µC/OS-II Real-Time Kernel for CR16C-Based Products

National Semiconductor
Application Note
Jeffrey Wright
September 2003

1.0 Introduction

Previously, I've described the ports of both µC/OS and µC/OS-II for National Semiconductor's CR16B-based CompactRISC™ microcontrollers. Recently, an updated version of the core has been added to the CompactRISC™ family — the CR16C — and µC/OS-II has been ported for its use. The CR16C offers many new and improved features, including:

- Expanded memory addressing (up to 16M bytes)
- Faster multiplication (only 4 cycles)
- Several new instructions — for example, **`MACx`** instructions to support DSP filter loops
- A new protected mode, including separate Supervisor and User stacks. This can result in significant memory savings in multitasking environments.

As with the CR16B, CR16C users will find that µC/OS-II offers a better price/performance fit than most other third party RTOSs. In this application note, I'll describe the µC/OS-II CR16C port, and how it works with the CR16C. I'll also provide some insights into getting your CR16C-µC/OS-II project up and running. For a more thorough and detailed description of µCµOS-II, see Jean Labrosse's book *MicroC/OS-II, The Real-Time Kernel*.

2.0 µC/OS-II Real-Time Operating System

As described in his book, µC/OS-II is a portable, ROMable, scalable, preemptive, real-time, multitasking kernel that can manage up to 63 tasks. µC/OS-II compares favorably in performance with the other RTOSs available for the CR16C. The execution times for every service provided by µC/OS-II (except one) is both deterministic and constant. µC/OS-II allows you to:

- Create and manage up to 56 user tasks (8 additional reserved by OS)
- Create and manage binary, counting, and mutual exclusion semaphores
- Create and manage message mailboxes and queues
- Create and manage event flags
- Delay tasks for integral number of ticks
- Lock and unlock the scheduler
- Change the priority of tasks
- Delete tasks
- Suspend and resume tasks

3.0 What's All This RTOS Stuff Anyway?

There's no doubt that the most obvious question often left unasked for fear of appearing ignorant is simply — *what exactly is an RTOS and how does it work?* What a stupid question!! I've no time to deal with such nonsense now. Just kidding; this is an excellent question. Let's start by defining a few terms.

An **Operating System** or **OS** is a collection of software designed to make the job of executing application programs easier. When you hear the term operating system, you typically envision the Windows or UNIX software that runs your PC or workstation. A typical OS comprises multiple layers, each of which provides application programs with specific services.

The **shell** is the visible portion of the OS that interacts with users and gives the OS its distinctive personality. This layer is most often associated with general-purpose operating systems such as UNIX and Windows. Deriving its name from early UNIX systems, this User Interface (UI) layer is a simple program that interprets the user's commands and executes the requested tasks. Several varieties of shells exist for the various flavors of UNIX. Though varying slightly in their syntax, they all provide the user with essentially the same services. For example under UNIX, the user types the **`ls`** command to list the contents of the current directory. The default shell for DOS was COMMAND.COM, and the user types the **`dir`** command to do the same thing.

Following UNIX's "nut" analogy, the **kernel** lies at the core of an OS and is the most critical piece of software. The kernel is that portion of an OS responsible for managing the orderly execution of *tasks* and regulating the communication between tasks. It is this layer that we're most concerned about when talking about an *embedded* RTOS. The most basic function of the kernel is *context switching*.

A **real-time** process is one in which the actions performed (outputs) in response to the system's inputs proceed in a *deterministic* manner. That is, responses to events must be generated within a prescribed time, and failure to do so may result in unacceptable consequences. Contrast this with non-real-time processes such as the batch processing used on many large computer systems. Users submit jobs, or programs, and the computer runs the job when it has time. Most embedded control applications can be classified as *real-time*.

Further differentiating real-time systems is the concept of *hard* real-time and *soft* real-time systems. Perhaps the best way to explain the difference is to look at an example of a hard real-time system. Consider a system designed to measure the RMS power associated with each of the harmonics present on a power line. If DSP (Digital Signal Processing) techniques are employed (e.g. FFTs), it is mandatory that the power line signal is sampled at consistently uniform intervals. Also mandatory is that the FFT algorithm must run through its complete loop, or "butterfly" (compute the next output), during the interval between successive samples. In this case, therefore, it is essential that the designer

knows with absolute certainty the latencies and delays within the software and have complete control over the timing of the system's outputs. A soft real-time system, on the other hand, can tolerate a bit more uncertainty in its timing relationships. A task may be scheduled to run at prescribed intervals, but there may be slight variations in these intervals that cannot always be predicted. Nonetheless, the use of a soft real-time kernel is usually completely satisfactory for most embedded applications.

A **Real-Time Kernel** or **RTK** will usually support **multitasking**. Multitasking (or multiprogramming) is simply the ability to run two or more independent **tasks** or programs on one CPU at what appears to be the same time. While not actually running concurrently (this would require multiple processors), tasks are executed by the CPU in such a manner as to *appear* to be running at the same time. For example, a user could be entering data on a keyboard, printing the results of a test, and reading an incoming message on a display all at (apparently) the same time.

Another of a kernel's basic responsibilities is **scheduling**. Scheduling is simply deciding which task gets to run. Each task must be given its fair share of CPU time, and it is the kernel's scheduler that determines what is fair. The method employed by the kernel to decide which task gets to run and at what time is its **scheduling policy**. Most real-time kernels are priority-based, meaning the user assigns each task a priority that reflects its relative importance in the application. Tasks with higher priorities will always be given control of the CPU in favor over lower priority tasks. Other kernels follow a time-sharing policy that permits multiple tasks to be equal in priority. Also called "round-robin" scheduling, this scheduling policy gives each task control of the CPU for a specific amount, or slice, of time, and each equal priority task will run in turn. Still other kernels permit both types of scheduling.

In addition to its scheduling policy, real-time kernels are further differentiated by just *when* they give control to a higher priority task. A **preemptive** kernel will always give control to the highest priority task ready to run. If a higher priority task is made ready by some external or internal event (such as an interrupt), the kernel will suspend the current lower priority task and allow the higher priority task to execute.

In contrast, tasks running under a **non-preemptive** kernel must voluntarily relinquish control of the CPU to allow other tasks to execute. Also called **cooperative multitasking**, non-preemptive kernels are not as common in real-time applications because of the inherently non-deterministic nature of their scheduling. That is, although a higher priority task may be ready to run, the currently executing task will continue to run until *it* decides to give control back to the kernel. Consequently, the time that elapses once the higher priority task has been made ready until it actually gets control of the machine is uncertain. To summarize, the *preemptive* kernel ensures that the more important or time-critical tasks are performed first, while the *non-preemptive* kernel relies upon the tasks themselves to cooperate in sharing the CPU.

In a multiple-task application, the kernel's scheduler decides which task should run and at what time. When another task needs to be executed, the kernel will perform a context switch. A **context switch** saves the currently running task's **context** (usually equivalent to the *state* of the machine — i.e. the CPU's registers) on its stack and restores another task's context (which had previously been saved in *its* stack space) to the CPU's registers. In this manner, the new task begins execution from where it left off, while the previous task sits poised for execution sometime in the future.

Notice this is very similar to what happens during the handling of an interrupt, except that each task will have its own stack space — *not* shared by the other tasks. ***Understanding this critical distinction is key***. Remember that a task thinks that it is the *only* program running on the CPU. When it is interrupted or preempted, its context is saved just as though it was interrupted, so that it can proceed where it left off once it resumes control of the CPU. Figure 2 may help illustrate the concept. Notice that each task has its own stack space, and the OS is the gateway to the CPU.

A context switch may occur in a number of situations:

- A task may call one of the kernel's services, which in turn makes a higher priority task ready to run. For example, the task could "post" a semaphore by calling `OSSemPost()`, for which another, higher priority task is waiting.
- A task might delay itself by calling the kernel service `OSTimeDly()`, in which case the next-highest priority task ready to run will be given control of the CPU.
- An interrupt may occur that makes a higher priority task ready to run. For example, the interrupt might deposit a message in an OS "mailbox" (with a call to `OSMbox()`), for which a higher priority task is waiting. Upon exiting the ISR, the kernel will return to the higher priority task instead of the interrupted task.

4.0 OK, Why Should I Use an RTOS?

It's certainly true that many or most applications can be written without the support of an RTOS. If your embedded programming background has not included experience with RTOSs, you might be hesitant to jump right in. Actually, you shouldn't let it trouble you; once you gain a bit of experience with an RTOS you're unlikely to ever want to go back. Here are just a few reasons to consider using an RTOS in your next project:

- The job of writing application software is generally easier using an RTOS, because the use of a kernel enforces certain disciplines in how your code is structured. This helps protect against writing wholly incoherent spaghetti code, often characteristic of programs written without an RTOS.
- On larger projects employing multiple programmers, having consistent APIs for kernel services and Inter-Process Communications (IPCs) further enforce the discipline of structure and modularity — leading to *portability* and *reusability*.
- While the illusion of *concurrency* can be created without the use of an RTOS (though not always), it almost always results in a much more complex piece of software.
- In addition to basic scheduling and context switching, a real-time kernel typically provides other valuable services to applications such as:
  - **Time Delays** — allowing tasks to delay themselves for an integral number of *system ticks*. This allows tasks to be scheduled at periodic intervals or simply return control to the kernel when they have nothing to do.
  - **System Time** — the kernel maintains the *master clock*, usable by tasks that need a consistent timebase.
  - **Inter-Process Communication (IPC)** — tasks may communicate with each other by sharing messages placed in *mailboxes* or *queues* managed by the kernel.
  - **Synchronization** — tasks may synchronize their activities by using kernel-managed *semaphores* or flags. That is, tasks may activate other tasks and coordinate their activities by using kernel resources, instead of relying upon global variables.
  - **Resource Protection** — the kernel can protect system resources (I/O devices, global variables/data structures, etc.) by requiring tasks to acquire exclusive access through a *mutex* or Mutual Exclusion Semaphore.

5.0 CR16C and Real-Time Operating Systems

The CR16C includes features specifically tailored for use with RTOSs. For example:

- The CR16C includes a new *protected* mode of operation called the **User** mode. While operating in User mode, tasks utilize their own unique program stacks, while all exceptions (interrupts and traps) use the **Supervisor** stack. This scheme helps to reduce RAM requirements because task stacks do not need to include space for exceptions or potential interrupt nesting. Rather, each task's stack only needs to be sized to meet task-specific demands, leaving all other kernel and interrupt-related stack issues to the OS itself. Furthermore, while operating in User mode, tasks are prohibited from modifying the CPU's core registers, providing an additional measure of protection from ill-behaved tasks.
- In addition to the User and Supervisor stacks, the CR16C retains the legacy **Interrupt** stack, on which the current Program Counter (PC) and Program Status Register (PSR) are saved during an exception. While this separate interrupt stack does complicate the kernel's task-switching process, it also reduces memory requirements in a multitasking system running in the default Supervisor mode.

6.0 CR16C and Embedded Applications

In addition to its support for RTOSs, the CR16C was specifically designed for use in embedded environments. That is, despite technically being a **RISC** machine with an instruction set architecture tailored to support High-Level Languages — it's a General Purpose Register (GPR) machine with an orthogonal (nearly) instruction set — it also includes features normally found only in CISC machines. These features include:

- **PUSH and POP Instructions** — allow up to 8 registers (9 if the return address register RA is included) to be pushed to/popped from the stack at a time. Interrupt latency is a critically important metric in real-time systems. Minimizing this, as well as other latencies, is a deliberate design goal for embedded architectures. The greater the time that interrupts are disabled, the less sensitive the system is to real-time events. The CR16C's PUSH and POP instructions help minimize this *numb time*, because on ISR entry the scratch registers (R0..R6) and the return address register (RA) can be saved on the stack in just one PUSH instruction (PUSH$7, R0, (RA)) consuming only 10 clock cycles. Should a context switch be indicated, the remaining safe registers (R7..R13) can be saved in only 11 additional cycles.
- **Atomic (Non-Interruptible) Bit Operations** — bit (Boolean) operations are fundamental to embedded applications. Testing or changing the state of a bit in memory can result in mysterious behavior if the process can be interrupted. For example, on a RISC machine, a bit-set operation would proceed as:

```
LOADW     flags, R0
ORW       $BIT2SET, R0
STORW     R0, flags
```

  If this sequence were interrupted prior to the STORW instruction, the Interrupt Service Routine (ISR) (or another task made ready by the ISR) *could* alter the same bit. This would normally be undesirable, and could be the cause of unpredictable or even catastrophic behavior. To

avoid these unpleasant consequences, the CR16C provides non-interruptible or *atomic* bit operations through its TBIT, SBIT, and CBIT instructions.

- **Fast MUL/MAC Instructions** — a fast multiply instruction can be extremely useful in a wide range of embedded applications. On many RISC machines, the lack of a multiply instruction means that all but the simplest multiply operations (x2, x4, etc.) must be emulated in software, which makes them unacceptably slow. The CR16C can multiply two 16-bit signed integers to generate a 32-bit product in just 4 cycles. The CR16C also includes powerful new MAC (Multiply-Accumulate) instructions, capable of multiplying two 16-bit integers and adding the product to a 32-bit sum. It also supports Q15 (16-bit signed fractions) format.

7.0 CR16C and µC/OS-II

Unlike the other RTOSs available for the CR16C, this µC/OS-II port supports *both* User and Supervisor operating modes, as well as *interrupt nesting*. Each of these options (and a few others) are defined by the user at build time, which provides the user with a great deal of flexibility. Figure 1 through Figure 3 will help resonate some understanding of the fundamental principles behind µC/OS-II. Figure 1 shows how µµC/OS-II initializes a task's context on its program stack. When creating a task, µC/OS-II initializes the task's stack frame with certain required parameters to ensure correct operation once the task gains control of the CPU. For example, every task begins with interrupts enabled (both locally and globally), and its stack frame is defined as type *Initial*. (The frame type is a new field added with the CR16C port to ensure that the most efficient macros are used within the kernel's context switch routines.) In addition, the initialization routine allows the user to pass an argument to the task for application-specific purposes. Note that the stacking order differs between the two modes. This is done to achieve maximum efficiency during context switching, but is invisible to the application.

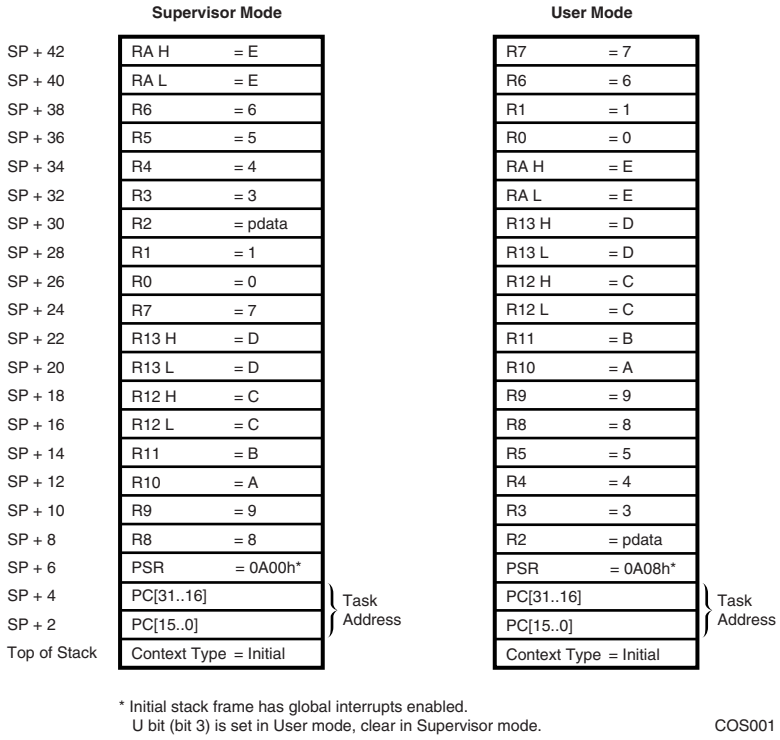

Figure 1. Task's Stack Frame as Initialized by `OSTaskCreate()`

Figure 2 shows how µC/OS-II manages the tasks running under it. Note that each task has its own unique environment, comprising its own stack space and register set. Only *one* of these tasks may control the CPU's registers at any one time, and it is the kernel's scheduler that decides which task has that control. The kernel maintains a Task Control Block (TCB) for each task. These TCBs are singly linked and hold task-specific data used by the kernel to manage each task.

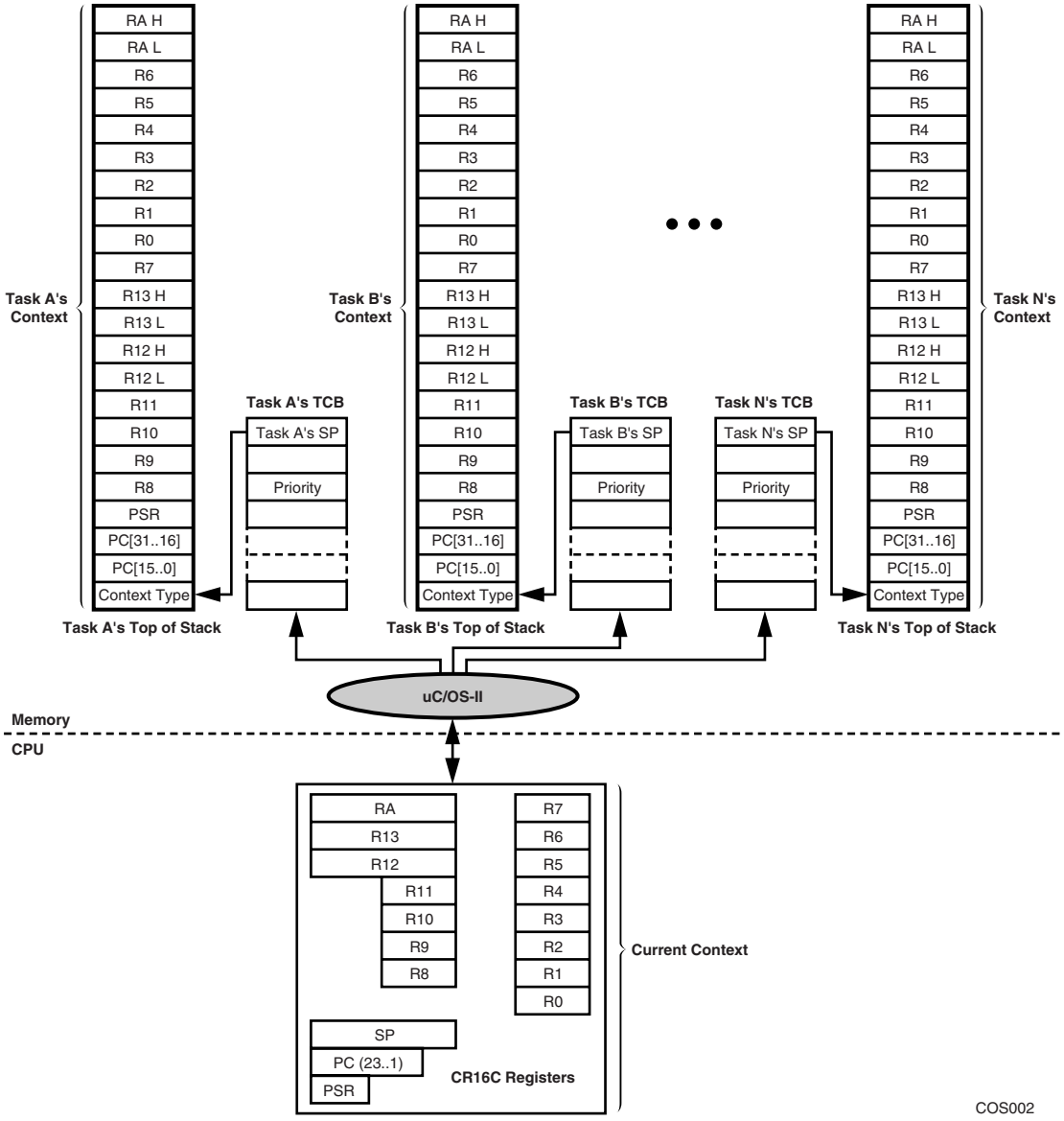

Figure 2. Multiple Tasks Managed by µC/OS-II (Supervisor mode)

Figure 3 shows the sequence of events that occur during a context switch, in which a lower priority task is preempted by a higher one. In the case shown, Task A is the current owner of the CPU and has called the OS's service **OSTimeDly()** (this sequence would be identical for *any* OS service call that resulted in a context switch). Upon completion of the **OSTimeDly()** service, the OS calls its scheduler to determine whether to return CPU control to the calling task (Task A) or to *preempt* this task and give control of the CPU to another task that is higher in priority and ready to run. In this case, the scheduler has determined that Task B should run next, so it invokes the kernel's context switch routine via the **OS_TASK_SW()** macro. This port assigns the **SVC** trap to µC/OS-II for this purpose. The context switch then proceeds as follows:

1. The **_PUSHSAFE_** (Supervisor mode) or **_SAVESAFE_** (User mode) macro saves all of the CPU's *safe* registers on Task A's program stack (there is no need to save the *scratch* registers because, by convention, the caller will have already saved those it requires).
2. The Program Status Register (PSR) which was saved on the Interrupt stack during the trap is retrieved and placed on Task A's program stack.
3. The Program Counter (PC) is also retrieved from the Interrupt stack, incremented (because, unlike an interrupt, a trap does not increment the PC), and placed on Task A's program stack.
4. Task A's context is defined as type Task (indicating that the kernel only needs to do a partial restore when the task next runs).
5. The CPU's SP register is saved in Task A's Task Control Block (TCB).
6. Task B's stack pointer is copied from its TCB into the CPU's SP register.
7. Task B's context type is retrieved from the top of its stack.
8. Task B's PC is read from its stack and written to the Interrupt stack.
9. Task B's PSR is read from its stack and written to the Interrupt stack.
10. As indicated by Task B's context type (which in this case happens to be type Task), a partial (Supervisor mode) or complete (User mode) restore is performed using the macro **_POPSAFE_** (Supervisor mode) or **_RESTOREALL_** (User mode).
11. Finally, the kernel returns to Task B through a **RETX** instruction.

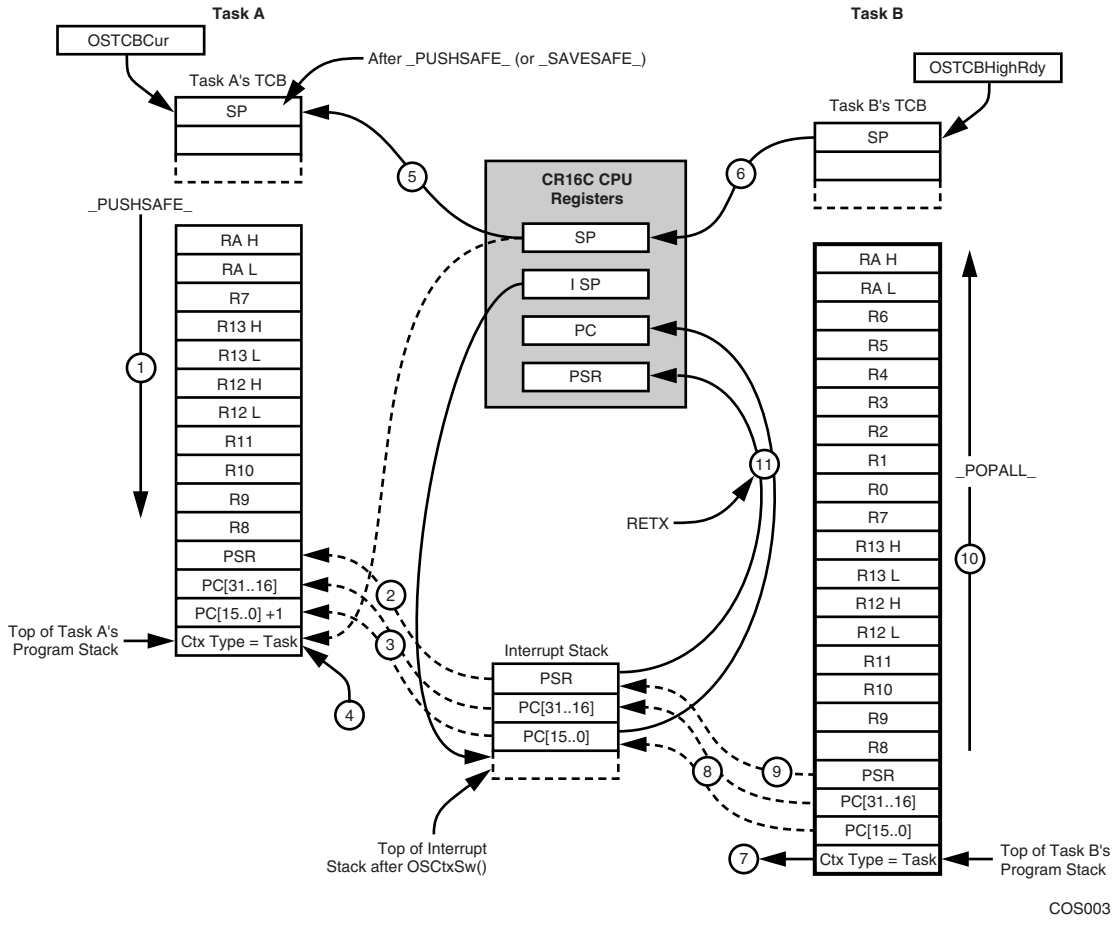

Figure 3. Context Switch (task level)

Again, why must the PC be incremented in step #3 above? In contrast with interrupts, certain traps on the CR16C *suspend* the current instruction. That is, interrupts are only acknowledged following the *completion* of the current instruction (except for interrupts occurring during **MULx** instructions), so the saved Program Counter will point to the instruction *following* the instruction during which the interrupt occurred. On the other hand, most traps (with the "exception" of the **TRC** and **DBG** traps) suspend the current instruction and save the *address of the trap instruction itself on the Interrupt stack*. Because µC/OS-II invokes its task-level context switch routine through the **SVC** trap, the value of the PC saved on the interrupt stack must be incremented to point to the *following* instruction. Otherwise, we would be "caught in a trap", as it were. Refer to Appendix A for the task-level, context switch routine.

8.0 Interrupt Service Routines (ISRs)

Porting an OS requires a detailed understanding of how a compiler handles ISRs. Upon entry to any interrupt handler, the compiler must generally save *only* those registers it needs to use within the handler, pushing them onto the current stack. In Supervisor mode, this is the interrupted task's stack, while in User mode this is the system stack. However, if the ISR makes a call to a function, the compiler must also save any/all of the remaining scratch registers (R0..R6), as well as the return address register RA. This must be done because, by convention, any function may freely clobber any of these non-safe or *scratch* registers (a function is only responsible for saving the safe registers it uses; i.e. R7..R13). Moreover, at the extreme, if an ISR is lengthy or written with too much complexity, the compiler may also be forced to save and utilize one (or more) of the safe registers as well. Consider for example the subtleties of the following, deceptively simple ISR:

```
void BadISR (void)
{
     UWORD temp;

     temp = INTERRUPT_REGISTER;   // Read (and clear) interrupt pending flag

     if (temp & THIS_FLAG)        // THIS_FLAG set?

          DoThis();               // YES - DoThis stuff...

     if (temp & THAT_FLAG)        // THAT_FLAG set?

          DoThat();               // YES - DoThat stuff...
}
```

Notice the local variable **temp** — it is used *across* a function call (in this scenario, the pending bit is cleared automatically just by reading the register). Here's what the compiler spits out:

```
\          BadISR:
     239   {
\          000000 F001          PUSH $8, R0, (RA)
     240             UWORD temp;
     241
     242             temp = INTERRUPT_REGISTER;
\          000002 7F8988FF      LOADW 0xffff88, R7
     243
     244             if (temp & THIS_FLAG)
\          000006 0706          TBIT $0, R7
\          000008 9310          BFC `??BadISR_0`:S
     245
     246                  DoThis();
\          00000A ........      BAL (RA), DoThis
     247
     248             if (temp & THAT_FLAG)
\                         `??BadISR_0`:
\          00000E 1706          TBIT $1, R7
\          000010 9310          BFC `??BadISR_1`:S
     249
     250                  DoThat();
\          000012 ........      BAL (RA), DoThat
     251   }
\                         `??BadISR_1`:
\          000016 F002          POP $8, R0, (RA)
\          000018 0300          RETX
     252
```

Do you see why the compiler had to use one of the available safe registers (R7) in this example? Again, this is a consequence of the fact that a called function may freely **destroy** any *scratch* register (R0..R6), so the local variable must be placed in a safe register.

Operating in User mode further complicates matters, because in User mode the registers are saved on the Supervisor stack — not on the interrupted task's stack. If the ISR uses any OS service (i.e. it is an "OS-aware" ISR), this requires that they be saved again on the interrupted task's stack.

Given these and other uncertainties, a decision was made to abandon the method implemented for the CR16B port and always return to an interrupted task through the ISR itself. This is simply due to the fact that while in 99% of the cases the compiler will stack registers in a uniform and consistent manner on entry to an ISR, the compiler will occasionally stray from this uniformity and demonstrate its capricious and unpredictable nature — as evidenced by crashed programs! This approach, while slightly slower, ensures a consistently correct context-switch mechanism, even when the compiler is making unusual optimizations in register usage.

Consequently, the previous restrictions placed upon the user when writing OS-aware ISRs (those that call any OS service) are significantly reduced. However, OS-aware ISRs must *still* be formatted in a manner consistent with Listing 1 below. (See Appendix C for the **`OSSaveContext()`** listing, which is the routine defined for **`SAVE_CONTEXT()`**).

9.0 µC/OS-II and Interrupt Service Routines (ISRs)

Under µC/OS-II, you may write all of your ISRs in C. A context switch occurring from within an ISR proceeds in only a slightly different manner from the task level. When a task is interrupted by a properly formatted, OS-aware interrupt, code inserted by the compiler saves the scratch registers (R0..R6) and the return address register (RA) on the stack. Before executing any ISR instructions, the user must call **`SAVE_CONTEXT()`**. This routine is used in both modes of operation and hides certain mode-related details from the user. For example, the routine increments the kernel's nesting counter and saves the balance of the task's context to its stack (but not if this is a nested interrupt!). If the kernel has been built to support interrupt nesting, the routine also checks the current nesting level to determine whether it needs to save the context at all. This is important because a task's context only needs to be saved once (because no context switch may occur from within a nested interrupt). The placement of this call at the start of your OS-aware ISR is ***mandatory*** to ensure that the interrupted task's context is properly saved in the event that the ISR results in a task switch.

Upon completion of the interrupt code, the ISR must call µC/OS-II's **`OSIntExit()`** routine. This OS function first decrements the interrupt nesting counter to determine whether a task switch is possible. If all interrupts have completed (**`OSIntNesting == 0`**), **`OSIntExit()`** calls the kernel's scheduler which determines whether to return to the interrupted task (though the ISR), or to a higher priority task made ready to run (presumably by the ISR itself). If a context switch is indicated, the kernel calls the port-specific **`OSIntCtxSw()`** function, which is a slightly modified version of the task-based context switch routine **`OSCtxSw()`**. This *interrupt-level* context switch routine is listed in Appendix B. Note that the PC is not incremented, because this routine is invoked through a *call*, not a trap.

```
void UserISR (void)
{
     UBYTE i;

     SAVE_CONTEXT()            // This OS macro saves user's context (mode dependent)

     INTREG |= INT_CLEAR;      // Clear interrupt

     ISRCode…                  // Do ISR stuff here…

     OSServiceXY();            // An OS service is called (e.g. OSSemPost, etc.)

     OSIntExit();              // This is the OS's ISR exit function

     RESTORE_CONTEXT()         // Restore interrupted task's context
}
```

Listing 1. Example of OS-Aware ISR Under µC/OS-II

In addition, if your application requires interrupt nesting, you must include the conditional code shown in Listing 2. In Supervisor mode, the **USE_SYSTEM_STK()** macro swaps out the current stack pointer and replaces it with a global, *system* stack pointer. This must be done before re-enabling interrupts, so that any enabled interrupt occurring during the ISR will use this system stack, not the task's stack. Otherwise, each and every task in your application would have to include additional space on its stack to accommodate each potential level of interrupt nesting. As you can imagine, this would greatly increase RAM requirements. Note that this macro is not used in User mode, because *all* exceptions use the system stack.

```
void UserISR (void)
{
     UBYTE i;

     SAVE_CONTEXT()           // This OS macro saves user's context (mode dependent)

     INTREG |= INT_CLEAR;     // Clear interrupt

#ifdef OS_NEST_EN             // If nesting is desired, do this ...
     #ifdef USER_MODE
     #else

     USE_SYSTEM_STK()         // In Supervisor mode, switch to System stack
                              // (used by nested interrupts)
     #endif

     ENABLE_INTERRUPTS();     // Re-enable Global interrupts

#endif

     ISRCode…                 // Do ISR stuff here…

     OSServiceXY();           // An OS service is called (e.g. OSSemPost, etc.)

#ifdef OS_NEST_EN

     OS_ENTER_CRITICAL();     // If nesting was enabled, disable before call to OS
     DISABLE_INTERRUPTS();

     #ifdef USER_MODE
     #else
     USE_TASK_STK()           // Supervisor mode: switch back to Task's stack
     #endif
     OS_EXIT_CRITICAL();
#endif
                              // This is the OS's ISR exit function
     OSIntExit();

     RESTORE_CONTEXT()        // Restore interrupted task's context
}
```

Listing 2. Example of OS-Aware ISR Under µC/OS-II Using Nested Interrupts

10.0 How Do I Write My Application Under µC/OS-II?

Writing an application to run under µC/OS-II should proceed as follows:

10.1 Define Your Tasks (Processes)

Tasks running under a multitasking kernel should be written in one of two ways:

A non-returning forever loop. For example:

```
void Task (void *)
{
     DoInitStuff();

     while (1){               // this is a long, long time...
          do this;
          do that;
          do the other thing;
          call OS service (); // e.g. OSTimeDelay, OSSemPend, etc.
     }
}
```

A task that deletes itself after running. For example:

```
void Task (void *)
{
     do this;
     do that;
     do the other thing;
     call OS service ();      // e.g. OSTimeDelay, OSSemPend, etc.
     OSTaskDelete();          // Ask the OS to delete the task
}
```

As a simple example, consider the **ADCTask** in Listing 3. While it is running (has control of the CPU), it is either:

- Reading, formatting, and displaying the results of the last A/D conversion
- Delaying itself by calling the OS service **OSTimeDly()**
- Interrupted by the system tick (or another interrupt) and perhaps preempted

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; NAME:   ADCTask        Reads the ADC 10 times per second and displays the averaged
;                        results. Written to run under µC/OS-II.
;
; PARAMETERS:       data      Optional parameter passed to task upon creation.
;
; CALLS:            ADCInit, OSTimeDly, ReadADC, DisplayPuts, Int2Str
;
; RETURNED:         Does not return.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
void ADCTask (void *data)
{
     TPB_T          *tpb;     // Allocate pointer to Task Parameter Block
     RESULT_T       *res;     // Allocate pointer to Results member of TPB

     tpb = &AdcTPB;           // Assign pointer to the ADC TPB
     res = tpb->Results;      // Assign pointer to Results structure

     ADCInit (0x00, 0x25);    // Initialize the ADC

     while(1){                // Run forever

          OSTimeDly (ONESEC/10); // Take readings every 100 ms

          res->Output[0] = ADC_Value;
                              // Save value in results structure
          DisplayPuts(0, 10, Int2Str ((UWORD)ReadADC(), ADC_Value));
                              // Display converted results
     }
}
```

Listing 3. Example of Task Written Under µC/OS-II

10.2 Allocate a System Timer to Provide the OS Tick, and Write an ISR That Calls `OSTickISR()`

All RTOSs require a periodic system interrupt or tick to perform its services. For example, user tasks may delay themselves by an integral number of OS system ticks by calling the OS service `OSTimeDly()`. Additionally, many µC/OS-II services provide a timeout facility to prevent tasks from waiting forever for an unavailable service. And finally, during every OS tick ISR, the kernel calls its scheduler to determine whether a higher priority task is ready to run. The system tick rate is application driven, but would most generally lie in the 5 ms to 100 ms range. Increasing the frequency of the system tick will increase OS overhead and reduce the available task time. Typically, a Worst Case Execution Time (WCET) analysis should be done on your application to determine the optimum system tick rate.

In the previous port, the user was responsible for writing the OS's tick handler. In the CR16C port, the tick handler has been integrated with the kernel. However, the user must still allocate a system timer and configure it for the desired interrupt rate. Additionally, you must write the ISR for this timer and format it as shown in Listing 4. The choice of timer is completely application driven. If available, I suggest that you use the Idle Timer (T0) found on many of the CR16-based microcontrollers. Generally, the Idle timer interrupt will be higher in priority than most or all other peripherals in a device. If an Idle timer is not available, you can use any other available timer, such as the MFT (Multi-Function Timer) or VTU (Versatile Timer Unit). Try to choose a timer with the highest possible interrupt priority. Whichever timer you use, you must model its ISR on the example shown in Listing 4.

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; NAME:              OS_TickISR uC/OS-II time tick interrupt. Since this ISR uses an
;                    OS service, it may cause another (higher priority) task to be made
;                    ready. Therefore, we must use the OS-defined macros SAVE_CONTEXT
;                    and RESTORE_CONTEXT prior to doing anything else. If we allow
;                    nested interrupts, we must also use the macro ENABLE_INTERRUPTS.
;
; PARAMETERS:        N/A
;
; CALLS:             SAVE_CONTEXT, OSTickISRHandler, OSIntExit, RESTORE_CONTEXT
;
; RETURNED:          N/A
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

#ifdef __ICCCR16C__
__interrupt void OS_TickISR (void)
#else
#pragma interrupt (OS_TickISR)
void OS_TickISR (void)
#endif
{

#if OS_TIMER == TWM
// UBYTE i;
#endif

     SAVE_CONTEXT()            /* You must use this macro to save user context. It */
                               /* the details of incrementing the nesting counter. */
#if OS_TIMER == TWM
// i = T0CSR;                  /* Clear interrupt */
     T0CSR;                    /* Clear interrupt */

#elif OS_TIMER == MFTX
     T1ICLR = 0x0f;            /* Clear interrupt */
```

```
#elif OS_TIMER == VTUA
     INTPND = 0x0003;

#elif OS_TIMER == VTUB
     INTPND = 0x0030;

#elif OS_TIMER == VTUC
     INTPND = 0x0300;

#elif OS_TIMER == VTUD
     INTPND = 0x3000;
#endif

     OSTickISR();              /* This is the call to the default OS tick handler */
     OSIntExit();              /* This is the OS's ISR exit function */
     RESTORE_CONTEXT()         /* This OS macro restores interrupted task's context */
}
```

Listing 4. Example of an OS TImer ISR

10.3 Make Certain Your Application Doesn't Use the CR16's SVC Trap!

The SVC trap is assigned to µC/OS-II, for task-level context switching:

```
#define OS_TASK_SW()_excp_(5)
```

10.4 Insert Vectors for µC/OS-II and the OS Tick ISR in the Interrupt Dispatch Table

For example, see Listing 5 below, where I'm using the vector assigned to the Idle timer interrupt (`IRQ31`).

```
#ifdef __ICCCR16C__
                                /* IAR */
#pragma constseg=DISPATCH_TBL
const DISPATCH_TBL_T _dispatch_table = {

#else                           /* NATIONAL */

void (*const _FAR _dispatch_table[])() = {

#endif

     itrap0,                    /* NULL pointer */
     itrap1,                    /* Vector 1 Non Maskable */
     itrap2,
     itrap3,
     itrap4,
     OSCtxSw,                   /* OS Supervisor Call */
     itrap6,                    /* Divide by zero trap */
     itrap7,
     itrap8,
     itrap9,
     itrap10,                   /* Undefined Opcode trap */
     itrap11,
     itrap12,                   /* Illegal Address trap */
     itrap13,
     itrap14,
     itrap15,

/*-------------------------------------*/

     irq0,                      /* IRQ0 = vector #16 RESERVED */
     irq0,                      /* FLASH = vector #17 */
     irq0,                      /* MIWU3 */
     irq0,                      /* MIWU2 */
     irq0,                      /* MIWU1 */
     irq0,                      /* MIWU0 */
//   irq0,                      /* /CTS */
```

```
    UARTFlowISR,            /* /CTS */
    UARTTxISR,              /* UTx = IRQ7 */
    irq0,                   /* MWIRE */

    VTUDisr,                /* VTUD = IRQ9 */
    VTUCisr,                /* VTUC = IRQ10 */
    VTUBisr,                /* VTUB = IRQ11 */
    VTUAisr,                /* VTUA = IRQ12 */

#if OS_TIMER == MFTX
    OS_TickISR,             /* IRQ13 Vector 29 */
    OS_TickISR,             /* IRQ14 */
#else
    MFT1isr2,
    MFT1isr1,
#endif

    AcbIsr,                 /* Access.Bus == IRQ15 */
    CvsdISR,                /* IRQ16 = PCM/CVSD */
    UARTRxISR,              /* IRQ17 */
    CodecISR,               /* AAI */
    CAN0ISR,                /* IRQ19 */
    irq0,                   /* IRQ20 = DMA3 */
    irq0,                   /* DMA2 */
    UARTDMATxISR,           /* DMA1 (UART Tx) */
    UARTDMARxISR,           /* DMA0 (UART Rx) */
    irq0,                   /* USB */
    irq0,                   /* BTooth LLC5 */
    irq0,                   /* BTooth LLC4 */
    irq0,                   /* BTooth LLC3 */
    irq0,                   /* BTooth LLC2 */
    irq0,                   /* BTooth LLC1 */
#if RTX_KERNEL > 0
    bt_llc0_isr,            /* BTooth LLC0 */
#else
    irq0,                   /* IRQ30 */
#endif

#if OS_TIMER == TWM
    OS_TickISR              /* IRQ31 Vector 47 */
#else
    irq0                    /* IRQ31 */
#endif

};
```

Listing 5. Adding the SVC Trap and OSTickISR Vectors to the Interrupt Dispatch Table

10.5 Create All Your Semaphores, Event Flags, Mailboxes, Queues, etc.

Your **main()** routine is usually where basic hardware initializations should occur. It is also where you should initialize the OS. An example of a **main()** routine is shown in Listing 6.

```
/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; FILENAME:          main.c
;
; PURPOSE:           LMX5100 Test Program. Designed for the CR16C-based LMX5100.
;                    Runs under uC/OS-II.
;
; AUTHOR(S):         Jeffrey Wright
;
; REVISION HISTORY: Initial - 8/14/2001
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/
#define MAIN_GLOBALS

#include "includes.h"

     /*****************************
          Starting point MAIN
     *****************************/

void main (void)
{

     OS_ENTER_CRITICAL();      /* Block interrupts (just to be sure) */

     IENAM0 = 0;               /* Ensure all interrupt sources are disabled */
     IENAM1 = 0;

     PRSFC &= 0xf0;            /* Set HF clock prescaler for 12 MHz */
     PRSSC = SLCLK_PRE;        /* Set prescaler for 50 kHz slow clock */

#ifdef EXTOSC                  /* If using external 32 kHz, select it for slow clock */
     CRCTRL |= BIT0;
     PMCSR |= BIT0;
#endif

     BCFG = 0x04;              /* Late Write (2 cycles), core bus observability */
     SZCFG0 = 0x0880;          /* No holds, no waits, not a single luxury */
     SZCFG1 = 0x0880;          /* Ditto… */
     IOCFG = 0x0680;

     MainFlags = 0;            /* Various task/error flags */

     OSInit();                 /* Initialize operating system */
                               /* Create necessary OS semaphores */
```

```
    KeySem = OSSemCreate(1);
                              /* Create the Shell task */
    OSTaskCreate   (BShell,

#ifdef USECOM1                /* Compile to use either com1 or com2 */
                   (void*)1,
#else
                   (void*)2,
#endif
                   (void *)&ShellStk[SHELL_STK_SIZE], SHELL_PRIORITY);

    OS_EXIT_CRITICAL();
    ENABLE_INTERRUPTS();

    OSTimerInit();            /* Initialize the OS time-base */
    OSStart();                /* Start multitasking */
}
```

Listing 6. Initializing µC/OS-II in a `main()` Function

10.6 Allocate Stack Space for Your Tasks

As mentioned earlier, the CR16C has three distinct stack spaces:

- A **User** stack located at the USP register. Actually, while running, it is the SP (R15) that points to the current Top of Stack (TOS). The USP is copied to the SP during the switch to User mode. This stack space is used by the compiler to:
  - Allocate any required local variables not in registers
  - Pass arguments incapable of being passed in registers R2, R3, R4, and R5
  - Save any scratch registers before calling a subroutine, or save any used safe registers after being called
- An **Interrupt** stack located at the ISP core register. This stack is used by the interrupt hardware to save the Program Counter (PC) and Program Status Register (PSR) in response to an exception (interrupt or trap). Because the interrupt stack is only used to hold these two registers, unless you enable nested interrupts, it only needs to be three words deep. However, if you do permit interrupts to be nested, you must increase this stack depth by three words for each potential additional nesting level.
- A **System** stack located at the SP (again, usually R15) register. This stack space is used during all exceptions to save any required registers. Following reset, this is the default stack pointer.

The necessary stack space for a task is allocated by defining an array of type **UWORD**. For example:

```
#define TASK_STK_SIZE 66
     // Size of task's user stack in WORDs
UWORD TaskStack[TASK_STK_SIZE];
     // Task's user stack
```

The size of each of your task's stacks should be based on:

- The 22 words (44 bytes) needed for its context
- Any additional requirements imposed by function arguments, local variables, and worst-case function nesting
- Any additional, unnecessary saving of registers done by the compiler on entry to the task. National's compiler is more intelligent than IAR's in that it recognizes that if a function never returns, none of its registers need to be saved

No consideration needs to be given for nested interrupts, because they use a separate stack.

The total amount of RAM required by an application running under µC/OS-II may be roughly estimated by the following equation:

RAM Required: 200
+ (1 + OS_NUM_TASKS) × 44)
+ OS_MAX_EVENTS × 8
+ OS_MAX_QS × 12
+ SUM (each task's User stack size)
+ SUM (each message queue's size)
+ OS_IDLE_TASK_STK_SIZE

Obviously, you should carefully examine your application's stack usage during development to arrive at final numbers.

10.7 Allocate Stack Space for OSIdleTask()

The minimum stack size will be 24 words (48 bytes), but may increase if you include an idle task hook routine in your application. I usually start out with a stack size of 40 words and reduce this number during development as indicated.

```
#define OS_IDLE_TASK_STK_SIZE 40  // Idle task stack size (WORDs)
#define OS_IDLE_TASK_STK_TOP 40   // idle user task top of stack
```

10.8 Allocate Stack Space for the System Stack

As stated above, the System stack is used during all exceptions while in User mode and for all nested interrupts while in Supervisor mode. You must ensure that sufficient space is provided to accommodate all system needs, including interrupt nesting. If you are using the NSC tool chain, this space is allocated in the output SECTIONS area of the linker definition file used by all projects. For example:

```
SECTIONS  {
          .SysStk (NOLOAD) ALIGN(2) INTO(data_mem): {. += 100;}
          .IntStk (NOLOAD) ALIGN(2) INTO(data_mem): {. += 24;}
          }

__SYSSTK_START = ADDR(.SysStk) + SIZEOF(.SysStk);
__INTSTK_START = ADDR(.IntStk) + SIZEOF(.IntStk);

_USER_CODE_START = 0x0000;
```

If you are using the IAR tool chain, this space is allocated using the following commands within the extended linker command file:

```
-D_SYSSTK_SIZE=240            /* USED BY SUPERVISOR */
-D_INTSTK_SIZE=12             /* USED BY SUPERVISOR */
-Z(DATA)ISTACK+_INTSTK_SIZE,CSTACK+_SYSSTK_SIZE#EC000-EE800
```

10.9 Define OS Parameter OS_MAX_TASKS

This is the total number of tasks you need to create or want the OS to manage. This and all other application-specific parameters are located in OS_CFG.H.

```
#define OS_MAX_TASKS 11       // Number of tasks in your application
```

10.10 Define OS Parameters OS_MAX_EVENTS, OS_MAX_QS, OS_MAX_FLAGS, and OS_MAX_MEM_PART

These numbers are obviously application driven, but must *always* be greater than zero. An Event Control Block (ECB) is required for each semaphore, event flag, queue, and mailbox you create.

```
#define OS_MAX_EVENTS    2    // Max. number of event control blocks in your
                              // application
#define OS_MAX_QS        1    // Max. num queue control blocks in your application
#define OS_MAX_FLAGS     1    // Max. number of Event Flag Groups in your application
#define OS_MAX_MEM_PART  1    // Max. number of memory partitions
```

10.11 Further Configure the OS for the Desired Services

```
#define OS_TASK_STAT_EN       0    // Enable (1) or Disable(0) the statistics task
#define OS_TASK_STAT_STK_SIZE 50   // Statistics task stack size
#define OS_ARG_CHK_EN         0    // Enable (1) or Disable (0) argument checking
#define OS_CPU_HOOKS_EN       1    // uC/OS-II hooks are found in the processor port
                                   // files
/* ----------------------- EVENT FLAGS ------------------------ */
#define OS_FLAG_EN            0    // code generation for EVENT FLAGS
#define OS_FLAG_WAIT_CLR_EN   0    // Include code for Wait on Clear EVENT FLAGS
#define OS_FLAG_ACCEPT_EN     0    // Include code for OSFlagAccept()
#define OS_FLAG_DEL_EN        0    // Include code for OSFlagDel()
#define OS_FLAG_QUERY_EN      0    // Include code for OSFlagQuery()
/* -------------------- MESSAGE MAILBOXES --------------------- */
#define OS_MBOX_EN            0    // code generation for MAILBOXES
#define OS_MBOX_ACCEPT_EN     0    // Include code for OSMboxAccept()
#define OS_MBOX_DEL_EN        0    // Include code for OSMboxDel()
#define OS_MBOX_POST_EN       0    // Include code for OSMboxPost()
#define OS_MBOX_POST_OPT_EN   0    // Include code for OSMboxPostOpt()
#define OS_MBOX_QUERY_EN      0    // Include code for OSMboxQuery()
/* --------------------- MEMORY MANAGEMENT -------------------- */
#define OS_MEM_EN             0    // Code generation for MEMORY MANAGER
#define OS_MEM_QUERY_EN       0    // Include code for OSMemQuery()
/* ---------------- MUTUAL EXCLUSION SEMAPHORES --------------- */
#define OS_MUTEX_EN           0    // code generation for MUTEX
#define OS_MUTEX_ACCEPT_EN    0    // Include code for OSMutexAccept()
#define OS_MUTEX_DEL_EN       0    // Include code for OSMutexDel()
#define OS_MUTEX_QUERY_EN     0    // Include code for OSMutexQuery()
/* ---------------------- MESSAGE QUEUES ---------------------- */
#define OS_Q_EN               0    // code generation for QUEUES
#define OS_Q_ACCEPT_EN        0    // Include code for OSQAccept()
#define OS_Q_DEL_EN           0    // Include code for OSQDel()
#define OS_Q_FLUSH_EN         0    // Include code for OSQFlush()
#define OS_Q_POST_EN          0    // Include code for OSQPost()
#define OS_Q_POST_FRONT_EN    0    // Include code for OSQPostFront()
#define OS_Q_POST_OPT_EN      0    // Include code for OSQPostOpt()
```

```
#define OS_Q_QUERY_EN         0    // Include code for OSQQuery()
/* ------------------------ SEMAPHORES ------------------------ */
#define OS_SEM_EN             1    // code generation for SEMAPHORES
#define OS_SEM_ACCEPT_EN      0    // Include code for OSSemAccept()
#define OS_SEM_DEL_EN         0    // