer Studio Startup Issues

The *board–specific reset* function provided as a selection on the *Start–>Programs–>Code Composer Studio Cxxxx* menu of the PC desktop is a tool used to reset both the target board, and the emulation environment. This reset function should be executed before running Code Composer Studio to establish a good baseline in the hardware for your debugging session.

When Code Composer Studio is invoked, it runs a GEL file that initializes the target board. The contents of the *StartUp* routine in this GEL file are typically commented out because of the variance in memory configurations, and initialization requirements of the different DSPs and boards. One of the first things you should do is examine this file, and customize it to perform the desired initializations for your board and CPU. To determine the location and name of the GEL file being executed, right–click on the CCS desktop icon. Select *Properties*. The *Shortcut* tab will indicate the command line used to start Code Composer Studio. The GEL file is a parameter on this line. The file specified here also appears in the *Project* view within Code Composer, in the *GEL files* folder.

Once Code Composer Studio is running, you can perform a reset of the hardware by using one of the reset selections available on the GEL pulldown. They perform a reset not only of the CPU, but also of the peripherals on the board. The *Reset DSP* function on the *Debug* pulldown resets only the DSP, not the board peripherals.

You can experiment with startup settings by creating multiple shortcuts to Code Composer Studio, each specifying a different GEL initialization file on the command line. This may be useful if you are working with multiple target configurations.

If you are working in a multiprocessor debug environment, in any CCS 1.x version, specifying the GEL initialization file in the shortcut may cause problems, since all DSPs will receive the same initialization. If all DSPs in the system are the same, there is no problem; however, if some are C54x and some are C6x, their initialization requirements are different. In some cases, different model CPUs in the same family also have differing initialization requirements. In this type of environment, remove the GEL file specification from the shortcut. Start Code Composer Studio, and select *Load GEL* from the *File* pulldown. Once it is loaded, perform a reset of that DSP from the *GEL* pulldown.

1.1 C54x Considerations

For C54x targets, the default GEL file is *c54x.gel*. Edit this file and uncomment the contents of the *StartUp* routine. In the case of the C54x, three registers must be set up at initialization for the target to function properly:

- Processor Mode Status Register (PMST)
- Software Wait-state Register (SWWSR)
- Bank-switching Control Register (BSCR)

The SWWSR and BSCR must be properly set up in order to access external memory. The PMST contains values such as the interrupt vector page pointer and the OVLY bit, that are critical to correct operation. These registers must be set appropriately for the target board and DSP. The default GEL file is correctly configured for the Spectrum Digital C54xEVM board. If you are using a different board or C54x processor, you must customize the file.

Occasionally during development, the C54x EVM board will get into a state where the commanded resets will not work. Code Composer Studio will report errors accessing the board or the JTAG hardware, or with clearing breakpoints. When this occurs, exit Code Composer Studio, press the *reset* button on the EVM, and restart Code Composer Studio. If this does not correct the problem, cycling power may be necessary on either the XDS–510, the EVM board, or both.

1.2 C6x Considerations

The default GEL file for C6x targets varies according to the CCS version; it may be *init.gel* or *c6x.gel*. In either case, it is configured for the C6x EVM board. Save the file and exit Code Composer Studio. When you restart Code Composer Studio, the initialization will take place. It initializes the interface to the external memory on the board. Without this initialization, that memory will not be accessible to your application.

If you are working with a board other than the EVM6x, use the contents of this file as a template, and modify it according to the requirements of your board. See the specific board documentation and the *TMS320C6000 Peripherals Reference Guide,* literature number SPRU190, for more information.

2 Stacks

In any system, there are two basic concerns about stacks: overflow and corruption. DSP/BIOS provides several tools that monitor stack usage, and can flag overflow or corruption occurrences. These tools are discussed in this section.

In DSP/BIOS, there are three basic kinds of execution threads: hardware interrupt ISR (HWI), software interrupt (SWI), and task (TSK). There are two basic types of stacks in a system: the task stack, and the ISR stack. Each task has its own private stack, whereas SWIs and HWIs do not. The ISR stack is also known as the *system stack* in some documentation; in general, it is the common stack used by all interrupts. Its limitations are described in the hardware–specific sections. The role of the two stacks during context switch is also discussed in the hardware–specific sections.

2.1 Viewing Stacks

Stacks may be viewed in *Memory View* windows. Locating the stack in memory depends on the type of stack you wish to view.

The ISR stack is placed in its own section by the linker: *.stack*. Also, there are several symbols you can type in at the *Address* line of the *Memory Properties Window* to access the ISR stack.

```
GBL_stackend                    :start of the stack buffer
HWI_STKBOTTOM                   :start (BOTTOM) of the usable stack (starting stack
                                 pointer)
HWI_STKTOP = GBL_stackbeg       :the end (TOP) of the stack
```

The symbol, *GBL_stackend*, may be slightly different from the HWI_STKBOTTOM address. GBL_stackend defines the start of the stack buffer, whereas HWI_STKBOTTOM is the highest value the stack pointer may take. HWI_STKTOP and GBL_stackbeg both refer to the same value, the end of the stack buffer. Information about the ISR stack is also available in the *Kernel/Object View*, which is discussed later in this section.

Task stack information such as start and end addresses and maximum amount used are also available in the Kernel/Object View. Enter these addresses at the *Address* line of the *Memory Property Window* to view the stack. An additional way to view a statically allocated task's private stack is to enter *<taskname>$stack* at the *Address* line of the *Memory Property Window*.

This information, as well as most other stack–related data, can also be obtained from the TSK_stat DSP/BIOS API, which is discussed later in this section.

2.2 TSK_checkstacks()

TSK_checkstacks() is a DSP/BIOS API that examines task stacks for overflow and corruption. The actual syntax of the API call is:

```
Void TSK_checkstacks ( TSK_Handle oldtask, TSK_Handle newtask )
```

This API is intended to be called at context switch time; *oldtask* is the task you are switching *from* while *newtask* is the task you are switching *to*.

At task creation time, each task's stack is initialized by DSP/BIOS with a known value written to all locations in the stack. This value is the same for all task stacks, and is referenced within DSP/BIOS as *TRG_STACKSTAMP*. The value differs between processor families (see the hardware–specific sections for details).

TSK_checkstacks uses the task handles to obtain the location and size of the two stacks. It then checks the last location for TRG_STACKSTAMP. If TRG_STACKSTAMP is not at the last location, it has been overwritten. The nature of the problem depends on which stack failed the test. If the *oldtask* stack failed the test, then the *oldtask* stack overflowed, and the problem is within *oldtask*. If the *newtask* stack failed the test, the *newtask* stack was corrupted by an improper write. Since *newtask* was not running, the problem lies in a different task. In either event, SYS_abort will be called with an error message.

Note that although the stack is filled with TRG_STACKSTAMP, the only location tested is the last one in the stack.

TSK_checkstacks can also be called directly by an application, but the API has greater value if used at context–switch time. This ensures that at every task context switch, the integrity of incoming and outgoing stacks is checked. This is done via the DSP/BIOS configuration tool. In the Task Manager, set the property *Call switch function* to *True*, and set the property *Switch function* to *_TSK_checkstacks*.

If a stack overflows, the stack should be manually examined to try to determine the extent of the overflow, and the next course of action. An overflow may simply indicate that your estimate of stacksize was incorrect for this task, or it may indicate a bug in the task. Experiment with the stacksize; make it larger to see if the problem recurs, and try to determine the actual required stacksize by growing and shrinking the stack. Once the required size is actually determined, then you can examine why your estimate was wrong. An additional approach is to over–allocate your stacks from the start, fully exercise the system, and base your final stack allocations on the amount of stack that was actually used.

The DSP/BIOS configuration tool automatically estimates the minimum required size of the ISR stack for a system, based on the DSP/BIOS objects statically defined in the configuration. This value can be changed by editing the *Stack Size* field of the *Memory Section Manager Properties* window; the memory segment to store the stack is also here and can be changed. The *Task Manager Properties* window specifies the default task stack size as well as memory segment; both can be changed. However, each task can have its own task size and segment specified in the *Properties* window for the individual task.

While growing the stacksize may be the final answer, it also may not be. Memory constraints are tight in a DSP–based system, and before you arbitrarily increase your stacksize, you need to understand why the stack overflowed in the first place. There are also some situations where TSK_checkstacks will miss problems or report errors erroneously; these are discussed in the next section. Manual examination and experimentation are always necessary.

Stack corruption is a difficult problem to pin down. An important thing to realize is that the current task may not have done the damage. Manual examination is required first, to see if there is a recognizable pattern to the corruption. Many times this will indicate the source of the problem quickly. The next step is to examine any pointer values used for dereferencing. Use the LOG facility to log the pointer values before they are used, to ensure they are correctly initialized. There are tools within CCS, such as the *Emulator Analysis Tool*, which provides hardware breakpoints on memory accesses. This useful tool is discussed more in the *Memory* section of this document.

