Application Report
SPRA820 – May 2003

# Online Stack Overflow Detection on the TMS320C28x DSP

*David M. Alter* *DSP Applications – Semiconductor Group*

**ABSTRACT**

A stack overflow in an embedded DSP application generally produces a catastrophic software crash due to data corruption, lost return addresses, or both. Traditional off-line approaches to sizing a stack during development, such as filling with a know value, or estimating based on code content, are not 100% reliable. Therefore, programmers often feel compelled to reserve larger stack sizes than are actually needed. This wastes valuable memory resources. Facilities exist on the TMS320C28x™ DSP that, when properly configured, allow for runtime detection of a stack overflow before it occurs. Detection of an impending stack overflow triggers a maskable interrupt, and software can then take whatever corrective action is desired before a software crash occurs. This application report presents the methodology for online stack overflow detection on the TMS320C28x DSP. C-source code is provided that contains functions for implementing the overflow detection on both DSP/BIOS™ and non-DSP/BIOS applications. The sample code described in this application report can be downloaded from http://www.ti.com/lit/zip/SPRA820.

## Contents



## Figures



## Tables

# 1 Introduction

A stack overflow in an embedded DSP application generally produces a catastrophic software crash due to data corruption, lost return addresses, or both. The traditional approach to avoiding stack overflow is to perform offline testing during software development. Typically, a stack will be comfortably oversized, and the entire stack memory filled with some known data value using a code debugger. The application software will then be run over some period of time (e.g. hours, days, or sometimes even weeks). At the end of the run period, the stack memory is examined using the code debugger. The unused portion of the stack will still contain the pre-filled data value, and thus the amount of stack used by the application is readily apparent. A factor of safety can then be applied, and a final stack size determined. While this offline method of stack sizing is invaluable as a first pass approach, it does not eliminate the possibility of a stack overflow occurring at runtime. Programmers may therefore use a larger stack than they might actually need, which can waste valuable on-chip RAM resources.

Facilities exist on the TMS320C28x DSP (hereafter referred to as the C28x™) that, when properly configured, allow runtime detection of a stack overflow before it occurs. When the (incrementing) stack pointer exceeds a specified address before the end of the stack, a maskable interrupt is triggered (the RTOSINT interrupt). Software can then take corrective action to prevent an application crash. The choice of action taken is solely up to the user, and should be suitable for the particular application at hand. For example, one might choose to simply perform a controlled shutdown, or perhaps restart the code by performing a DSP reset. Alternately, certain applications might allow one to throttle back the number of duties being performed until the stack usage reduces to a safer level. One should remember that a stack overflow is not caused by a bug in the code, but rather is due to a stack that has been sized too small for the worst-case demands of the application.

The on-chip resource that enables stack overflow detection on the C28x DSP is known as the *emulation analysis block*. Although primarily intended for use by the Code Composer Studio™ debugger, the analysis block registers are accessible to software and therefore can also be utilized by application code. The analysis block monitors the internal address and data buses, and triggers the RTOSINT interrupt when a specified bus and mask matches a specified value. Hence, the basic approach for detecting stack overflow will be to configure the analysis block to trigger an interrupt when the *data write address bus* falls within some range prior to the end of a stack. This is illustrated in Figure 1. Since this memory is reserved for stack usage only, a data write within the specified address range indicates that the stack usage is approaching its allocated size limit.

The stack overflow detection technique described in this report applies applications using the DSP/BIOS real-time operating system, as well as non-DSP/BIOS applications. In the case of a non-DSP/BIOS application, there is generally only a single stack (e.g., the stack used by the C/C++ compiler). A single watchpoint monitoring the stack is therefore sufficient. In the case of a DSP/BIOS application, there are multiple stacks: a single system stack (analogous to the single stack used by non-DSP/BIOS applications), and a stack for each DSP/BIOS task object. Here, a single watchpoint will be configured to monitor the system stack, and a second watchpoint will be configured to monitor the task stacks. This second watchpoint will be dynamically reconfigured each time a task switch occurs such that it is always monitoring the stack of the currently active task.

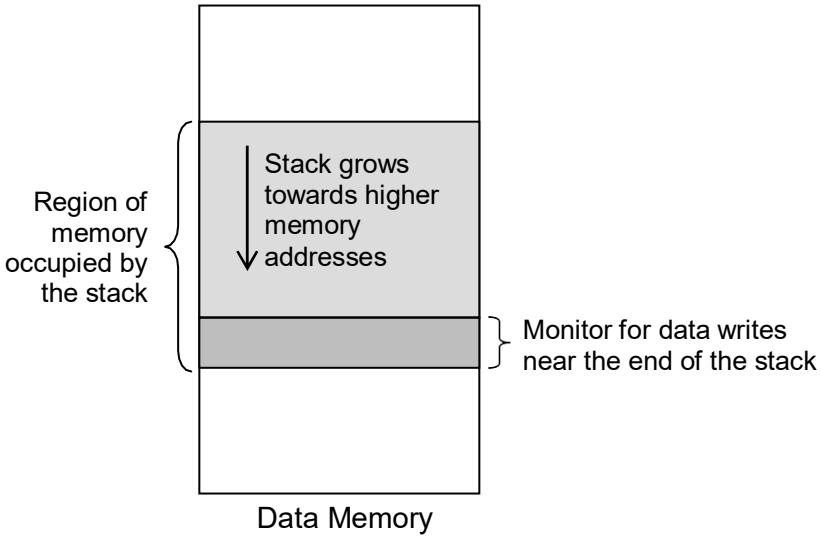


**Figure 1. Stack Overflow Monitoring**

## 2 The C28x Emulation Analysis Block

The emulation analysis block in the C28x DSP core has extensive capabilities, most of which are intended for use by the Code Composer Studio debugger. Documenting the entire analysis block is beyond the scope of this application report, and therefore only the resources needed to perform stack overflow detection will be covered. Additional information can be found in reference [1].

Performing stack overflow detection requires use of one or both of the *analysis units* in the analysis block. The analysis units are utilized as hardware watchpoints (denoted as WP0 for watchpoint 0, and WP1 for watchpoint 1). A watchpoint triggers when either an address bus, or both an address bus and a data bus match what they are being compared against. The address portion is compared against a reference address and bit mask, and the data portion is compared against a reference data value and a different bit mask. If only addresses are to be compared, two watchpoints can be set; if both address and data are to be compared, only one watchpoint can be set. When an emulator is connected to the DSP and Code Composer Studio is active, a triggered watchpoint can cause the debugger to take certain action (e.g. halt the DSP). However, a triggered watchpoint also causes a RTOSINT interrupt, and this interrupt can be utilized by software even if no emulator is connected. Note that the RTOSINT is a maskable interrupt. It must be enabled in the interrupt enable register (IER), and the global interrupt mask bit must be cleared in status register 1 (INTM bit in ST1), or the interrupt will not be serviced.

## 2.1 Analysis Block Watchpoint Registers

The analysis block registers are data memory mapped, and may be accessed from code in the same way as any other data variable. A description of the various registers and their bits follows.

**CAUTION:**

**Read the register bit descriptions very carefully. Some of the bit settings are not intuitive. Also, whereas the bit fields for the MASKxx and REFxx registers of WP0 and WP1 are the same, subtle differences exist in the bit fields for the EVT0_CNTL and EVT1_CNTL registers (such as the buses selected by bits 4-2), and also for the EVT0_ID and EVT1_ID registers.**

**Table 1. C28x Analysis Block Watchpoint Registers**

| Address | Register | Description |
|---|---|---|
| 0x0828 | MASK1L | Lower 16-bits of WP1 address mask |
| 0x0829 | MASK1H | Upper 16-bits of WP1 address mask |
| 0x082A | REF1L | Lower 16-bits of WP1 base address |
| 0x082B | REF1H | Upper 16-bits of WP1 base address |
| 0x082E | EVT1_CNTL | WP1 event control register |
| 0x082F | EVT1_ID | WP1 event ID register |
| 0x0848 | MASK0L | Lower 16-bits of WP0 address mask |
| 0x0849 | MASK0H | Upper 16-bits of WP0 address mask |
| 0x084A | REF0L | Lower 16-bits of WP0 base address |
| 0x084B | REF0H | Upper 16-bits of WP0 base address |
| 0x084E | EVT0_CNTL | WP0 event control register |
| 0x084F | EVT0_ID | WP0 event ID register |

Note: All of the above registers are EALLOW protected.

### MASK0L and MASK1L Registers

Bits 15-0: Contain the lower 16-bits of the corresponding watchpoint address mask. Set address bits to be masked (i.e., ignored) to 1, all non-masked (i.e., used) bits to 0.

### MASK0H and MASK1H Registers

Bits 15-0: Contain the upper 16-bits of the corresponding watchpoint address mask. Set address bits to be masked (i.e., ignored) to 1, all non-masked (i.e., used) bits to 0.

### REF0L and REF1L Registers

Bits 15-0: Contain the lower 16-bits of the corresponding watchpoint address. Set address bits being masked (i.e., ignored) to 1, all other bits to the desired address value.

### REF0H and REF1H Registers

Bits 15-0: Contain the upper 16-bits of the corresponding watchpoint address. Set address bits being masked (i.e., ignored) to 1, all other bits to the desired address value.

### EVT0_CNTL Register

Bits 15-13: Write as 000b

Bits 12-11: 00b - Reserved

01b - Write watchpoint

10b - Read watchpoint

11b - Reserved

Bit 10: Write as 0

Bit 9: Set to 1 to require the external event qualifier, DEXTQ, to be active. Else set to 0.

Bit 8: Write as 0

Bit 7: Set to 1 to automatically re-arm the watchpoint after triggering. Set to 0 for single-shot operation.

Bit 6: Write as 1

Bit 5: Write as 0

Bits 4-2: 000b - Monitor program reads and writes on the program address bus (PAB)

001b - Monitor data reads on the data read address bus (DRAB)

010b - Monitor data writes on the data write address bus (DWAB)

All other bit settings are reserved.

Bits 1-0: 00b - Release an owned watchpoint

01b - Claim ownership of the watchpoint (or disable an owned watchpoint)

10b - Enable an owned watchpoint

11b - Reserved

**EVT1_CNTL Register**

Bits 15-13: Write as 000b

Bits 12-11: 00b - Reserved

01b - Write watchpoint

10b - Read watchpoint

11b - Reserved

Bit 10: Write as 0

Bit 9: Set to 1 to require the external event qualifier, AEXTQ, to be active. Else set to 0.

Bit 8: Write as 0

Bit 7: Set to 1 to automatically re-arm the watchpoint after triggering. Set to 0 for single-shot operation.

Bit 6: Write as 1

Bit 5: Write as 0

Bits 4-2: 010b - Monitor program reads and writes on the program address bus (PAB)

100b - Monitor data reads on the data read address bus (DRAB)

110b - Monitor data writes on the data write address bus (DWAB)

All other bit settings are reserved.

Bits 1-0: 00b - Release an owned watchpoint

01b - Claim ownership of the watchpoint (or disable an owned watchpoint)

10b - Enable an owned watchpoint

11b - Reserved

**EVT0_ID Register (read-only)**

Bits 15-14: 00b - Watchpoint is unclaimed

01b - The application software owns the watchpoint

10b - The debugger owns the watchpoint

11b - Reserved

Bits 13-0: These bits will always read as 0x1002 (01 0000 0000 0010b)

**EVT1_ID Register (read-only)**

Bits 15-14: 00b - Watchpoint is unclaimed

01b - The application software owns the watchpoint

10b - The debugger owns the watchpoint

11b - Reserved

Bits 13-0: These bits will always read as 0x1001 (01 0000 0000 0001b)

## 2.2 Watchpoint Register Configuration Procedure

Analysis block resources may be used by both the application and the Code Composer Studio debugger. To avoid resource contention, the following protocol must be followed by application software when making use of the watchpoints in the analysis block.

1. Execute an EALLOW assembly instruction to enable writes to the analysis block registers. With C code, one should use inline assembly:

```
asm(" EALLOW");
```

2. Set EVTx_CNTL[1:0] to 01b to attempt to claim ownership of the watchpoint.

3. Wait at least three cycles for the write to EVTx_CNTL[1:0] to occur in the CPU pipeline. During this time, instructions that don't involve accessing the analysis block registers can be executed, or more simply just execute three NOP instructions. The most compact C code to do this uses inline assembly:

```
asm(" RPT #1 || NOP");       /* 3 cycles, 2 words */
```

4. Read the EVTx_ID register and verify that the application is the owner of the watchpoint by checking bits 15-14. The application must be the owner before proceeding. If the application is not the owner, software must either retry from step #2, or abort the attempt to setup the watchpoint (depends on how the user would like to handle this in software). When the emulator is not being used (i.e. the final product, after code development), there is no reason why the application should not achieve ownership. When the emulator is being used (e.g. during code development) the application could fail to achieve ownership if the Code Composer Studio debugger is already using the emulation analysis unit that corresponds to the requested watchpoint. Appendix C provides some troubleshooting assistance in the event that a Code Composer Studio conflict does arise.

5. Once the application owns the watchpoint, the registers for that watchpoint can be programmed. Specifically, REFxL, REFxH, MASKxL, MASKxH, and EVTx_CNTL must be configured. The last register one should configure is the EVTx_CNTL register, where bits 1-0 should be set to 10b to enable the watchpoint. Note that if the application does not own the watchpoint, software writes to all the watchpoint registers are ignored.

6. Execute an EDIS assembly instruction to disable writes to the analysis block registers. With C code, one should use inline assembly:

```
asm(" EDIS");
```

# 3 Configuring a Watchpoint for Stack Overflow Detection

The C28x emulation analysis block provides for the monitoring of the data read address bus or the data write address bus. Since the stack pointer (SP) uses these two buses when making data accesses to the stack, stack overflow detection can be performed by configuring a watchpoint to monitor for data write activity occurring near the end of stack memory. Watching data write activity as opposed to data read activity is justified since it is reasonable to assume that properly functioning code would perform a push onto the stack (i.e., a write operation) before attempting to pop that data off the stack (i.e., a read operation). Erroneously reading data from locations past the end of the stack when no previous write has occurred, although certainly a problem, is more indicative of a code bug rather than a stack overflow.

## 3.1 Determining the Watchpoint Location and Range in Memory

Ideally, one would want the watchpoint to trigger whenever a write occurred to the stack memory at an address greater than some reference address (as opposed to exactly matching the reference address). The reference address would be set some number of words before the actual end of the stack in order to provide the application with sufficient time to take corrective action. In this way, any stack activity in the memory range between the reference address and the actual stack end would trigger the watchpoint. However, the C28x watchpoints do not allow the specification of an arbitrary address range. Rather, they allow only a reference address (REFxL and REFxH registers) and a bit mask (MASKxL and MASKxH registers) to be applied to this address. Therefore, one must utilize the bit mask to implement a suitable address range.

A bit mask provides the ability to specify a range of size 2^N, aligned on a N-bit boundary in memory (where N = 0, 1, 2, ...). With this in mind, let's examine the two quantities needed in order to locate the watchpoint range in memory: the starting address of the range prior to the stack end, and the size of the range. Let's first consider the starting address of the range. Once the watchpoint triggers, there must be enough space left on the stack for any stack pushes that might occur before the RTOSINT is serviced (at which point the software takes corrective action). The C28x DSP automatically saves 14 registers (16-bit words) onto the stack when an interrupt occurs. The worst-case situation then is when an interrupt occurs, and the first of the automatically saved words causes the watchpoint to trigger. The stack must have sufficient space for the following:

- The 14 automatically saved words for the triggering interrupt
- The 14 automatically saved words for the RTOSINT caused by the watchpoint
- Up to 6 32-bit stack writes (pushes) that could already be in the CPU pipeline when the triggered interrupt occurs. This equates to 12 words of space. Although the likelihood of having 6 consecutive 32-bit pushes in source code is small, it is a worse-case situation none the less.
- Any stack space needed by the RTOSINT interrupt service routine (ISR)

Adding together these requirements, one sees that the watchpoint should trigger a minimum of 40 words prior to the end of a stack (neglecting the requirements of the RTOSINT ISR, which is software specific). Keep in mind that the aim of stack overflow detection is to provide a last line of defense against unforeseen stack overflow. The needed stack sizes for an application should have previously been determined during code development using, for example, the offline approach previously discussed in this report. The stack overflow detection discussed here is not intended to be part of the normal software flow. In general, the actions taken by the stack overflow routine will be relatively drastic (e.g. safe shutdown). Hence, one will not want the watchpoint to trigger too much earlier than is needed to safely take corrective action before stack overflow occurs. Setting the watchpoint too far before the end of the stack simply wastes memory, as the memory occurring at addresses after the watchpoint cannot be utilized by the application.

The size of the watchpoint range must be large enough to ensure that accesses into the stack don't accidentally skip over the monitored addresses (e.g., because of manual stack pointer manipulation that skips some number of stack locations, or 32-bit stack pushes where only the even address goes out on the address bus). The downside of using too large a range is that the start of the range must be aligned on an N-bit boundary, and therefore larger ranges may need to have their starting address shifted significantly from the specified reference address in order to achieve alignment. This can further waste memory by reducing the amount of available stack space before the watchpoint triggers. In general, stack growth on the C28x DSP is sequential, with no skipped spaces, so a fairly small range size may be used. It is recommended that a range size of 8 or 16 be used (recall that the range size is specified as a bit mask, and therefore must be of size 2^N). This is small enough to avoid significant alignment shifting, and large enough to handle unforeseen stack pointer manipulation issues.

Let's look at a configuration example:

stack start address = 0x00008123

stack end address = 0x00008523 (i.e., stack length is 0x400 words)

specified overflow range = 8 (i.e., the range mask is 0x0007)

specified range starting distance from end of stack = 45 words

The calculation to determine the aligned starting address of the watchpoint range is the stack end address minus the specified range starting distance, and then AND'd with the Boolean 1's compliment of the range mask. In other words:

aligned address = (stack end - specified range starting distance) AND (~mask)

= (0x00008523 - 45) & (~0x0007)

= (0x000084F6) & (0xFFF8)

= 0x000084F0

Hence, the watchpoint will monitor addresses in the range 0x000084F0 to 0x000084F7, inclusive. Figure 2 shows the relationship between the various addresses and the watchpoint range in memory. Notice that the starting address of the range is actually 51 words before the end of the stack. This is 6 words more than the specified distance of 45 words. The additional 6 words were needed to keep the monitored range aligned on an N-bit boundary.

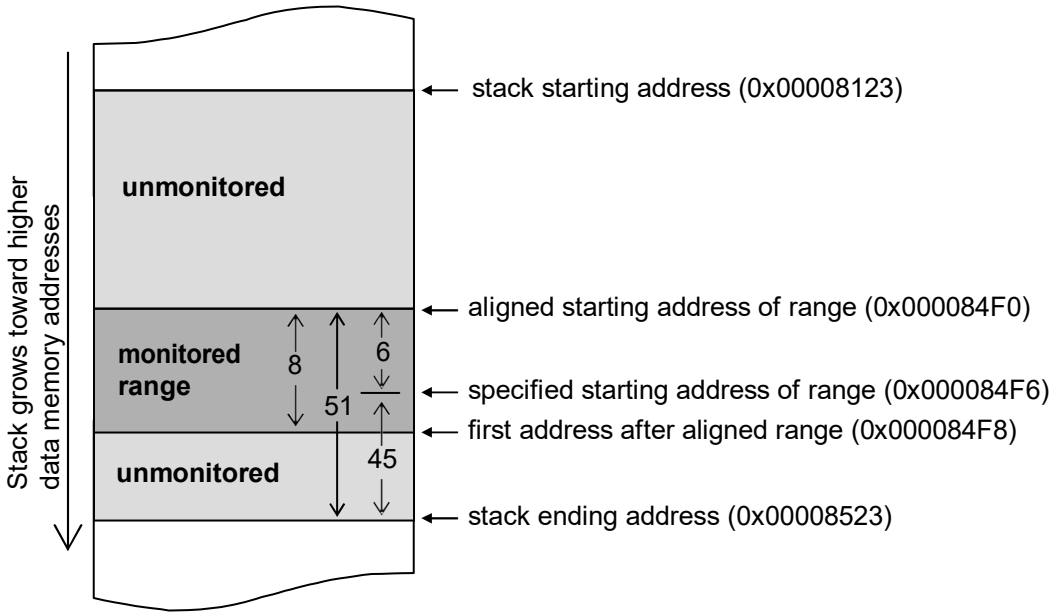


**Figure 2. Watchpoint Range in Relation to the Stack in Memory**
(numerical values refer to specific example given in the text)

## 3.2 Watchpoint Registers Values

Given a desired aligned starting address and range mask, the following section presents the needed bit settings in the relevant watchpoint registers.

*MASKxL and MASKxH Registers*

The range mask is simply written to the MASKxL and MASKxH registers as is. Since these registers occupy consecutive addresses in the memory map, a single 32-bit write can be used to write the range mask.

*REFxH and REFxL Registers*

The aligned reference address should be OR'd with the mask range, and then written to the REFxH and REFxL registers. This is because of how the reference registers are designed: all address bits that will be masked by the contents of the mask registers should be written as 1's. Since these registers occupy consecutive addresses in the memory map, a single 32-bit write can be used to write the range mask.

*EVTx_CNTL Register*

EVTx_CNTL[15:13]: These bits should always be written as zeros.

EVTx_CNTL[12:11]: These bits should be configured for "Write watchpoint" (01b).

EVTx_CNTL[10]: This bit should always be written as zero.

EVTx_CNTL[9]: Setting this bit to 1 gates the watchpoint trigger with an additional external event qualifier signal: DEXTQ for EXT0_CNTL, and AEXTQ signal for EVT1_CNTL. The intent of this is to allow an external device to control the watchpoint activity (e.g., to keep some time critical code from getting interrupted). However, these signals are internally tied off on in an inactive state on the TMS320F2812 and TMS320F2810 DSP devices. Therefore, this bit should be set to 0 (or the watchpoint trigger will never occur). If using a different C28x device, consult the device datasheet to see if DEXTQ and AEXTQ signals are pinned out in the (unusual) event that this capability is needed.

EVTx_CNTL[8]: This bit should always be written as zero.

EVTx_CNTL[7]: In general, this bit should be set to a 1 for single-shot operation of the watchpoint. The application can re-arm the watchpoint, if desired, after it takes corrective action.

EVTx_CNTL[6]: This bit should always be written as one.

EVTx_CNTL[5]: This bit should always be written as zero.

EVTx_CNTL[4:2]: Since writes to the stack are of interest here, and since these writes are always performed as data memory writes (as opposed to program memory writes using the PWRITE assembly code instruction), one should set these bits to monitor the data write address bus (DWAB). The binary value for this settings is different between WP0 (use 010b for data write) and WP1 (use 110b for data write). Be sure to carefully check the required setting in Section 2.1.

EVTx_CNTL[1:0]: The description for these bits is self-explanatory. Set to 01b to attempt to claim ownership of the watchpoint (or to disable an already owned watchpoint). Set to 10b to enable an owned watchpoint. There is generally little need for an application to release an owned watchpoint (the 00b setting).

# 4 Application Issues

Instructions have so far been provided on how to configure the watchpoint for stack overflow detection given the stack end address. What remains to be shown is how user code can determine the stack end address at run time, and in the case of DSP/BIOS applications, how to configure the watchpoints to handle the multiple stacks used by the software.

## 4.1 Non-DSP/BIOS Applications

The C28x C/C++ compiler employs a single stack. During the C-environment setup performed by the compiler runtime support library (e.g., rts2800.lib or rts2800_ml.lib), the SP is initialized to the beginning of the allocated stack memory, and remains pointed to somewhere in the stack memory throughout execution of the code. Barring any manual relocation of the SP by the user at the assembly code level, it is sufficient to statically configure a single watchpoint to monitor the end of the C/C++ stack.

The C/C++ compiler allocates the stack in a section called *.stack*. Since the link address of the .stack section is specified in the linker command file of the code project, and since the stack length is specified by the user as a Code Composer Studio project option, one could compute the end address of the stack and hard-code the address at which to set the watchpoint into their application. However, this is not a particularly elegant nor easily maintainable solution. A better method is to have the linker generate a symbol for the ending address of the .stack section, and then have software utilize this symbol. It may also be useful to have a symbol for the starting address of the stack section for error correction purposes (e.g., to determine if any portion of the watchpoint range lies outside the stack memory). The C28x linker has the capability to automatically generate such global symbols as follows. Suppose one wants to link the .stack section to some memory called "RAM" that has already been defined on PAGE 1 in the MEMORY section of the linker command file. The following SECTIONS entry in the linker command file will achieve this:

```
SECTIONS
{
    .stack:    RUN = RAM,
               RUN_START(_HWI_STKBOTTOM),
               RUN_END(_HWI_STKTOP),
               PAGE = 1
}
```

The above defines symbols *_HWI_STKBOTTOM* and *_HWI_STKTOP* representing the start and end addresses in memory of the .stack section (these symbol names are just examples, and the reader is free to change them). Note that the end address is actually the first address after the last word of the stack. Three items of importance:

1. Notice the use of the leading underscore in the symbol names. This allows the symbols to be accessed from C code using just *HWI_STKBOTTOM* and *HWI_STKTOP*, as C automatically appends a leading underscore to all symbol names.

2. These symbols actually represent the 16-bit values that exist on the stack at its starting and ending addresses. It is the addresses of these symbols that represent the start and end address of the stack. Therefore, an ampersand should be used in C code to denote the symbol address, e.g. *&HWI_STKBOTTOM*.

3. To access these symbols from C source code, they must be declared as external global symbols for unsigned integer (16 bit) values in any source file that uses them.

A simple example of how to incorporate these symbols in C source code is as follows.

```
/* C28x DSP C Code example to access the stack symbols */

extern unsigned int HWI_STKBOTTOM;
extern unsigned int HWI_STKTOP;

void MyFunc(void)
{
     unsigned long x,y;

     x = (unsigned long)&HWI_STKBOTTOM;  /* assign x the address of the stack start */
     y = (unsigned long)&HWI_STKTOP;     /* assign x the address of the stack end */
}
```

Note that when the address (32 bits) of *HWI_STKTOP* is assigned to the variable x, one must typecast the address to an unsigned long since *&HWI_STKTOP* is of the type unsigned 16-bit integer pointer, but x is of type unsigned long.

The C function *STKOV_initSystemStack()* described in Appendix A and found in Appendix B.1 will configure and enable a watchpoint to monitor for overflow of the C/C++ stack. This function can be directly incorporated into user code.

## 4.2 DSP/BIOS Applications

DSP/BIOS employs multiple stacks. Hardware interrupts (HWIs) and software interrupts (SWIs) use the system stack, which is analogous to the single stack employed by non-DSP/BIOS C/C++ applications. A single watchpoint statically configured will effectively handle overflow detection for the system stack. The DSP/BIOS configuration tool generates a linker command file that defines global variables for the stack start and end addresses: HWI_STKBOTTOM and HWI_STKTOP, respectively (accurate as of Code Composer Studio v2.20). These symbols can be used with the C function *STKOV_initSystemStack()* as described in Section 4.1.

DSP/BIOS tasks (TSKs) each have their own stack. When the DSP/BIOS scheduler prepares to run a task, it changes the SP to point to the stack of that task. Therefore, a static configuration of a watchpoint will not suffice here. Instead, the remaining C28x watchpoint must be dynamically reconfigured to monitor the stack of whichever task is active at the moment (recall that there are two available watchpoints on the C28x, and one is already in use to monitor the system stack).

DSP/BIOS provides support for this dynamic reconfiguration in the form of the *task switch hook function*. The task switch hook function is specified by the user and is run each time a task switch occurs. DSP/BIOS passes to this function a handle (i.e., a pointer) for the task being switched to. Using this handle, the switch hook function can access information about the stack starting address and length for the task, and then dynamically change the watchpoint to monitor this stack.

Some static computations are also useful for task stack monitoring in order to avoid repeatedly calculating unchanging values in the task switch hook function (which would impact the cycle efficiency of the switch function). Specifically, the address at which the watchpoint should be set for each task needs to be calculated only once. The address information must then be stored in the environment of the task[†]. DSP/BIOS provides support for the static computations in the form of the *task create hook function*. Performing the watchpoint address calculation once at task creation time provides for a more cycle efficient task switch hook function.

Figure 3 shows how to specify the task switch hook and task create hook functions in the DSP/BIOS configuration tool. More information on DSP/BIOS hook functions can be found in the online help within Code Composer Studio.

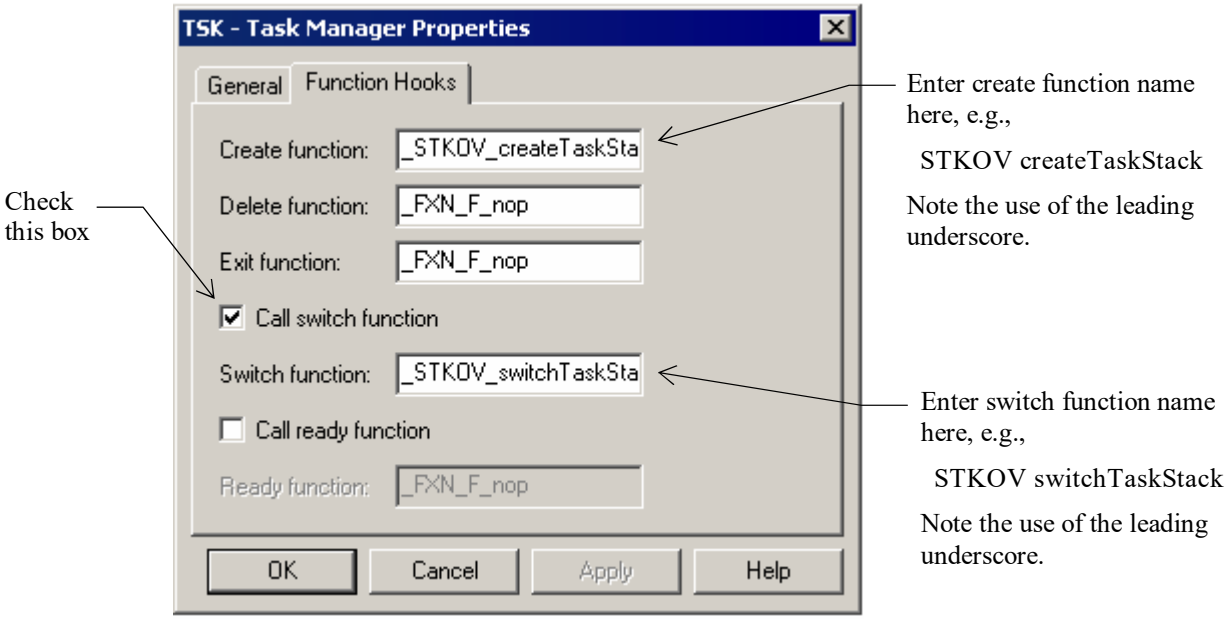


**Figure 3. Specifying the Task Hook Functions in Code Composer Studio v2.20**

[t] All DSP/BIOS task objects contain a pointer to the environment of that task. The task environment is a user defined global data structure. To implement the task stack overflow detection scheme herein described, it could be required that the environment of each task contain the watchpoint address as one of the elements. However, since only a single environment element is needed for each task (i.e., the watchpoint address), it is most efficient to use the environment pointer itself as the actual value. In other words, the environment pointer of each task object is assigned a value equal to the watchpoint address for the task. Additional environment space is therefore not needed, and the task switch function is also more efficient since the watchpoint address is easily accessed as the environment pointer of the task object. This is the approach used by the code provided in this application report. If the reader's application requires use of the task environment, the code presented in this report is easily modified to allow for this.

The C functions *STKOV_createTaskStack(), STKOV_initTaskStack(),* and *STKOV_switchTaskStack()* described in Appendix A and found in Appendix B.2 will configure and enable a watchpoint to monitor for overflow of DSP/BIOS task stacks. These functions can be directly incorporated into user code. *STKOV_createTaskStack()* is the task create hook function, *STKOV_switchTaskStack()* is the task switch hook function, and *STKOV_initTaskStack()* performs some required static initialization before either of the other two functions can be run.

If the RTOSINT is triggered by stack overflow, and both the system stack and task stacks are being monitored, the user might want to determine which stack overflowed in the RTOSINT ISR order to take appropriate corrective action. Unfortunately, there is no hardware flag available that differentiates between WP0 and WP1. Instead, one approach would be to read the SP in the ISR, and then compare against the settings in the REFH and REFL registers for the two watchpoints.

# 5 Conclusion

A method for online stack overflow detection on the TMS320C28x DSP has been presented. The approach entails configuring an emulation analysis block watchpoint to monitor bus activity in an address range near the end of a stack. For non-DSP/BIOS application code, a single, statically configured watchpoint monitoring the C stack is sufficient. For DSP/BIOS application code, two watchpoints are needed: a static watchpoint for the system static, and a dynamically re-configured watchpoint to monitor task stacks. The task stack watchpoint is reconfigured using the task switch hook function of DSP/BIOS. C-source code has been provided that contains functions for implementing the overflow detection.

# 6 References

1. *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430)

# Appendix A. C Function APIs

**STKOV_initSystemStack** Initialization function for monitoring the system (C/C++) stack

**Source File** stkov_systemstack.c

**Include Files** stkov.h

**Function Prototype**

```
unsigned int error STKOV_initSystemStack(
                        unsigned long stackStartAddr,
                        unsigned long stackEndAddr,
                        unsigned int margin);
```

**Arguments**

| | |
|---|---|
| stackStartAddr: | Address of first word in stack. |
| stackEndAddr: | Address of first word after the last word in the stack. For example, if the stack starts at 0x100, and is of length 0x80, then stackEndAddr = 0x180. |
| margin: | The minimum number of words before StackEndAddr to set the watchpoint at. |

**Return Value**

| | |
|---|---|
| error: | 0 = no error<br>1 = software failed to gain control of the watchpoint<br>2 = watchpoint range falls outside the stack |

**Description**

This function configures and enable a watchpoint to monitor the system stack (for DSP/BIOS applications) or the C/C++ stack (for non-DSP/BIOS applications). It also enables the RTOSINT interrupt. The function should be called once during system initialization, generally in main().

The source file contains two #define constants that affect this function. These constants may be changed if desired. The constants are:

WP: The watchpoint to use (valid values are 0 or 1). Default value is 0. If only one watchpoint is in use for stack monitoring (e.g., task stack monitoring is not being used), it is recommended to use WP = 0 since the Code Composer Studio debugger uses watchpoint 1 for numerous debug features. Note that the watchpoints used for system and task stack monitoring must be different.

STKOV_RANGEMASK: Mask specifying the range covered by the watchpoint. Default is 0x0007 (range is 8 words). Note that the value must be (2^N -1) in form, e.g., 0x0001, 0x0003, 0x0007, 0x000F, 0x001F, etc.

If any value other than 0 is returned, it means that the watchpoint was not enabled.

**Example**

```
#define margin 45
extern unsigned int HWI_STKBOTTOM, HWI_STKTOP;
unsigned int error;
error = STKOV_initSystemStack( (Uint32)&HWI_STKBOTTOM,
                               (Uint32)&HWI_STKTOP,
                               margin);
```

**STKOV_createTaskStack** DSP/BIOS task create hook function for task stack monitoring

**Source File** stkov_taskstack.c

**Include Files** stkov.h

**Function Prototype** void STKOV_createTaskStack(TSK_Handle task);

**Arguments** task: Handle for the task being created.

**Return Value** none

**Description** This function computes the address that the watchpoint should be set to for monitoring the stack of a task. It should be setup to serve as the DSP/BIOS task create hook function using the DSP/BIOS Configuration tool. DSP/BIOS will then execute this function each time a task is created.

The source file contains two #define constants that affect this function. These constants may be changed if desired. The constants are:

STKOV_MARGIN:
The minimum number of words before the end of the task stack to set the watchpoint at. Default is 45.

STKOV_RANGEMASK:
Mask specifying the range covered by the watchpoint. Default value is 0x0007 (range is 8 words). Note that the value must be (2^N -1) in form, e.g., 0x0001, 0x0003, 0x0007, 0x000F, 0x001F, etc.

**Example** Not applicable. This function should be specified as the DSP/BIOS task create hook function inside Code Composer Studio.

**STKOV_initTaskStack** Initialization function for monitoring the DSP/BIOS task stacks

**Source File** stkov_taskstack.c

**Include Files** stkov.h

**Function Prototype** unsigned int error STKOV_InitTaskStack(void);

**Arguments** none

**Return Value**

error: 0 = no error
1 = software failed to gain control of the watchpoint

**Description**

This function reserves a watchpoint to monitor the DSP/BIOS task stacks. It also configures the static portion of the watchpoint, and enables the RTOSINT. This function should be called once during system initialization (before any DSP/BIOS tasks are executed), generally in main().

The source file contains two #define constants that affect this function. These constants may be changed if desired. The constants are:

WP: The watchpoint to use (valid values are 0 or 1). Default value is 0. If only one watchpoint is in use for stack monitoring (e.g., system stack monitoring is not being used), it is recommended to use WP = 0 since the Code Composer Studio debugger uses watchpoint 1 for numerous debug features. Note that the watchpoints used for system and task stack monitoring must be different.

STKOV_RANGEMASK: Mask specifying the range covered by the watchpoint. Default value is 0x0007 (range is 8 words). Note that the value must be (2^N -1) in form, e.g., 0x0001, 0x0003, 0x0007, 0x000F, 0x001F, etc.

**Example**

```
unsigned int error;
error = STKOV_initTaskStack();
```

**STKOV_switchTaskStack** DSP/BIOS task switch hook function for task stack monitoring

**Source File** stkov_taskstack.c

**Include Files** stkov.h
std.h (part of the TI C compiler runtime support library)
tsk.h (part of the TI DSP/BIOS software)

**Function Prototype**
```
void STKOV_switchTaskStack(TSK_Handle oldtask,
                           TSK_Handle newtask);
```

**Arguments**

| | |
|---|---|
| oldtask: | Handle to old task (task being switched from) |
| newtask: | Handle to new task (task being switched to) |

**Return Value** none

**Description** This function switches a watchpoint to monitor the stack of the new task. It should be setup to serve as the DSP/BIOS task switch hook function using the DSP/BIOS Configuration tool. DSP/BIOS will then execute this function each time a task switch occurs.

The source file contains one #define constant that affects this function. This constant may be changed if desired. The constant is:

| | |
|---|---|
| WP: | The watchpoint to use (valid values are 0 or 1). Default value is 0. If only one watchpoint is in use for stack monitoring (e.g., system stack monitoring is not being used), it is recommended to use WP = 0 since the Code Composer Studio debugger uses watchpoint 1 for numerous debug features. Note that the watchpoints used for system and task stack monitoring must be different. |

Note that this function has been carefully written to use as little stack space as possible (by using few local variables and by using immediate valued pointers). This is motivated by that fact that this function will use the stack from the old task. Therefore, *every* task stack will need to be of sufficient size to handle needs of this switch function, in addition to everything else that goes on the task stack (e.g., the tasks local context, hardware interrupt context switching, etc.). This function has also been written to be as cycle efficient as possible (since it is run each time a task is switched). The cycle count of this function, including the function call and return, is approximately 60 cycles (with or without compiler optimization). Some additional overhead (e.g. maybe 10 to 20 cycles) in the DSP/BIOS scheduler is also incurred since it passes two parameters to this function.

**Example** Not applicable. This function should be specified as the DSP/BIOS task switch hook function inside Code Composer Studio.

# Appendix B. C Code Functions

## B.1 stkov_systemstack.c

```
/*************************************************************************
* File: stkov_systemstack.c                                              *
* Device: TMS320C28x                                                     *
* Author: David M. Alter, Texas Instruments Inc.                         *
* History:                                                               *
*   May 1, 2003 - Original (D. Alter)                                    *
*************************************************************************/
/*************************************************************************
* THIS PROGRAM IS PROVIDED "AS IS".  TI MAKES NO WARRANTIES OR           *
* REPRESENTATIONS, EITHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING       *
* ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A               *
* PARTICULAR PURPOSE, LACK OF VIRUSES, ACCURACY OR COMPLETENESS OF       *
* RESPONSES, RESULTS AND LACK OF NEGLIGENCE.  TI DISCLAIMS ANY           *
* WARRANTY OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, AND              *
* NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS       *
* WITH REGARD TO THE PROGRAM OR YOUR USE OF THE PROGRAM.                 *
*                                                                        *
* IN NO EVENT SHALL TI BE LIABLE FOR ANY SPECIAL, INCIDENTAL,            *
* CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY THEORY       *
* OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED OF THE             *
* POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF THIS            *
* AGREEMENT, THE PROGRAM, OR YOUR USE OF THE PROGRAM.  EXCLUDED          *
* DAMAGES INCLUDE, BUT ARE NOT LIMITED TO, COST OF REMOVAL OR            *
* REINSTALLATION, COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS     *
* OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF         *
* BUSINESS.  IN NO EVENT WILL TI'S AGGREGATE LIABILITY UNDER THIS        *
* AGREEMENT OR ARISING OUT OF YOUR USE OF THE PROGRAM EXCEED FIVE        *
* HUNDRED DOLLARS U.S.$500).                                             *
*                                                                        *
* Unless otherwise stated, the Program written and copyrighted by       *
* Texas Instruments is distributed as "freeware".  You may, only         *
* under TI's copyright in the Program, use and modify the Program        *
* without any charge or restriction.  You may distribute to third        *
* parties, provided that you transfer a copy of this license to the      *
* third party and the third party agrees to these terms by its first     *
* use of the Program.  You must reproduce the copyright notice and       *
* any other legend of ownership on each copy or partial copy, of the     *
* Program.                                                               *
*                                                                        *
* You acknowledge and agree that the Program contains copyrighted        *
* material, trade secrets and other TI proprietary information and       *
* is protected by copyright laws, international copyright treaties,      *
* and trade secret laws, as well as other intellectual property          *
* laws.  To protect TI's rights in the Program, you agree not to         *
* decompile, reverse engineer, disassemble or otherwise translate        *
* any object code versions of the Program to a human-readable form.      *
* You agree that in no event will you alter, remove or destroy any       *
* copyright notice included in the Program.  TI reserves all rights      *
* not specifically granted under this license. Except as                 *
* specifically provided herein, nothing in this agreement shall be       *
* construed as conferring by implication, estoppel, or otherwise,        *
* upon you, any license or other right under any TI patents,             *
* copyrights or trade secrets.                                           *
*                                                                        *
* You may not use the Program in non-TI devices.                         *
*************************************************************************/
```

```
// Choose which watchpoint to use (User configurable)
#define WP 0                                // Valid values are 0 or 1 (Default is 0)

// Address and value definitions for Emulation Watchpoint Registers
#if WP == 0
    #define WP_MASK      (volatile unsigned long *)0x00000848  // WP0 MASK register addr
    #define WP_REF       (volatile unsigned long *)0x0000084A  // WP0 REF register addr
    #define WP_EVT_CNTL  (volatile unsigned  int *)0x0000084E  // WP0 EVT_CNTL register addr
    #define WP_EVT_ID    (volatile unsigned  int *)0x0000084F  // WP0 EVT_ID register addr
    #define EVT_CNTL     0x080A                                // EVT_CNTL value for WP0
#else
    #define WP_MASK      (volatile unsigned long *)0x00000828  // WP1 MASK register addr
    #define WP_REF       (volatile unsigned long *)0x0000082A  // WP1 REF register addr
    #define WP_EVT_CNTL  (volatile unsigned  int *)0x0000082E  // WP1 EVT_CNTL register addr
    #define WP_EVT_ID    (volatile unsigned  int *)0x0000082F  // WP1 EVT_ID register addr
    #define EVT_CNTL     0x081A                                // EVT_CNTL value for WP1
#endif

#define STKOV_RANGEMASK  0x0007             // 0x0007 = trigger range is 8 words

// Other Definitions
extern cregister volatile unsigned int IER;


/*********************************************************************
* Function: STKOV_initSystemStack()                                  *
* Description: Configures a hardware watchpoint to trigger an        *
*   RTOSINT on write access in a specified range at the end of the   *
*   system (or C/C++) stack.                                         *
* DSP: TMS320C28x                                                    *
* Include files: none                                                *
* Function Prototype:                                                *
*   unsigned int STKOV_initSystemStack(                              *
*                    unsigned long, unsigned long, unsigned int);    *
* Usage: error = STKOV_initSystemStack(                              *
*                    stackStartAddr, stackEndAddr, margin);          *
* Input Parameters:                                                  *
*   unsigned long stackStartAddr = Address of first word in stack.   *
*   unsigned long stackEndAddr = Address of first word after last    *
*     word in stack.  For example, if the stack starts at 0x100, and *
*     is of length 0x80, then StackEndAddr = 0x180.                  *
*   unsigned int margin = The minimum number of words before         *
*     stackEndAddr that the watchpoint is to be set at.              *
* Return Value:                                                      *
*    unsigned int error:                                             *
*      0 = no error                                                  *
*      1 = software failed to gain control of the WP                 *
*      2 = the watchpoint range falls outside the stack              *
* Notes:                                                             *
*   1) If any value other than 0 is returned, it means that the      *
*      overflow detection was not enabled.                           *
*********************************************************************/
unsigned int STKOV_initSystemStack(unsigned long stackStartAddr,
                                   unsigned long stackEndAddr,
                                   unsigned int margin)
{
unsigned long addr;                         // Address to set the WP at

// Compute starting address of watchpoint range
    addr = (stackEndAddr - margin) & (unsigned long)(~STKOV_RANGEMASK);
```

```
// Check to be sure the watchpoint range falls within the stack.
    if(addr < stackStartAddr)               // Check if range underruns the stack start
        return(2);                          // Return error code
    if(addr > stackEndAddr)                 // Catch arithmetic underflow
        return(2);                          // Return error code

// Enable EALLOW protected register access
    asm(" EALLOW");

// Attempt to gain control of the watchpoint
    *WP_EVT_CNTL = 0x0001;                  // Write 0x0001 to EVT_CNTL to claim ownership
                                            //   of the watchpoint
    asm(" RPT #1 || NOP");                  // Wait at least 3 cycles for the write to occur

// Confirm that the application owns the watchpoint
    if((*WP_EVT_ID & 0xC000) != 0x4000)     // Software failed to gain control of watchpoint
    {
        asm(" EDIS");                       // Disable EALLOW protected register access
        return(1);                          // Return error code
    }

// Proceed to configure the watchpoint
    *WP_MASK = (unsigned long)STKOV_RANGEMASK;        // Watchpoint reference address mask
    *WP_REF = addr | (unsigned long)STKOV_RANGEMASK;  // Watchpoint reference address
                                                      //   (write all masked bits as 1's)
    *WP_EVT_CNTL = EVT_CNTL;                          // Enable the watchpoint
    IER |= 0x8000;                                    // Enable RTOSINT

// Successful Return
    asm(" EDIS");                           // Disable EALLOW protected register access
    return(0);                              // Return with no error

} //end of STKOV_initSystemStack()

// end of file stkov_systemstack.c
```

## B.2 stkov_taskstack.c

```
/*************************************************************************
* File: stkov_taskstack.c                                                *
* Device: TMS320C28x                                                     *
* Author: David M. Alter, Texas Instruments Inc.                         *
* History:                                                               *
*   May 1, 2003 - Original (D. Alter)                                    *
*************************************************************************/
/*************************************************************************
* THIS PROGRAM IS PROVIDED "AS IS".  TI MAKES NO WARRANTIES OR           *
* REPRESENTATIONS, EITHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING       *
* ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A               *
* PARTICULAR PURPOSE, LACK OF VIRUSES, ACCURACY OR COMPLETENESS OF       *
* RESPONSES, RESULTS AND LACK OF NEGLIGENCE.  TI DISCLAIMS ANY           *
* WARRANTY OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, AND              *
* NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS       *
* WITH REGARD TO THE PROGRAM OR YOUR USE OF THE PROGRAM.                 *
*                                                                        *
* IN NO EVENT SHALL TI BE LIABLE FOR ANY SPECIAL, INCIDENTAL,            *
* CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY THEORY       *
* OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED OF THE             *
* POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF THIS            *
* AGREEMENT, THE PROGRAM, OR YOUR USE OF THE PROGRAM.  EXCLUDED          *
* DAMAGES INCLUDE, BUT ARE NOT LIMITED TO, COST OF REMOVAL OR            *
* REINSTALLATION, COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS     *
* OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF         *
* BUSINESS.  IN NO EVENT WILL TI'S AGGREGATE LIABILITY UNDER THIS        *
* AGREEMENT OR ARISING OUT OF YOUR USE OF THE PROGRAM EXCEED FIVE        *
* HUNDRED DOLLARS U.S.$500).                                             *
*                                                                        *
* Unless otherwise stated, the Program written and copyrighted by       *
* Texas Instruments is distributed as "freeware".  You may, only         *
* under TI's copyright in the Program, use and modify the Program        *
* without any charge or restriction.  You may distribute to third        *
* parties, provided that you transfer a copy of this license to the      *
* third party and the third party agrees to these terms by its first     *
* use of the Program.  You must reproduce the copyright notice and       *
* any other legend of ownership on each copy or partial copy, of the     *
* Program.                                                               *
*                                                                        *
* You acknowledge and agree that the Program contains copyrighted        *
* material, trade secrets and other TI proprietary information and       *
* is protected by copyright laws, international copyright treaties,      *
* and trade secret laws, as well as other intellectual property          *
* laws.  To protect TI's rights in the Program, you agree not to         *
* decompile, reverse engineer, disassemble or otherwise translate        *
* any object code versions of the Program to a human-readable form.      *
* You agree that in no event will you alter, remove or destroy any       *
* copyright notice included in the Program.  TI reserves all rights      *
* not specifically granted under this license. Except as                 *
* specifically provided herein, nothing in this agreement shall be       *
* construed as conferring by implication, estoppel, or otherwise,        *
* upon you, any license or other right under any TI patents,             *
* copyrights or trade secrets.                                           *
*                                                                        *
* You may not use the Program in non-TI devices.                         *
*************************************************************************/
```

```
/*** Include Files ***/
#include <std.h>
#include <tsk.h>

// Choose which watchpoint to use (User configurable)
#define WP 1                              // Valid values are 0 or 1 (Default is 1)

// Address and value definitions for Emulation Watchpoint Registers
#if WP == 0
    #define WP_MASK      (volatile unsigned long *)0x00000848  // WP0 MASK register addr
    #define WP_REF       (volatile unsigned long *)0x0000084A  // WP0 REF register addr
    #define WP_EVT_CNTL  (volatile unsigned  int *)0x0000084E  // WP0 EVT_CNTL register addr
    #define WP_EVT_ID    (volatile unsigned  int *)0x0000084F  // WP0 EVT_ID register addr
    #define EVT_CNTL     0x080A                                // EVT_CNTL value for WP0
#else
    #define WP_MASK      (volatile unsigned long *)0x00000828  // WP1 MASK register addr
    #define WP_REF       (volatile unsigned long *)0x0000082A  // WP1 REF register addr
    #define WP_EVT_CNTL  (volatile unsigned  int *)0x0000082E  // WP1 EVT_CNTL register addr
    #define WP_EVT_ID    (volatile unsigned  int *)0x0000082F  // WP1 EVT_ID register addr
    #define EVT_CNTL     0x081A                                // EVT_CNTL value for WP1
#endif

#define STKOV_MARGIN 45                   // Trigger margin is 45 words
#define STKOV_RANGEMASK 0x0007            // 0x0007 = trigger range is 8 words

/*** Other Definitions ***/
extern cregister volatile unsigned int IER;


/*************************************************************************
* Function: STKOV_createTaskStack()                                      *
* Description: Retrieves a tasks stack start address and length, and *
*   places these into the tasks environment.  This function is       *
*   designed to be the task create hook function in DSP/BIOS.        *
* DSP: TMS320C28x                                                        *
* Include files: std.h, tsk.h                                            *
* Function Prototype:                                                    *
*   void STKOV_createTaskStack(TSK_Handle);                              *
* Usage: STKOV_switchTaskStack(task);                                    *
* Input Parameters:                                                      *
*   TSK_Handle task = handle to task.                                    *
* Return Value: none                                                     *
* Notes:                                                                 *
*************************************************************************/
void STKOV_createTaskStack(TSK_Handle task)
{
static TSK_Stat status;
unsigned long addr;                       // Address to set the WP at

// Get the task attributes
    TSK_stat(task, &status);

// Compute the watchpoint start address
    addr = ((unsigned long)status.attrs.stack
           + (unsigned long)status.attrs.stacksize - STKOV_MARGIN)
           & (~STKOV_RANGEMASK);

// Assign 'addr' as the value of the task environment pointer
// Note: the environment pointer is not pointing to 'addr'.  Rather,
// the value of the environment pointer is set equal to 'addr'.
    TSK_setenv(task, (unsigned int *)addr);
```

```
// Successful Return

} //end of STKOV_createTaskStack()


/*********************************************************************
* Function: STKOV_initTaskStack()                                    *
* Description: Initialization for the DSP/BIOS task switch hook      *
*   function "STKOV_switchTaskStackOvDetect()".  Run this function   *
*   once in main().                                                  *
* DSP: TMS320C28x                                                    *
* Include files: none                                                *
* Function Prototype: unsigned int STKOV_initTaskStackOvDetect(void);*
* Usage: error = STKOV_initTaskStackOvDetect();                      *
* Input Parameters: none                                             *
* Return Value:                                                      *
*    unsigned int error:                                             *
*      0 = no error                                                  *
*      1 = software failed to gain control of the WP                 *
* Notes:                                                             *
*********************************************************************/
unsigned int STKOV_initTaskStack(void)
{
// Enable EALLOW protected register access
    asm(" EALLOW");

// Attempt to gain control of the watchpoint
    *WP_EVT_CNTL = 0x0001;                  // Write 0x0001 to EVT_CNTL to claim ownership
                                            //   of the watchpoint
    asm(" RPT #1 || NOP");                  // Wait at least 3 cycles for the write to occur

// Confirm that the application owns the watchpoint
    if((*WP_EVT_ID & 0xC000) != 0x4000)     // Software failed to gain control of watchpoint
    {
        asm(" EDIS");                       // Disable EALLOW protected register access
        return(1);                          // Return error code
    }

// Proceed to configure the static portion of the watchpoint
    *WP_MASK = (unsigned long)STKOV_RANGEMASK;  // Watchpoint reference address mask
    IER |= 0x8000;                              // Enable RTOSINT

// Successful Return
    asm(" EDIS");                           // Disable EALLOW protected register access
    return(0);                              // Return with no error

} //end of STKOV_initTaskStack()


/*********************************************************************
* Function: STKOV_switchTaskStack()                                  *
* Description: Configures a hardware watchpoint to trigger an        *
*   RTOSINT on write access at the end of a TSK stack.  This         *
*   function is designed to be the task switch hook function in      *
*   DSP/BIOS.                                                        *
* DSP: TMS320C28x                                                    *
* Include files: std.h, tsk.h                                        *
* Function Prototype:                                                *
*   void STKOV_switchTaskStack(TSK_Handle, TSK_Handle);              *
* Usage: STKOV_switchTaskStack(oldtask, newtask);                    *
```

```
* Input Parameters:                                                    *
*   TSK_Handle oldtask = handle to old task.                           *
*   TSK_Handle newtask = handle to new task.                           *
* Return Value: none                                                   *
* Notes:                                                               *
*   1) The function STKOV_initTaskStack() must be run once             *
*      before this function can be run.                                *
*   2) The function STKOV_createTaskStack() must have been used as     *
*      the task create hook function.                                  *
***********************************************************************/
void STKOV_switchTaskStack(TSK_Handle oldtask, TSK_Handle newtask)
{
unsigned long addr;                          // Address to set the WP at

// Retrieve 'addr' from the task environment pointer
    addr = (unsigned long)TSK_getenv(newtask);

// Enable EALLOW protected register access
    asm(" EALLOW");

// Disable the already owned watchpoint
    *WP_EVT_CNTL = 0x0001;

// Proceed to configure the dynamic portion of the watchpoint
    *WP_REF = addr | (unsigned long)STKOV_RANGEMASK;   // Watchpoint reference address
                                                       //   (write all masked bits as 1's)

// Enable the watchpoint
    *WP_EVT_CNTL = EVT_CNTL;

// Successful Return
    asm(" EDIS");                            // Disable EALLOW protected register access

} //end of STKOV_switchTaskStack()

// end of file stkov_taskstack.c
```

## B.3 stkov.h

```
/*****************************************************************
* File: stkov.h                                                  *
* Device: TMS320C28x                                             *
* Author: David M. Alter, Texas Instruments Inc.                 *
* Description: Include file for StackOverflow.c                  *
* History:                                                       *
*   May 1, 2003 - Original (D. Alter)                            *
*****************************************************************/
/*****************************************************************
* THIS PROGRAM IS PROVIDED "AS IS".  TI MAKES NO WARRANTIES OR   *
* REPRESENTATIONS, EITHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING*
* ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A       *
* PARTICULAR PURPOSE, LACK OF VIRUSES, ACCURACY OR COMPLETENESS OF*
* RESPONSES, RESULTS AND LACK OF NEGLIGENCE.  TI DISCLAIMS ANY   *
* WARRANTY OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, AND      *
* NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS*
* WITH REGARD TO THE PROGRAM OR YOUR USE OF THE PROGRAM.         *
*                                                                *
* IN NO EVENT SHALL TI BE LIABLE FOR ANY SPECIAL, INCIDENTAL,    *
* CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY THEORY*
* OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED OF THE     *
* POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF THIS    *
* AGREEMENT, THE PROGRAM, OR YOUR USE OF THE PROGRAM.  EXCLUDED  *
* DAMAGES INCLUDE, BUT ARE NOT LIMITED TO, COST OF REMOVAL OR    *
* REINSTALLATION, COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS*
* OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF *
* BUSINESS.  IN NO EVENT WILL TI'S AGGREGATE LIABILITY UNDER THIS *
* AGREEMENT OR ARISING OUT OF YOUR USE OF THE PROGRAM EXCEED FIVE *
* HUNDRED DOLLARS U.S.$500).                                     *
*                                                                *
* Unless otherwise stated, the Program written and copyrighted by*
* Texas Instruments is distributed as "freeware".  You may, only *
* under TI's copyright in the Program, use and modify the Program*
* without any charge or restriction.  You may distribute to third*
* parties, provided that you transfer a copy of this license to the*
* third party and the third party agrees to these terms by its first*
* use of the Program.  You must reproduce the copyright notice and*
* any other legend of ownership on each copy or partial copy, of the*
* Program.                                                       *
*                                                                *
* You acknowledge and agree that the Program contains copyrighted*
* material, trade secrets and other TI proprietary information and*
* is protected by copyright laws, international copyright treaties,*
* and trade secret laws, as well as other intellectual property  *
* laws.  To protect TI's rights in the Program, you agree not to *
* decompile, reverse engineer, disassemble or otherwise translate*
* any object code versions of the Program to a human-readable form.*
* You agree that in no event will you alter, remove or destroy any*
* copyright notice included in the Program.  TI reserves all rights*
* not specifically granted under this license. Except as         *
* specifically provided herein, nothing in this agreement shall be*
* construed as conferring by implication, estoppel, or otherwise,*
* upon you, any license or other right under any TI patents,     *
* copyrights or trade secrets.                                   *
*                                                                *
* You may not use the Program in non-TI devices.                 *
*****************************************************************/
```

```
#ifndef STKOV_
#define STKOV_

#include <tsk.h>

// C++ Support
#ifdef__cplusplus
    extern "C" {
#endif

// Global Function Prototypes
extern unsigned int STKOV_initSystemStack(unsigned long, unsigned long, unsignedint);
extern void STKOV_createTaskStack(TSK_Handle);
extern unsigned int STKOV_initTaskStack(void);
extern void STKOV_switchTaskStack(TSK_Handle, TSK_Handle);

// Global symbols defined in the linker command file
extern unsigned int HWI_STKBOTTOM;
extern unsigned int HWI_STKTOP;

#ifdef__cplusplus
    }
#endif

#endif  // end of STKOV_ #ifndef

// end of file stkov.h
```

# Appendix C. Troubleshooting Analysis Block Resource Conflicts

Code Composer Studio (including DSP/BIOS) makes use of the emulation analysis block resources for various debugging features. Conflicts can therefore arise when software attempts to gain ownership of a watchpoint for stack overflow monitoring. This section presents the most likely reasons for a resource conflict with Code Composer Studio, and indicates what action is needed to eliminate the conflict.

It is important to understand that each analysis unit resource can be used either by Code Composer Studio or by the stack overflow detection software, but not by both. Therefore, some debugging capabilities will need to be sacrificed if the stack overflow detection software is made operational during debug. The obvious solution to this problem is to enable the stack overflow detection near the end of the software development cycle (after the major debug and development work is completed). One simply needs to comment out the function calls to STKOV_initSystemStack() and STKOV_initTaskStack() from their code. The task switch hook function STKOV_switchTaskStack() and the task create hook function STKOV_createTaskStack() can be left designated as the task hook functions in DSP/BIOS. The application will consume the same execution cycles as before (which is important for application benchmarking and real-time requirements debug), but will have no effect on the watchpoint configuration registers since ownership of the task stack monitoring watchpoint was not secured in the STKOV_initTaskStack() function.

## C.1 Hardware Breakpoints

A hardware breakpoint monitors the program address bus and causes a CPU halt when the address bus matches a configured value. This can be differentiated from a software breakpoint, where a special emulation halt instruction is actually inserted into the code by the debugger in place of the instruction previously located at the specified address. Therefore, hardware breakpoints are used when debugging in non-volatile memory (such as ROM or Flash), since a software breakpoint cannot be used in read-only memory. On the C28x DSP, hardware breakpoints utilize the same two emulation analysis units that watchpoints use.

If memory has been defined as read-only in the Code Composer Studio memory map, and a breakpoint is set on code in that memory, Code Composer Studio will automatically use a hardware breakpoint instead of a software breakpoint. This could cause the stack overflow detection code to fail to gain control of the watchpoint resources. User configured breakpoints can be examined within Code Composer Studio on the *Debug->Breakpoints* menu, *Breakpoints* tab. Figure C-1 shows an example. The breakpoint at address 0x2000 is a software breakpoint, whereas the breakpoint at address 0x1000 is a hardware breakpoint, and is clearly denoted by "H/W Break." If a resource conflict arises, one should disable any hardware breakpoints.

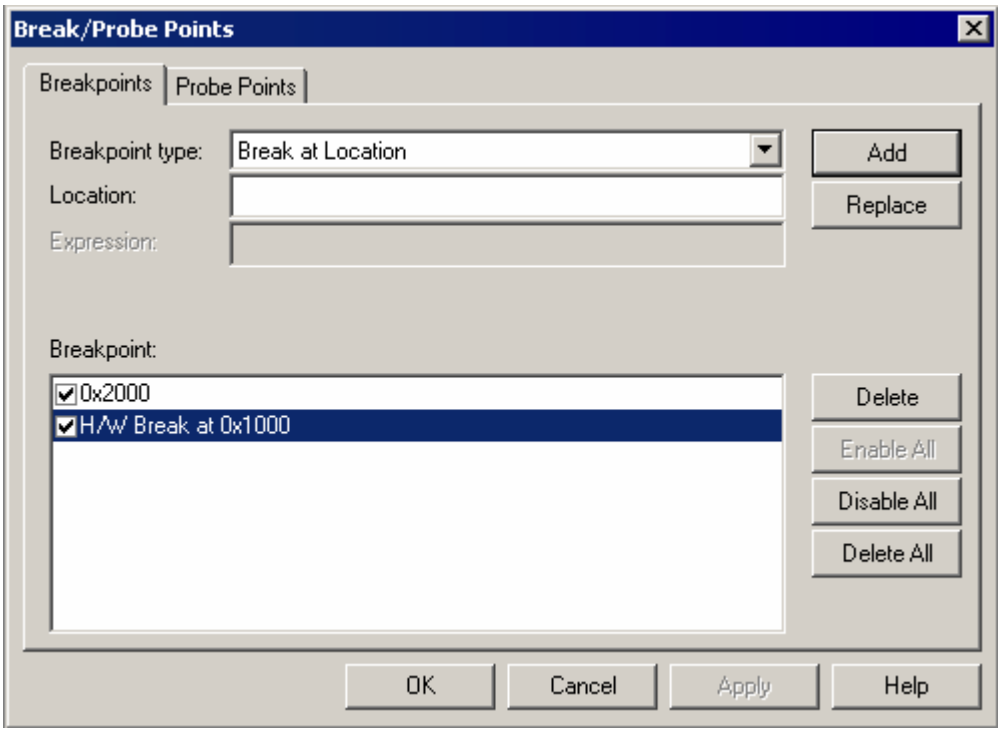


**Figure C-1. Hardware Breakpoints in Code Composer Studio v2.20**

Code Composer Studio automatically sets two breakpoints at program load for a C/C++ program that will not appear on the breakpoint menu: a CIO breakpoint, and an end of program breakpoint. If these locations fall into flash memory, Code Composer Studio will use hardware breakpoints. This could cause the stack overflow detection code to fail to gain control of the watchpoint resources. You can instruct Code Composer Studio to not set these breakpoints by deselecting the appropriate boxes on the *Option->Customize* menu, *Program Load Options* tab, as shown in Figure C-2. More information on these two breakpoints may be found by clicking the Help button on that tab from within Code Composer Studio.

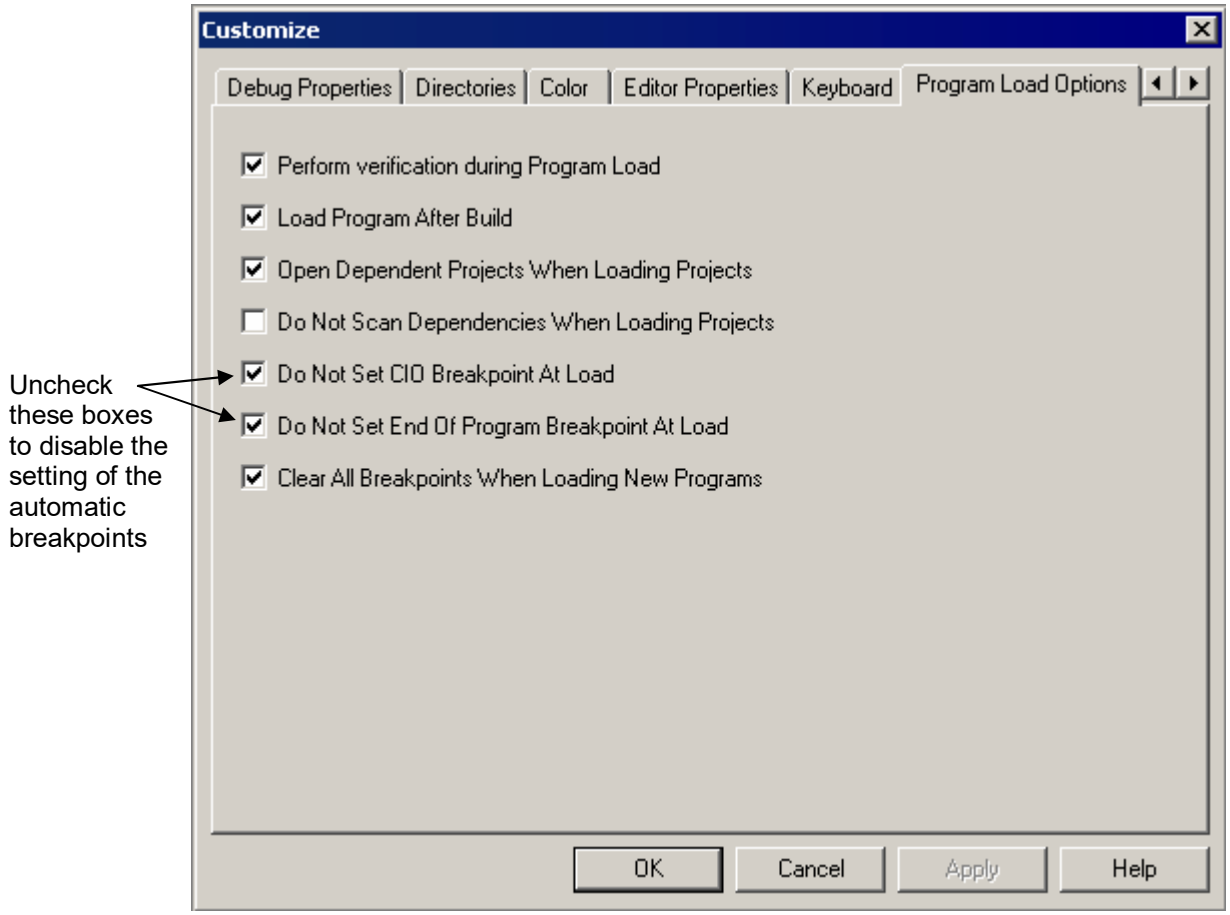


**Figure C-2. Automatic Breakpoint Setting Options in Code Composer Studio v2.20**

## C.2 Real-time Analysis Tools

The real-time analysis (RTA) features of DSP/BIOS use analysis unit #1, which will cause a conflict with stack overflow detection code attempting to use watchpoint #1. To disable the RTA tools and also remove all RTA code from your project, open your project configuration file (i.e., the *.cdb file) inside Code Composer Studio, open the Input/Output properties tree, then right-click the *RTDX - Real-time Data Exchange Settings* and select properties. Uncheck the *Enable Real-time Data Exchange (RTDX)* box, as shown in Figure C-3.

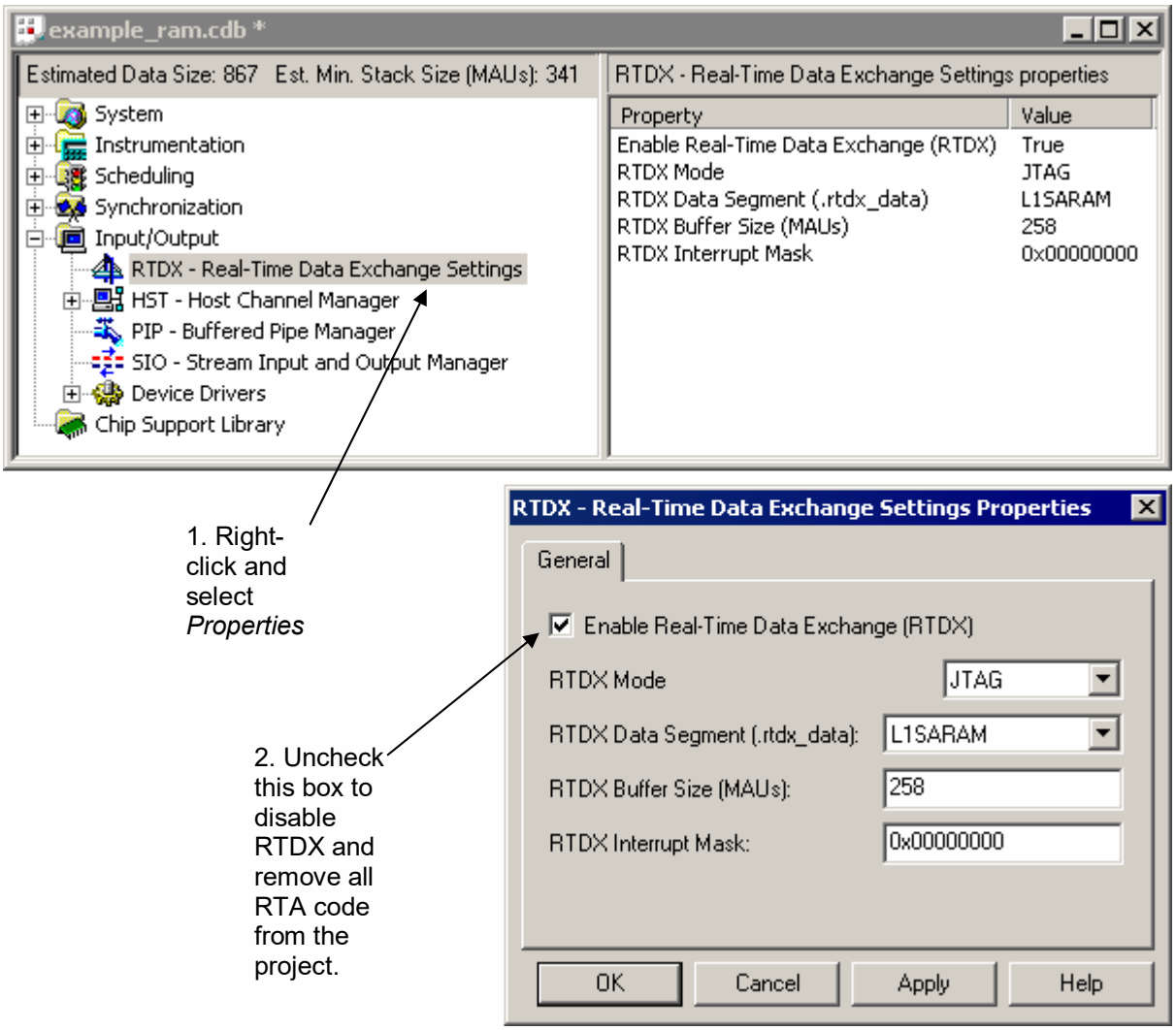


**Figure C-3. DSP/BIOS RTDX Control Window in Code Composer Studio v2.20**

Alternately, one can disable the RTA tools at runtime (but leave the RTDX/RTA code in the project). To do this, open the *DSP/BIOS->RTA_Control_Panel* menu in Code Composer Studio, as shown in Figure C-4, and uncheck the *Global host enable* box.

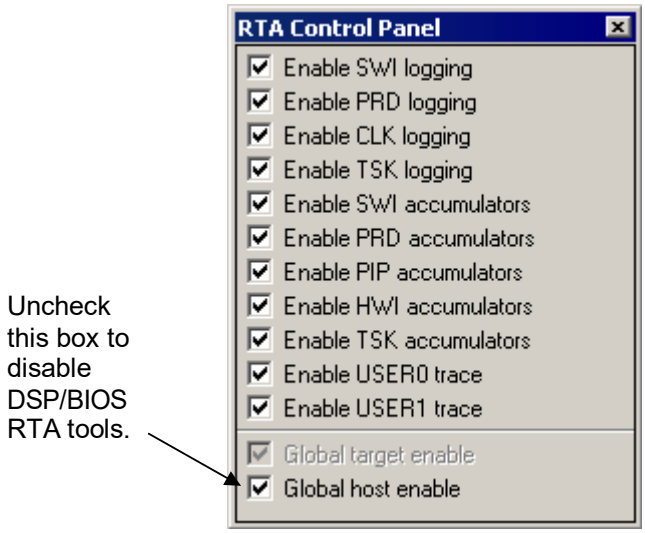


**Figure C-4. RTA Control Panel in Code Composer Studio v2.20**

## C.3 Code Profiler

The Code Composer Studio code profiler uses analysis unit #1, which will cause a conflict with stack overflow detection code attempting to use watchpoint #1. You can disable the profiler clock on the *Profiler* menu within Code Composer Studio.

## C.4 Resetting the Emulator

In some cases, it may be necessary to reset the emulation link after disabling the Code Composer Studio feature causing the resource conflict. This is because Code Composer Studio does not necessarily relinquish control of the analysis unit resource when the offending feature is disabled. To reset the emulation link, select *Debug->Reset_Emulator* from within Code Composer Studio.

**CAUTION:**

**The Code Composer Studio debugger will generally take control of any emulation analysis resource it wants regardless of whether the application currently owns the resource or not. The user should be aware of this when performing debug. The stack overflow code may be working just fine one minute, but then, for example, if the user sets a hardware breakpoint or enables the profiler, Code Composer Studio will take control of the analysis units and not provide any warning.**

## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265