A more common cause of memory corruption is writing beyond the bounds of an array. Whenever you write to an array, the index should be checked to ensure you are writing to the declared storage region for that array. Such bounds checking may be too much overhead for the final application, but it can be done in the debug phase with ASSERTs, and compiled out when debugging is complete. ASSERTs are discussed later in this document.

2.2.1 Limitations of TSK_checkstacks

As mentioned above, TSK_checkstacks has several potential shortcomings.

Only the individual task stacks are checked by TSK_checkstacks(); the ISR stack is not.

An overflow will be reported if the last word of the stack is overwritten. Technically, this may not be an overflow, if that were the last location written to on the stack. In that case, the stack took up exactly the size declared, although it is not recommended to size the stack so closely. Adding extra storage to your stack size will clear up this problem, and give you the spare room you need. An additional 10% stack storage is suggested to handle unanticipated conditions that may occur.

An overflow may occur and go undetected, due to holes in the stack left for alignment purposes. Stack space may be allocated, but not used. In either of these cases, the TRG_STACKSTAMP value will remain in that location. If the TRG_STACKSTAMP happens to be there, the stack may continue to overflow undetected.

2.3 TSK_stat()

TSK_stat() is a DSP/BIOS API that retrieves status data about a task, including the task's stack. The actual syntax of the API call is:

Void TSK_stat (TSK_Handle task, TSK_Stat *buf)

The buffer is used to hold a structure of data containing the current status of the task. Included in this data are the following stack–related items: *current stack pointer*, *maximum number of MAU's used on the stack*, *stack base address*, *memory segment for the stack*, *declared stack size*. If the number used is equal to the size of the stack, then the stack has overflowed. A task can check its own status, or the status of other tasks, and can take action if the stack overflows, or comes close to overflowing. The actual structure returned in the buffer is as follows:

```
struct TSK_Stat {
       TSK_Attrs    attrs; /* task attributes */
       TSK_Mode     mode;  /* task execution mode enum */
       Ptr          sp;    /* task stack pointer */
       Uns          used;  /* task stack used */
};
struct TSK_Attrs {
       Int          priority;    /* execution priority */
       Ptr          stack;       /* pre–allocated stack location */
       Uns          stacksize;   /* stack size in MAU's */
       Int          stackseg;    /* memory segment for stack allocation */
       Ptr          environ;     /* global environment data structure */
       String       name;        /* printable name */
       Bool         exitflag;    /* program termination requires this task terminate
*/
       TSK_DBG_Mode debug;       /* debug enum */
};
```

For more detailed information on these structures or this API, see the *DSP/BIOS User's Guide*.

2.4 HWI Monitor Property

Each HWI in the system has a property associated with it called *monitor*, that can be used to monitor data values, system registers, and the stack pointer. An STS object is associated with the HWI that is being monitored. This allows you to open up the Statistics View within CCS and watch the maximum, minimum, etc., values of the monitored data as the application runs. The feature is accessed via the *Property Window* of the *HWI Window* in the configuration tool. Details of how to use this feature are provided in Chapter 3 of the *DSP/BIOS User's Guide*. **This tool is only useful in systems that do NOT use tasks.**

Be aware that when the monitor property is enabled for a particular HWI, a code preamble is inserted into the HWI ISR, which will make this monitoring possible. The overhead for monitoring is 20–30 instructions per interrupt, per HWI monitored. It is not recommended to leave this instrumentation turned on after debugging, since HWI processing is the most time critical part of the system.

2.5 Kernel/Object View

The Kernel/Object View is a new tool in version 1.2 of CCS. The Kernel/Object View allows the examination of the status of all DSP/BIOS objects in the system, whether created statically or dynamically. Within the Kernel/Object View is the ability to view status information about the DSP/BIOS Kernel (KNL), tasks (TSK), mailboxes (MBX), semaphores (SEM), memory sections (MEM), and software interrupts (SWI).

For the KNL object, the following information is available:

- `ISR stack start address, end address, size, and maximum used`
- `Tasks currently blocked`
- `Kernel mode`
- `Processor ID`
- `Current clock`

For the TSK object, the following information is available:

- `Task name & handle`
- `Task stack start address`
- `Task stack end address`
- `Task stack maximum used`
- `Task state`
- `Task priority`

For the MBX object, the following information is available:

- `Mailbox name & handle`
- `Current & maximum number of messages per mailbox`
- `Message size`
- `Tasks pending on read`
- `Tasks pending on post`
- `Memory segment for the mailbox`

For the SEM object, the following information is available:

- `Semaphore name & handle`
- `Count`
- `Tasks pending`

For the MEM object, the following information is available:

- `Memory section (heap) name`
- `Maximum contiguous space in the heap`
- `Free space`
- `Start address`
- `End address`
- `Amount used`
- `Memory segment`

For the SWI object, the following information is available:

- `SWI name & handle`
- `State`
- `Priority`
- `Associated mailbox`
- `Associated function and arguments`

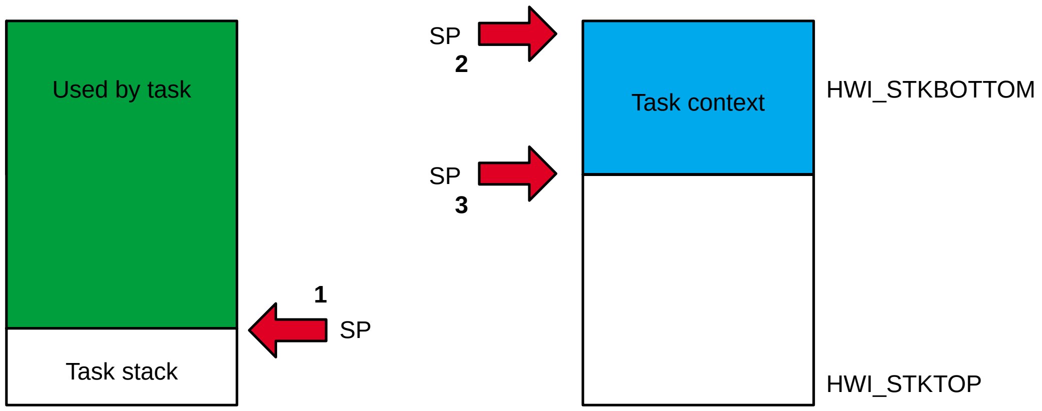
Of particular value is the stack information. For both system and task stacks, if an overflow is detected, the *max used* or *peak used* boxes will turn red and the text will be yellow. However, note that a stack containing holes due to alignment may escape overflow detection, similarly to TSK_checkstacks. See the previous sections on *TSK_checkstacks* for more information.

The Kernel/Object View is accessed from the *DSP/BIOS* menu on the *Tools* pulldown in CCS. For more information on this tool, see the *DSP/BIOS User's Guide, literature number* .

2.5.1 Stack Usage During Context Switch

If a task is preempted by interrupt (HWI or SWI), its context is saved on the ISR stack, and processing of the interrupt proceeds on the ISR stack. *Task context* is defined here as *the contents of the CPU registers at the time a task was interrupted*.

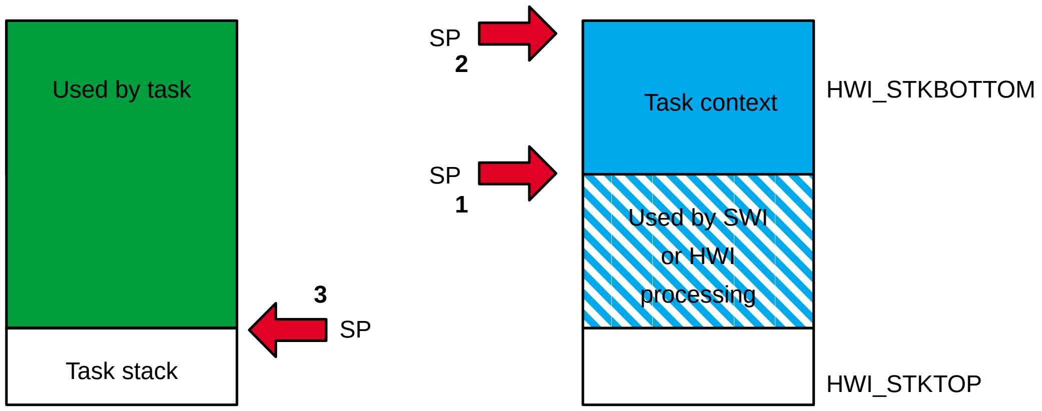
Figure 1. Task Preempted by Interrupt



In Figure 1, the task is running on its stack until it is interrupted at point **1**. The stack pointer then changes to the bottom of the ISR stack at point **2**. The context of the interrupted task is placed on the ISR stack, advancing the stack pointer to point **3**, which is where the ISR processing actually begins. The stack builds downward in memory (toward low addresses).

When the interrupt processing is complete, if the previously running task should be resumed, then the context is restored, and processing of the task continues on that task's stack.

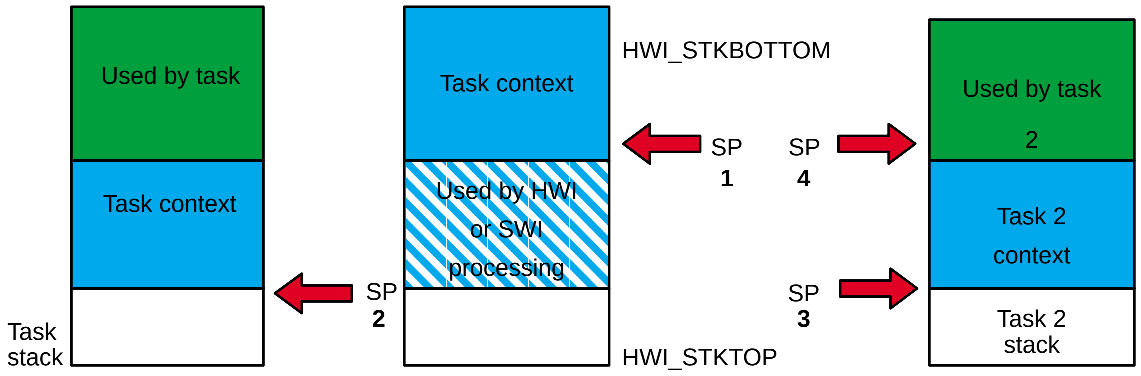
Figure 2. Resuming Task Processing After Interrupt



In Figure 2, all interrupt processing has completed at point **1**. The interrupted task context is then popped from the ISR stack, moving the stack pointer to point **2**. The task context is then restored, returning the stack pointer to its value when interrupted, at point **3**.

If, however, the preempted task is blocked and a different task is ready to run, the saved context is transferred from the ISR stack to the preempted task's stack, the context of the new task is retrieved from its task stack, and the new task starts executing on its task stack.

Figure 3. Task Blocked After Interrupt Processing



In Figure 3, all interrupt processing has completed at point **1**. The kernel determines that task 2 should run, so the interrupted task context is transferred from ISR stack to the first task's stack at point **2**. The context of stack 2 is then retrieved from its task stack at point **3** and restored, returning the stack pointer to its value when task 2 was blocked, at point **4**.

If the system does not use any TSK objects, then there are no task stacks, and all interrupt processing takes place on the ISR stack.

2.6 C54x Considerations

On the C54x, the *minimum addressable unit* (MAU) is 16 bits. Stack alignment is 2 MAU, by C programming convention. This alignment must be maintained by user code when interfacing between C and assembly. The stack pointer register is the SP register.

As discussed earlier, the top of each stack is tagged with a special value, TRG_STACKSTAMP, which is used by TSK_checkstacks to detect overflow and corruption. On the C54x this value is 0xBEEF, which includes the ISR stack.

A known potential source of stack and memory corruption on the C54x is improper setting of CPL and DP before calling DSP/BIOS APIs and C routines from assembly. This topic is discussed in detail in the section on *DSP/BIOS Preconditions*.

2.7 C6x Considerations

On the C6x, the *minimum addressable unit* (MAU) is 8 bits. Stack alignment is 8 MAU, by C programming convention. This alignment must be maintained by user code when interfacing between C and assembly. The stack pointer register is B15.

As discussed earlier, the top of each stack is tagged with a special value, TRG_STACKSTAMP, which is used by TSK_checkstacks to detect overflow and corruption. On the C6x, TRG_STACKSTAMP is used for task stacks only. The ISR stack is tagged with a different value. These values are as follows:

```
TRG_STACKSTAMP:     0xBEBEBEBE
Top of ISR stack:   0x00C0FFEE
```

3 Hardware Interrupts

Interrupts are the backbone of a real-time system. They are the events that represent real-world occurrences (e.g., a button pressed, a signal lost, a signal acquired, new data has arrived) to the DSP. Therefore, interrupt processing is one of the most complex and critical issues in an embedded system. In a DSP/BIOS system, interrupts are handled by the hardware interrupt (HWI) manager. In short, any APIs that create or delete system objects, or can potentially block on a system resource (e.g., semaphore, mailbox, etc.), should **not** be called from an HWI. Appendix A in the *DSP/BIOS User's Guide* contains a complete list of DSP/BIOS APIs and their context sensitivities, including those which should and should not be called from an HWI context. Consult this table to ensure your ISR is performing only operations that are legal in that context.

If you need to use one of these APIs in your application, your alternatives are calling them from an SWI, or from a task. The *Context Sensitivity of the DSP/BIOS API* section presents the rest of the context sensitivities of the APIs.

If your ISR does not make any DSP/BIOS calls, there is no special requirement for it. This is because it will not take an action that results in a context switch.

3.1 ISRs That Use DSP/BIOS

If your ISR makes any DSP/BIOS calls, there are requirements that must be met. Two macros, HWI_enter and HWI_exit, must be called from the ISR, whether it is written in C or assembly. HWI_enter must be placed *before* any DSP/BIOS calls, and HWI_exit must be **the last instruction executed** of the ISR. Use *HWI_exit* instead of *return* to end your ISR. Together, these macros perform the following functions for your ISR:

- Save and restore CPU registers as specified (preserve context for C calls – particularly DSP/BIOS API calls)
- Disable and enable specific interrupts (control nesting of interrupts)
- If nested interrupts occur, call the DSP/BIOS scheduler only after the outer (first) ISR completes
- See the *C54x Considerations* section for an additional feature of HWI_exit.

The HWI Dispatcher can be used in place of HWI_enter and HWI_exit. The HWI Dispatcher is useful when you want to write your ISR in C. Section 3.7 contains more information on the HWI Dispatcher.

WARNING:
You should never use the compiler's 'interrupt' keyword with a DSP/BIOS program. The interrupt keyword does not generate code that is aware of the DSP/BIOS scheduler. The HWI_enter and HWI_exit macros and the HWI Dispatcher should be used instead. Using the interrupt keyword in a DSP/BIOS application will cause unpredictable program behavior.

When an HWI is triggered, interrupts are disabled globally by the DSP; then processing jumps to the ISR in the vector table. HWI_enter uses the interrupt mask specified in the call to disable specific interrupts only. This mask allows the user to enable or disable interrupt nesting, on an individual HWI basis. Finally, HWI_enter re-enables unmasked interrupts globally.

When HWI_exit is called, it globally disables interrupts. It then uses the interrupt mask specified to re-enable any interrupts disabled with the HWI_enter call. HWI_exit will re-enable only those interrupts you specify, and *ONLY* if those interrupts were previously disabled with HWI_enter. It then globally re-enables interrupts.

3.2 Interrupts and the Pipeline

Writing assembly code is a more detailed process than writing in C; writing assembly code for a pipelined DSP like the C6x or C54x presents its own particular set of challenges. This is not a thorough discussion on pipeline issues for these processors; it is a discussion of how interrupt interactions with the pipeline make life interesting for assembly language programmers, and how to deal with it. These examples are presented for both C6x and C54x users.

Example 1.

Consider the following example, for the C6x:

```
1      LDW    *A5++,A2
2      LDW    *A6++,A2
3      SUB    B4,B1,B2
4      NOT    B2,B4
5      NOP
6      ADD    A2,A3,A4
7      MV     A2,A3
```

Example 1A

This code sample may be the result of pipelining code for optimization. Although the load instructions in lines 1 and 2 load to the same register, A2, each load instruction takes four delay slots to take effect. For the first load instruction, the value is in A2 after line 5. For the second load instruction, the value is in A2 after line 6. Therefore, the add instruction in line 6 uses the first value loaded into A2. After line 6, A2 is then loaded with the second value, so line 7 moves the second value into A3.

This all works fine until an interrupt happens. If an interrupt occurs between line 2 and line 6, both load instructions will complete before line 6 is executed. When the processor is interrupted, any instructions that were previously fetched to the E1 phase of the pipeline will complete, so each load instruction will complete after the usual four delay slots. This occurs while the ISR is processing. After the ISR completes, A2 will have already been overwritten by the second load instruction, so line 6 will add this value to A3: error. Line 7 executes normally, since this line should have operated on the second loaded value. But the end result is that the value in A4 is wrong; it was calculated with the wrong value in A2. Note that if the interrupt had occurred between line 1 and line 2, processing would complete normally, since A2 would be loaded with the first value by the time the ISR completes. If no more interrupts came in, the code would complete correctly.

A similar example for the C54x follows:

```
1      STLM   A,AR2
2      STLM   B,AR2
3      NOP
4      LD     *AR2,A
5      LD     *AR2,B
```

Example 1B

Lines 1 and 2 require two delay slots to take effect, which means that if an interrupt occurs between line 2 and line 4, the same problem will occur as on the C6x.

This code is an example of *double assignment*, since the same register is used to hold two different values, one or both of which is pending in the pipeline. In double assignment, timing and instruction order are critical. Any interruption will throw off the timing and instruction order, and will therefore cause erroneous results. This can be extremely difficult to debug, since stepping through the code may yield the correct result. Both pieces of code, the main code and the ISR, may execute perfectly well independently from each other. The bug is in their interaction, in real time.

How to deal with this? There are two ways. First, this code could be rewritten to use *single assignment*. Whereas double assignment uses the same register to hold one value while another value is pending in the pipeline for that register, single assignment requires that no register is used that has a value pending in the pipeline. Basically, what you see is what you get with single assignment. Here is Example 1A rewritten for single assignment:

```
1      LDW    *A5++,A2
2      LDW    *A6++,A7
3      SUB    B4,B1,B2
4      NOT    B2,B4
5      NOP
6      ADD    A2,A3,A4
7      MV     A7,A3
```

Example 1C

This code performs exactly as does the first code sample. However, consider an interrupt coming in somewhere between line 2 and line 6, the previous “trouble area.” By the time the ISR completes, both load instructions have completed; A2 & A7 are completely set up for their respective operations in lines 6 and 7. The operations execute with no errors, and all is right with the world. The downside to this solution is the use of one additional register. While this may not break the back of your resource budget, in some scenarios, a similar tradeoff might be a difficult one to make.

Here is the single assignment solution for example 1B:

```
1      STLM   A,AR2
2      STLM   B,AR3
3      NOP
4      LD     *AR2,A
5      LD     *AR3,B
```

Example 1D

There is another solution: disable interrupts anytime you use double assignment. This is what the compiler and assembly optimizer do when dealing with software pipelined loops: disable interrupts before the critical code segment and enable them afterward.

Add this functionality to the code in Example 1A, and the result is the following, for C6x:

```
1      MVC    CSR,B3
2      CLR    B3,0,0,B3
3      MVC    B3,CSR
4      LDW    *A5++,A2
5      LDW    *A6++,A2
6      SUB    B4,B1,B2
7      NOT    B2,B4
8      NOP
9      ADD    A2,A3,A4
10     MV     A2,A3
11     MVC    CSR,B3
12     SET    B3,0,0,B3
13     MVC    B3,CSR
```

Example 1E

This example uses the GIE (global interrupt enable) bit in the CSR register to disable and enable interrupts. This bit controls all maskable interrupts, and is the best way to ensure that no interrupts will occur in this section of code. GIE is bit 0 in the CSR.

Lines 1–3 disable interrupts by clearing bit 0 in the CSR. Lines 11–13 enable interrupts by setting the same bit in the CSR. The rest of the code is exactly the same as in example 1A. The basic rule of thumb is: when writing single assignment code, interrupts may be enabled; when writing double assignment code, interrupts must be disabled for that section of code. This will accomplish the desired effect for the C6x.

The solution for C54x is presented in the next section, because it’s not quite as simple. In addition, the door isn’t quite closed on the C6x example, either.

Example 2.

Now that we know we can disable interrupts during critical code sections, let’s look at another example of how interrupts can cause trouble.

The C54x example will use the code from Example 1B. Since it uses double assignment, interrupts must be disabled before it, and enabled after. The resulting code follows:

```
1      SSBX   INTM
2      STLM   A,AR2
3      STLM   B,AR2
4      NOP
5      LD     *AR2,A
6      LD     *AR2,B
7      RSBX   INTM
```

Example 2A

This example uses the INTM (global interrupt mask) bit in the ST1 register to disable and enable interrupts. This bit controls all maskable interrupts, and is the best way to ensure that no interrupts will occur in this section of code. INTM is bit 11 in the ST1.

Line 1 disables interrupts by setting bit 11 in the ST1. Line 7 enables interrupts by clearing the same bit in the ST1. The rest of the code is exactly the same as in Example 1B.

3.3 HWI_disable, HWI_enable, & HWI_restore

As the previous examples have shown, disabling interrupts is a bit tricky on the C54x and C6x DSPs. In addition, the procedures involved are different for each family. Because of this, DSP/BIOS provides a central mechanism for dealing with interrupt enable/disable, that is guaranteed to work, and is common across all processor families.

HWI_disable globally disables maskable interrupts, in much the same way shown in the above examples. It protects against interruption, so that the operation will always succeed, and no special processor–specific steps need to be taken by the programmer. HWI_disable returns the value of the global interrupt enable flag (among other status flags) before interrupts are disabled. This is for use later in HWI_restore.

HWI_enable always enables global interrupts, no matter the state of the global interrupt bit or the PGIE bit, if it exists. HWI_enable should not be used if enabling/disabling of interrupts may be nested. The previous state of the interrupt enable bit will not be maintained.

HWI_restore restores the global interrupt bit to its previous state before HWI_disable was called. HWI_restore also requires that interrupts be disabled before it is called, which is not true of HWI_enable. HWI_enable does not have an argument, whereas HWI_restore specifies the previous CSR state (saved in the call to HWI_disable). Use HWI_restore if the calls may be nested.

Example 3.

The final recommended solutions to both C6x and C54x cases are as follows:

```
HWI_disable  B0
LDW          *A5++,A2
LDW          *A6++,A2
SUB          B4,B1,B2
NOT          B2,B4
NOP
ADD          A2,A3,A4
MV           A2,A3
HWI_restore  B0
```

Example 3A (C6x)

```
HWI_disable  *AR1~~AR1~~

STLM         A,AR2
STLM         B,AR2
NOP
LD           *AR2,A
LD           *AR2,B
HWI_restore  *AR1~~AR1~~
```

Example 3B (C54x)

Note that in the examples, HWI_restore could be replaced by HWI_enable if nesting is not required.

3.4 C54x Considerations

On the C54x, there is an additional benefit to using HWI_exit to end your ISRs. If you have built your application with the far memory model, HWI_exit automatically executes a far return from ISR (the far memory model is discussed in detail in the section on *Memory*).

3.4.1 HWI_enter and HWI_exit Preconditions

The preconditions for these macros are presented here because of their importance. Most of the postconditions for HWI_enter (the conditions set by the macro) are the same as the preconditions for HWI_exit. The preconditions for HWI_exit are:

```
cpl = ovm = c16 = frct = cmpt = 0 (cpl is compiler mode bit;
                                   ovm is overflow mode bit;
                                   c16 is ALU precision mode bit;
                                   frct is fractional mode bit;
                                   cmpt is compatibility mode bit;
                                   all bits located in ST1 register)
dp = GBL_A_SYSPAGE                (dp is data page pointer, in ST0)
```

The meaning of these preconditions is discussed in more detail in the section on *DSP/BIOS API Preconditions*.

The only precondition for calling HWI_enter is that interrupts must be globally disabled. This is automatically accomplished by the DSP in response to the interrupt, before jumping to the ISR. Do not call HWI_enter in a context where interrupts have not been globally disabled.

3.5 HWI_disable and HWI_restore

On the C54x, the assembly API for HWI_disable differs slightly from the C API. The C API always returns the value of the ST1 register before the interrupts were disabled. The ST1 register contains the INTM bit, which is the global interrupt enable, in bit 11. The assembly API takes one optional parameter which determines whether or not the ST1 value will be returned, and where it will be stored. If the API is called without the parameter, interrupts are disabled and the ST1 value is NOT saved. If the API is called with the parameter, then the ST1 value is returned in the location specified by the parameter. The parameter may be any of the following values:

A (old value returned in accumulator a)

B (old value returned in accumulator b)

*arx (old value stored in memory location pointed to by arx register)

The HWI_restore takes the same parameter values. Using HWI_restore with no parameter assumes the value is in accumulator A. The only bit that is important in the restore operation is bit 11, which is the INTM bit of ST1. In the restore operation, the INTM bit will be set according to the value of bit 11 in the old ST1 value. No other bits in ST1 are affected in this operation.

3.6 C6x Considerations

On this processor family, HWI_restore is more efficient in code size than HWI_enable.

3.6.1 HWI_enter and HWI_exit Preconditions

The preconditions for these macros are presented here because of their importance. Most of the postconditions for HWI_enter (the conditions set by the macro) are the same as the preconditions for HWI_exit. The preconditions for HWI_exit are:

```
b14 = pointer to the start of .bss      (b14 is the data page pointer)
amr = 0                                 (amr is addressing mode register)
```

The meaning of these preconditions is discussed in more detail in the section on *DSP/BIOS API Preconditions*.

The only precondition for calling HWI_enter is that interrupts must be globally disabled. This is automatically accomplished by the DSP in response to the interrupt, before jumping to the ISR. Do not call HWI_enter in a context where interrupts have not been globally disabled.

3.7 HWI Dispatcher

The HWI dispatcher provides the same functionality as HWI_enter and HWI_exit. The dispatcher is an option, selectable from the *HWI Module Properties* menu. Using it removes the requirement to specify HWI_enter and HWI_exit wrappers to your ISR. The parameters for HWI_enter and HWI_exit can be found as additional entries on the *Properties* menu. This is the preferred method of writing HWI routines (ISRs), since it centralizes the function, reduces codespace, makes code more readable, and provides the same functionality as HWI_enter and HWI_exit.

The dispatcher is turned off by default, to maintain backward compatibility. When turned off, the HWI ISRs, with calls to HWI_enter and HWI_exit, will function normally. However, when the dispatcher is turned on, any HWI_enter and HWI_exit calls must be removed from your code.

Use of HWI_enter and HWI_exit calls when the dispatcher is turned on will crash your application.

4 Memory Issues

4.1 Accessing Out–of–Range Memory Addresses

If you are familiar with embedded microprocessors, or just about any other CPU, probably the most important thing to know is that on these DSPs there is no bus/address exception.

The second important thing is that on these DSPs there is no illegal instruction exception.

Therefore, you can access memory that does not exist, you can fetch instructions from outside your code space, and you can execute any garbage you happen to fetch as an instruction. The DSP will just keep running, with unpredictable and most certainly negative results. This provides the DSP programmer with some unique challenges in debugging. How do you deal with an environment that does not tell you when something so fundamental goes wrong?

The suggested methods are presented in the hardware–specific sections. They work well for memory accesses that are outside the intended codespace. However, for accesses to nonexistent memory, a logic analyzer is an excellent tool. Code Composer Studio also offers the *Emulator Analysis Tool*, which performs some of the same functionality as a logic analyzer. This tool is discussed in the next section.

4.2 Emulator Analysis Tool

The Emulator Analysis Tool is a plugin to the CCS environment that allows you to set hardware breakpoints and count events. Hardware breakpoints are useful in environments where software breakpoints are problematic, such as ROM–based code. The analysis tool can be used to monitor program buses, data buses, or I/O buses, and break on different types of bus cycles. The tool is customized for the individual processor families, based on the capabilities and features of the particular DSP. Complex conditions for breakpoints can be set up with this tool. Most of the debugger functionality available with software breakpoints is also available with hardware breakpoints (step, run, etc.).

Some CPUs also offer the ability to set global breakpoints. These are breakpoints that are set in a multiprocessor system that, when hit by one DSP, will cause all DSPs in the system to halt. This feature is dependent on the setup and capabilities of the emulation system.

The Emulator Analysis Tool is accessible from the Tools pulldown in Code Composer Studio. Consult the *Help* facility in the tool to learn about the tool's capabilities on your target processor.

4.3 C54x Considerations

4.3.1 Fill Memory with Interrupt–Generating Instruction

One way to detect when the CPU has advanced the program counter beyond the range of valid code is to fill memory with an instruction that generates an interrupt to the DSP. Also, place a breakpoint at the vector for the particular interrupt you've chosen. Then download your application code and run. If the processor tries to execute out of range, it will generate the interrupt, go to the vector, and hit the breakpoint. The state of the machine, including stack, will be available for tracing and analysis.

CCS allows memory to be filled with a single word. Therefore, this method works on any DSP that requires a single word instruction to generate a software interrupt. The C54x falls into this group.

There are two different instructions that can accomplish this: INTR and TRAP.

The syntax for the INTR instruction is: INTR <vector #>. The opcode is 0xF7Cx, where x is the vector number of the interrupt. For example, the opcode for *INTR 5* is *0xF7C5*.

The syntax for the TRAP instruction is: TRAP <vector #>. The opcode os 0xF4Cx, where x is the vector number of the interrupt. For example, the opcode for *TRAP 2* is *0xF4C2*.

The difference between the two is that TRAP does not disable interrupts; INTR does. However, neither instruction is maskable; they will generate the interrupt whether interrupts are enabled or not.

4.3.2 Near vs Far Addressing

The original C54x devices were 16-bit address devices, and therefore had a maximum address space of 64K words. Newer devices in this family incorporate a wider addressing scheme and can accommodate larger applications. However, this increased codesize capability comes with added complexity.

Although the memory models are referred to as “near” and “far”, there are really three choices to the memory model your application chooses. These choices all come with advantages and disadvantages, and they are differentiated primarily by the size of the application you wish to run.

The near model, as stated above, gives you 64K of program memory. There are two flavors of far model; one with the *OVLY* bit clear and one with this bit set. The far model with OVLY = 1 yields 4.03M unique program space. The far model with OVLY = 0 yields 8M unique program space, the maximum supported by the C54x devices. Actually, that is the theoretical maximum, not the practical one. This will be discussed later. The OVLY bit is located in the processor’s PMST register. It controls how extended memory and onchip memory are mapped in the processor space.

Since the original C54x devices addressed up to 64K, any memory beyond that in the newer devices is termed *extended memory*. Extended memory is segmented into pages of 64K each. The maximum number of extended pages supported is 128. The PC of the C54x always indicates an address in the range 0–FFFF (64K), with the page number specified in an additional register, the *XPC register*. The basic difference between near addressing and far addressing is in the use of the XPC register. Far addressing modifies and uses it; near addressing uses it but does not modify it. The processors that support far addressing have special instructions that use the XPC in address calculation.

In addition to the information contained here, application notes SPRA599 and SPRA492 offer more information about this subject. These notes are included in the *References* section.

4.3.2.1 When OVLY = 1

This setting causes 32K of memory (0–0x7fff)to be *overlaid on every extended memory page*. If every page is 64K, and we are duplicating the first 32K on every page, that leaves 32K unique storage per page, or 4M. The overlaid page is an additional 32K, so the total available storage in program space is 4.03M.

This is a large loss of potential address space; virtually 50%. However, there are advantages to this. When an interrupt occurs, the processor does a near access to the vector table (e.g., the processor assumes the vector table is located on the current page). If the vector table were on a different page, the processor would jump to the vector offset on the current page and execute the wrong code. The solution to this problem is to make the vector table visible on each page. Then, when the processor jumps to the vector, it will always execute the actual vector code. The 32K of overlaid memory provides a mechanism to accomplish this.

The interrupt vector table is a 128-word table that normally resides at 0xFF80, the top of the first 64K memory page. **If you set OVLY=1, the vector table must be moved from this address to be visible on the extended pages**. The vector table **must be aligned on a 128–word boundary within the first 32K.** The first 128 words of on-chip memory (0x0–0x7f) contain memory–mapped registers, and do not constitute valid program space. **This results in an acceptable range for the vector table of 0x80–0x7F80.** If you do not make this change, the vector table will exist only in page 0. If an interrupt occurs during processing on any other page, the processor will jump to what it thinks is a vector but is not, with unpredictable results.

Any on-chip memory will be mapped into data as well as program space, in the overlay page. On–chip memory is the fastest memory available to your application; the most critical algorithms and/or data should be placed here.

There are other restrictions on placement of code elements when OVLY = 1. The C wrappers used to access DSP/BIOS must also reside in the overlay page. This is because the core of DSP/BIOS must be near–callable (within the same 64K page) to both the C wrappers and the function glue stubs it uses. Therefore, DSP/BIOS and the function glue stubs must reside either in the overlay page, or page 0.

ISRs may be located in any page, but this comes with a restriction. If they are located in any page other than the overlay page, ISRs must terminate with either HWI_exit or a far return instruction. HWI_exit automatically executes a far return if the application was assembled with the far memory model. If your ISR does not terminate in one of these two ways, the results are unpredictable. This is just another good reason to use HWI_enter and HWI_exit.

These restrictions affect your partitioning of the application in memory, depending on your needs for onchip memory and overlay space.

4.3.2.2 When OVLY = 0

It is important to note that at this time, the DSP/BIOS configuration tool does not permit the OVLY=0 far mode to be used. All far mode configurations must have OVLY=1.

In this case, extended memory is used, but there is no overlay page. Also, onchip memory is not mapped into program space. Theoretically, this gives you a usable address range of 8M. But all is not as it seems.

You do not have the automatic overlay of storage that you have when OVLY = 1. This does not relax the requirement for the overlay; it just means you will have to perform memory duplication yourself.

The interrupt vector table must be copied to every page where interrupts are enabled during processing. If a page does not have the vector table present, then interrupts must be disabled for the duration of processing on that page. If this is not done and an interrupt occurs, the results are unpredictable.

In addition, since the C wrappers and function glue stubs must be near–callable to DSP/BIOS, all three components must be duplicated on the same pages together. If, however, a particular page contains code that does not make DSP/BIOS calls, none of these three need be duplicated there. It is possible to partition your code so that this is the case, if you wish to maximize your usable memory.

But it is a certainty that some duplication will be necessary; that is why 8M is a theoretical number. Taking into account duplication, that number drops, although it is still larger than 4.03M.

The rule about using far returns for ISRs applies here as well; far support is included at assembly time, so HWI_exit is the best way to do this.

Another disadvantage is that the necessary duplication, as part of the user application, must all be done manually. This requires block copies and special consideration since you are copying to code space, not data space.

Probably the most significant disadvantage is the lack of onchip memory in program space. Although the onchip will still be available for data space, your application will run entirely from external memory. This is a speed consideration that must be taken into account by the designer.

4.3.2.3 Selecting the Desired Memory Mode

This section does not discuss setup for OVLY=0 far mode, since this configuration is not supported.

Open the DSP/BIOS configuration for your application. Right–click on *Global Settings*, and select *Properties* from the popup menu.

If you want the near model, set *Function Call Model* to *near*. You are finished.

If you want the far model, set the *Function Call Model* to *far* and ensure that bit 5 of the *PMST(6–0)* value is 1. Close the *Global Settings* window, and double–click on *Memory Section Manager.* Right–click on *VECT* to edit the base address of the properties of the interrupt vector table.

In *Memory Section Manager*, you may see two sections, *EPROG0* and *EPROG1*. These are two extended memory page objects, to use as starters. You may create as many of these as you need, up to 128. If OVLY = 1, the maximum size of these sections is 32K. To create them, right–click on *Memory Section Manager* and add these sections. Once they are there, right–click on each one and edit the *properties*. Enter the appropriate *base* and *length* for each page. Set the *space* value for each one to *code*.

Add the "–mf" flag and the "–v54*x*" (e.g., –v548, –v549, etc., depending on your CPU) flags to the options in the *Compiler* tab and the *Assembler* tab of the *Options* window of the *Project* pull-down.

Finally, the linker command file must be modified to load the desired code to the appropriate extended sections. The following two examples accomplish this in different ways.

In the first case, an entire source file is to be placed into an extended section. For each source file, add the following line to the linker command file:

```
<srcfilename>Xtext:{<srcfilename>.obj(.text)}.EPROGX PAGE X
```

where <srcfilename> is the name of the source file, and **X** is the page number to allocate to.

In the second case, a source file is split up so that different routines are placed into different sections. This example consists of several parts. First, use *#pragma CODE_SECTION* directives in the source code to allocate space for individual routines in specific sections. The format is:

```
#pragma CODE_SECTION(<routine>, “<section_name>”);
```

Then, in the linker command file, within the SECTIONS portion at the end of the file, specify the following:

```
<section_name>:{}>EPROGX PAGE X
```

where **X** is the page number to allocate to.

The above two examples first appeared in application note SPRA599 (see *References* section). They are included here for completeness. More detailed information may also be found in the manuals for C and assembly tools for C54x, also in the *References* section.

4.4 C6x Considerations

4.4.1 Branch to Same Location (C6x only)

One way to detect when the CPU has advanced the program counter beyond the range of valid code is to fill memory with an instruction that branches to itself. If normal functioning stops but the code continues to run, halt the DSP; you may be stuck on one of these instructions. This method is not foolproof, due to the pipelined nature of the C6x. On the C6x, the smallest loop size is six instructions. That means that instead of looping continuously on the one instruction, it will loop continuously on six of these instructions. This will work most of the time, except in the case where the six instructions cross a “boundary” with real data or code (e.g., three of the instructions are “branch to self”, and the next three are code or data). In that case, the results are unpredictable. But most of the time, the processor will get stuck in this “no–op” loop. The stack can be traced to determine how you ended up in this bad range of addresses, and the status bits will have additional debugging information that would have been lost otherwise.

The opcode to use is 0x00000012.

CCS allows memory to be filled with a single word. Therefore, this method works on any DSP that requires a single word instruction to generate this branch. The C6x falls into this group.

4.4.2 Near vs Far Addressing on the C6x

There are five memory models available on the C6x: one small memory model, and four variations of large memory models. The model you choose for your application determines how the .bss section is allocated in memory as well as how subroutines are called, both of which directly affect the time and space efficiency of your application. The .bss section is where global and static data are stored.

4.4.2.1 Small Memory Model

This memory model sets a maximum size for the .bss section of 32K bytes. This data is accessed in runtime as an offset from the *data page pointer* (DP). By convention, this pointer is kept in register B14, and is set only once, at initialization. This simple scheme allows the processor to use direct addressing to access data in the .bss section.

The small memory model also assumes that a subroutine being called is within plus or minus 1M from the calling routine. This allows the processor to use PC-relative branching.

This is also known as *near addressing*. It is the default memory model when none is specified to the compiler. It is also the model with which DSP/BIOS was compiled. It is the most efficient of the memory models, since it takes one instruction to access .bss data and one instruction to call a subroutine. This results in saved memory and saved execution time, with the disadvantage of size limitations. The following are examples of near addressing:

```
LDW    *DP(_data_item),A0  ;_data_item is accessed as an offset
                           ;from DP
B      _function1          ;branch to function1 is relative to
                           ;program counter
```

4.4.2.2 Large Memory Models

There are four varieties of large memory models, each differing somewhat in the degree of “largeness”.

4.4.2.2.1 Aggregate Data Large Memory Model

Aggregate data consists of structures and arrays. Aggregate data only is accessed via far addressing. This means aggregate data will not be placed in the .bss section, and the method of referencing it will differ from the method used for accessing .bss data. The 1M limitation on routine proximity for function calls still applies.

Far addressing of data means the object’s address must be loaded into a register before the object can be accessed. This requires two extra assembly instructions; more overhead in space and time. The following example shows this operation:

```
MVK    _x, A0       ;load lower half of address of x
MVKH   _x, A0       ;load upper half of address of x
LDW    *A0, B0      ;load the value of x
```

This model should be used if your global and static data take up more than 32K of storage. Note that far addressing is not imposed on function calls or .bss data accesses. It is invoked by specifying the flag “–ml0” in the compiler options.

4.4.2.2.2 Function Call Large Memory Model

This model causes function calls only to use far addressing. This means that, similar to far addressing of data, the address of the destination routine must be loaded into a register before the branch is executed. The overhead incurred is, again, two instructions:

```
MVK    _func, A0    ;load lower half of address of func
MVKH   _func, A0    ;load upper half of address of func
B      A0           ;branch to func
```

The 32K limitation on .bss data still applies. This model should be used if functions may reside more than 1M away from each other in memory. Note that far addressing is not imposed on any data accesses. It is invoked by specifying the flag “–ml1” in the compiler options.

4.4.2.2.3 Aggregate Data and Function Call Large Memory Model

This model is really a combination of the first two. Far addressing is not imposed on .bss data accesses. It is invoked by specifying the flag “–ml2” in the compiler options. The 32K limitation on .bss data still applies.

4.4.2.2.4 Function Call and All Data Large Memory Model

This model builds on the previous one by additionally accessing all data via far addressing. In this case, all .bss data will be accessed in the same way as aggregate data. This relieves the limitations on routine proximity as well as .bss size, with a tradeoff in efficiency. It is invoked by specifying the flag “–ml3” in the compiler options.

4.4.2.3 DSP/BIOS Considerations

Although DSP/BIOS is compiled using the small memory model, objects created in the configuration tool are not placed in the .bss section, regardless of the memory model with which you build your application. Instead, each type of object is placed in a separate section, named according to the object type (e.g., SIO$obj, TSK$obj, etc). You need to be aware of this, and choose how you will deal with it in your application. If you don’t deal with this issue, your application may work (by sheer luck of the linker) or it may not. Even worse, it may work for a while, until you add enough storage or move storage around in just the right (or wrong) way, and then it will break.

There are several options available to you; each has its own advantages and disadvantages.

The most straightforward solution is to compile with a large memory model. This automatically takes care of accessing far objects. It also allows you to place the objects anywhere you wish. However, it incurs overhead in storage and execution time to access the objects. This overhead can be minimized by your selection of large memory model (ml0, ml2, or ml3).

You can also declare a global object pointer for each object, and access the objects via these pointers only. This method will make the far access transparent, but it incurs the extra storage required for the pointers, in addition to the overhead of the far addressing. An example follows:

```
Extern  PIP_Obj inputObj;
PIP_Obj *myInput = &inputObj;
/* use myInput to access the inputObj object */
```

A third method of access is to declare each object as *far*. This would be done in the *extern* which references the object, as in the following examples:

```
extern far PIP_Obj inputObj;
extern far SIO_Obj inStream;
extern far LOG_Obj trace;
```

This is probably the easiest method to implement, since only the accesses to those objects are made with far addressing, and the rest of the application can be in the small memory model.

One final method, which will minimize overhead much more than any of the others, is to locate all the configuration objects adjacent to the .bss section in memory. The compiler will access all data up to 32K from the start of .bss using near addressing, whether the data is located in the .bss section or not. Therefore, if your .bss section is small enough, you may direct the linker to place the config objects immediately after .bss, thereby eliminating the overhead of far addressing. The drawback here is that you must be aware of the size of the .bss section and the size of your configuration objects throughout development, and if at any time your .bss and config objects together total more than 32K, you will have to change your methods. Placement of the config objects is accomplished by the following:

1. Define a memory segment to hold these objects and the .bss data, in the *MEM manager*
2. Assign each object to that segment by entering its name on the *property page* of the individual objects
3. On the *property page* of the *MEM manager*, enter the same segment name for the .bss section

The recommended approach is to declare the config objects as *far*, but the final choice always depends on your specific situation.

4.4.3 Reset Vector (C6x only)

A strange behavior sometimes seen on the C6x during debug is an application that resets, sometimes repeatedly. This will usually be visible because main runs repeatedly. The cause of a restart is that the program counter has jumped to location 0x00000000. This is the location of the reset vector, so a jump to it will result in a clean system reset.

Under normal circumstances, the program counter should not jump to zero in the middle of an application; arguably, this behavior should be reserved for a deliberate action taken as the result of a catastrophic system state. When it happens unintentionally, there may be several causes. Usually it happens because a NULL pointer was used as a pointer to a routine. This can happen when a pointer is retrieved from a table where the data is bad, or has been corrupted. It may also occur on the return from a subroutine, where the stack has been corrupted in just the right (or wrong) way.

Some application programs can run fine upon reset without a fresh download, so after the reset, the code runs down the same path it took before; the error happens again, and so the reset happens again. This explains the repeating pattern.

The C6x is unique in exhibiting this scenario because of its reset vector location. However, those familiar with the microprocessor world have no doubt seen the same phenomenon on the Motorola 68K family, which have their reset vector at location 0 also.

One method of safeguarding yourself against this is to set a breakpoint within the reset vector code. If this problem occurs, you'll catch it when it happens.

5 DSP/BIOS API Preconditions

One of the most important rules to follow when using DSP/BIOS is to **always meet the preconditions specified in the User's Guide for each DSP/BIOS API call**. These preconditions specify the states that specific bits and values need to have before the API call is made. The preconditions are required; if they are not met, the API call may not function as expected, and the results are unpredictable. The manual also specifies postconditions, which indicate the states of specific bits and values as they are left by the API call.

Meeting the preconditions does not necessarily mean that you will have to set all the indicated values to the indicated states; the preconditions may already be met by the current processor state. However, they may not, and this is the case where you will have to set them.

If a particular API call is not functioning as expected and the preconditions are suspect, they can be logged via one of the LOG functions. When doing this, don't forget that LOG functions are also APIs with their own particular preconditions.

The preconditions for APIs are very different from processor to processor. The following sections give detailed information about some of the more often–specified data items for preconditions on the specific processor families. These sections are meant to give some familiarity with the more important registers and bits in the processors. For more detail, see the related DSP architecture manuals indicated in the *References* section.

For the complete specification of preconditions per individual DSP/BIOS API, see the applicable DSP/BIOS User's Guide, also in the *References* section.

5.1 C54x Considerations

The preconditions of DSP/BIOS on the C54x reflect the complex programming environment of this processor family.

5.1.1 INTM

The INTM bit is found in register ST1, *Status Register 1*. As stated in the section on interactions between interrupts and the pipeline, the INTM bit is used to globally enable and disable interrupts. In those code examples, INTM was set using the *SSBX* instruction and cleared using the *RSBX* instruction. You should always use these instructions to manipulate the INTM bit, because it cannot be set or cleared by write operations. When INTM = 1, interrupts are disabled. When INTM = 0, interrupts are enabled.

INTM is set upon CPU reset and when a maskable interrupt occurs. It is cleared when RETE or RETF (return from interrupt) is executed. This bit does not affect non–maskable interrupts.

The required state of the INTM bit for DSP/BIOS is based on the individual API.

The information in this section is provided for completeness only; it is not meant to serve as an alternative to the use of *HWI_disable*, *HWI_enable*, or *HWI_restore* to affect changes on the INTM bit. The use of these APIs is always the preferred method.

5.1.2 CPL and DP

The CPL bit is found in register ST1, *Status Register 1*. It is the compiler mode flag, and indicates which pointer is used in *relative direct addressing*. When CPL = 0, the data page pointer (DP) is used. When CPL = 1, the stack pointer (SP) is used.

In *relative direct addressing,* also known as *direct addressing*, the instruction itself contains the lowest 7 bits of the data address. This 7 bits is treated as an offset, which is combined with a base address to form the final 16–bit data address. The base address may be either the stack pointer, in which case the two are added, or the data page pointer, in which case the two are concatenated. The advantage of direct addressing is that it encodes the instruction and the data address into a single word, making it a memory–efficient addressing mode.

The DP is a 9–bit field that is found in register ST0, *Status Register 0*. It is the data page pointer, which specifies the nine most significant bits of a data address generated in *relative direct addressing* mode when CPL = 0. The value in this field specifies which 128–word data page will be used. There are 512 possible pages.

DSP/BIOS accesses its internal variables via the data page pointer; therefore, DSP/BIOS requires CPL = 0. These variables are all placed on a single memory page at a location referenced by GBL_A_SYSPAGE. Before calling any DSP/BIOS API that accesses this page, the DP must be set to point to GBL_A_SYSPAGE in addition to clearing the CPL bit.

C programs on the C54x use the stack for storage of variables, so they require CPL = 1. Since they do not use the data page pointer, the value of DP is irrelevant.

If you are writing all code in C, you don’t need to worry about these issues; the DSP/BIOS C wrapper functions take care of them for you. However, if you are writing in assembly language, you need to know the difference between calling C functions and calling DSP/BIOS APIs, and set these values accordingly. The following example is taken from the C wrapper for the SWI_post call:

```
_SWI_post:
       RSBX   CPL                 ;clear CPL for upcoming call to SWI_post
       PSHM   ST0                 ;save DP onto stack (DP kept in st0)
       STLM   A,AR2               ;ar2 contains the address of the SWI object
       LD     *(GBL_A_SYSPAGE),DP ;use DP for direct addressing
       NOP                        ;for latency of STLM
       SWI_post                   ;actual call to BIOS macro
       POPM   ST0                 ;restore the DP from stack
       SSBX   CPL                 ;set CPL for return to C environment
       GBL_return                 ;return to caller
```

The effects of improperly handling these values is catastrophic. The stack, the data page, or memory mapped registers may be corrupted by software that thinks it is writing to one place but really is writing to another, and once this happens, the results are unpredictable.

The CPL bit should only be set and cleared using the SSBX and RSBX instructions. DP may be set using the LD instruction with a constant value, or from memory.

5.1.3 OVM

The OVM bit is found in register ST1, *Status Register 1*. It is the overflow mode bit, and specifies how an overflow will be handled after it occurs (e.g., what value will end up in the destination accumulator). When OVM = 0, the truncated result of the calculation that caused the overflow is placed in the destination accumulator normally. When OVM = 1, overflow saturation logic is enabled to actually prevent a result from overflowing.

The saturation logic works by placing either the most positive 32–bit value or the most negative 32–bit value in the destination accumulator when an overflow occurs. It is useful in algorithms that are computation intensive, such as filters.

DSP/BIOS usually requires OVM = 0 for API calls. The OVM bit should only be set and cleared using the SSBX and RSBX instructions.

5.1.4 C16

The C16 bit is found in register ST1, *Status Register 1*. It determines the arithmetic mode of the arithmetic/logic unit (ALU) of the DSP. When C16 = 0, the ALU operates in double–precision mode. When C16 = 1, it operates in dual 16–bit mode. Dual 16–bit mode is a special mode that performs two 16–bit mathematical operations in one cycle.

DSP/BIOS usually requires C16 = 0 for API calls. The C16 bit should only be set and cleared using the SSBX and RSBX instructions.

5.1.5 FRCT

The FRCT bit is found in register ST1, *Status Register 1*. It is the fractional mode bit. When set, fractional mode is enabled, which causes the multiplier output to be left–shifted by one bit to compensate for an extra sign bit generated by multiplying two 16-bit two's–complement numbers.

DSP/BIOS usually requires FRCT = 0 for API calls. The FRCT bit should only be set and cleared using the SSBX and RSBX instructions.

5.1.6 CMPT

The CMPT bit is found in register ST1, *Status Register 1*. It is the compatibility mode bit, and it affects indirect addressing mode operation.

DSP/BIOS requires CMPT = 0 for API calls. The CMPT bit should only be set and cleared using the SSBX and RSBX instructions.

5.2 C6x Considerations

DSP/BIOS requires very few preconditions on the C6x platform.

5.2.1 AMR

The *Address Mode Register* is the most specified register in the DSP/BIOS API preconditions. The C6x supports two types of addressing modes: linear and circular. Linear addressing is the standard mode, and can be used by all registers. Circular addressing is used for managing circular buffers, and can only be used by eight specific registers. The AMR indicates which mode is to be used for each of these eight registers.

In every case where AMR is specified as a precondition, it should be set to 0, which indicates linear mode on all registers.

6 Use of Assertions

Assertions are a useful tool to catch error conditions soon after they arise, while the code is running in real time. They are particularly useful in checking input parameters at the start of a function, or output parameters at the end. Many times, bounds checking is not done in real-time DSP applications because of timing overhead and resource considerations. ASSERTs allow the programmer to perform bounds checking in debug mode, and remove it for the final product. Keep in mind that since ASSERTs add code to the application, they also affect timing, so after they are removed, the application must be re–verified. An ASSERT macro can be defined to do a variety of things; write to a log, or even halt the CPU. A generic format for such a macro is as follows:

```
#ifdef DEBUG
#define ASSERT(a) if(!a){<action to take>}
#else
#define ASSERT(a)
#endif
```

It is a simple format that can be very useful to capture error conditions as they occur. “a” is a parameter to the macro. In your code, substitute it with a condition you want to trap on. If the condition tests FALSE, the action(s) will be taken. The #ifdef is included so that all ASSERTs placed in the code can be left in place and simply complied out. From the *Project* pulldown of CCS, define the symbol DEBUG in the *Compiler* tab of the *Options* window. The *Symbols* category allows you to define such symbols on the compiler command line. When you no longer need the symbol, simply remove it from this entry, rebuild, and the macro ASSERT results in no executable code. It remains in the source for debugging use at a later date, if necessary.

The macro above should be placed in a high level header file so that it can be accessed by all code modules.

The specific action to take can be a variety of things. You can execute a LOG_printf, LOG_error, LOG_message, increment a counter, or some other action that has meaning to your particular situation. Multiple actions can be combined in an ASSERT.

6.1 C54x Considerations

On the C54x, a very useful action for an ASSERT is the insertion of a breakpoint opcode. This will cause the CPU processing to halt when the ASSERT tests FALSE. The definition is as follows:

```
#define ASSERT(a) if(!a){asm(“ .word 0xf4f0”);}
```

If the condition tests FALSE, this code inserts a breakpoint opcode, 0xf4f0, into the execution stream. If the condition tests TRUE, execution continues as usual. Halting the CPU is extremely useful in that the bad value is detected early and the state of the machine is preserved, permitting the cause of the problem to be traced. This use of breakpoints may be combined with a call to a logging function to facilitate tracing the problem.

Note that although the breakpoint instruction is a single word, do not attempt to fill memory with it, as was suggested previously with the software interrupt instructions. When used in that way, CCS does not acknowledge the breakpoint.

6.2 C6x Considerations

Code Composer does not acknowledge breakpoints placed in an ASSERT (as described above) on the C6x.

7 Initialization of Global Storage

In standard C, global variables that are not initialized explicitly by the application are initialized to 0 by the compiler. The TI DSP compilers do not perform such a function. You must explicitly initialize all globals which need to have a value before runtime, including those you wish to have a value of 0. You can do this simply by indicating the initialization values in the declarations of all globals in your C code. For each value, the compiler creates an initialization record. These records are all gathered at link time into the .cinit section, resulting in an initialization table. This table is used either at load time or at runtime to initialize the system globals in a process known as *autoinitialization*.

Runtime autoinitialization is also known as *ROM autoinitialization* in the CCS environment. The .cinit records are copied into target memory for use by the bootloader at initialization time. The bootloader runs through the table and performs the indicated initializations. This is the type of autoinit your system will need if it will ultimately run from ROM, because the initialization data must be contained within the system. Resets take a little longer in this type of system because of the initialization process.

Load time autoinitialization is also known as *RAM autoinitialization* in the CCS environment. It causes the .cinit records not to be copied into the memory of the target, but instead, the loader on the host performs the initialization on the fly. With this type of autoinit, resetting the target will not cause re–initialization of the global data; only a reload of the program will cause that. Resets are quicker because this initialization is not done.

The selection of autoinit type is made on the Linker tab of the Options window of the Project pulldown.

The generation of .cinit records is a function of the C compiler, not the assembler. This means that if you declare globals in C modules, the compiler will generate the .cinit records for you; if you declare them in an assembly module, you must do it yourself. The formats of .cinit records are described in the processor–specific sections.

Creating these records can introduce errors in the initialization of your application if the above formats are not followed very carefully. If the count does not match the number of data values exactly, your program may never reach main(). If your application is not getting to main(), you may need to examine your .cinit records, and possibly step through the initialization process to debug the problem.

There are several ways to determine the location of the .cinit section. One way is to open a DOS window, cd to your project directory, and enter the command *sectti*. This will list every section in your application, along with the start addresses and sizes. If you'd like more detail about your application, create a map file by going to the *Linker* tab of the *Options* window of the *Project* pulldown, and entering a map filename. Rebuild your application, and then edit the map file. Once the location of the .cinit section is known, use a *memory* window in CCS to examine the records.

7.1 C54x Considerations

On the C54x, there is one format for a .cinit record. The first word of the record is the size of the init data in MAUs. If this number is positive, the record format is as follows:

```
Size of init data (in MAU's)
Address where init data is to be copied
Initialization data
```

More detailed format information and examples of .cinit records can be found in *the Optimizing C Compiler User's Guide* for the C54x family.

If you need to step through the code which does the .cinit initialization on the C54x, the code is in the routine *_c_int00.* The source code for this routine is in the boot.s54 file in the /ti/c5400/bios/src/misc subdirectory.

7.2 C6x Considerations

On the C6x, there are two possible formats a .cinit record may take; one is for initializing data values, and the other is for initializing pointers. The first word of the record is the size of the init data in MAUs. If this number is positive, the record format is as follows:

```
Size of init data (in MAU's)
Address where init data is to be copied
Initialization data
```

For the C6x, if the first word of the record is negative, the format is as follows:

```
Flag indicating DP patch
Addresses needing to be patched
```

This second format is for variables needing to point to other variables in the .bss section. Since .bss starts at the value of DP, and on the C6x the DP is set only once, all the addresses in the list are .bss addresses whose values need to have DP added to them.

.cinit records on the C6x have the additional requirement that they must be aligned on 8–byte boundaries.

More detailed format information and examples of .cinit records can be found in *the Optimizing C Compiler User's Guide* for each CPU family.

If you need to step through the code which does the .cinit initialization, step through the routine *_auto_init* on the C6x.

8 Context Sensitivity of DSP/BIOS APIs

Earlier in this document, the *Interrupts* section mentioned context sensitivity of DSP/BIOS APIs. In addition to HWI, SWI and tasks also have sensitivities to the APIs. Appendix A in the *DSP/BIOS User's Guide* details the APIs that should and should not be used in all of these contexts. In addition, Appendix A indicates which APIs may cause a context switch, and which do not. This is an important factor in deciding when to use a particular API.

8.1 Special Considerations for *main()*

Traditionally, the routine *main()* is the starting point of a user application. It is executed after all operating system initialization has taken place, and the operating system is fully functional. However, in a DSP/BIOS application, main() is really an extension of the DSP/BIOS initialization which can be customized by the user. When main() runs, DSP/BIOS has been initialized, but it is not yet running. Therefore, the scheduler is not available until after main() exits and DSP/BIOS is actually started. Interrupts are disabled until DSP/BIOS is started.

For this reason, APIs which cause a context switch are discouraged in main(). If the call were to block, execution will stop, waiting for an event that will never happen. Consult Appendix A of the *DSP/BIOS User's Guide* before placing any API calls in main().

Main() should not be thought of as part of your application; it is really part of DSP/BIOS. It can be used to perform some system initialization functions, such as hardware and data structure initialization. But be careful with hardware init, because interrupts are disabled when main() runs. Individual tasks will not run until after main() exits and DSP/BIOS is started. It is therefore safer to allow tasks to do their own initializations. In a system which uses SWIs instead of tasks, it is more appropriate to do such system initializations in main().

The following is the order of execution at system startup:

1.	BIOS_init	initialize DSP/BIOS modules
2.	Main()	perform application–specific initializations, if needed
3.	BIOS_start	start scheduler, enable DSP/BIOS modules, enable interrupts
4.	Application	

Interrupts and context switches can occur once BIOS_start exits. **Under no circumstances should interrupts be enabled in main().**

8.1.1 C54x Considerations

There are no special considerations for C54x.

8.2 C6x Considerations

There are no special considerations for C6x.

9 References

1. *TMS320C54x DSP Mnemonic Instruction Set User's Guide* (SPRU172)
2. *TMS320C54x DSP CPU and Peripherals User's Guide* (SPRU131F)
3. *TMS320C54x DSP/BIOS User's Guide* (SPRU326B)
4. *TMS320C54x Optimizing C Compiler User's Guide* (SPRU103C)
5. *TMS320C54x Assembly Language Tools User's Guide* (SPRU102D)
6. *TMS320C62x/C67x CPU and Instruction Set* (SPRU189C)
7. *TMS320C6000 DSP/BIOS User's Guide* (SPRU303A)
8. *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187E)
9. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186E)
10. *TMS320C6000 Peripherals Reference Guide* (SPRU190C)
11. *Issues About Using the CPL, XF, INTM, OVM, and SXM bits of the '54x Devices, by Changlin Chen*
12. *DSP/BIOS and TMS320C54x Extended Addressing* (SPRA599)
13. *Interrupt Handling Using Extended Addressing of the TMS320C54x Family* (SPRA492*)*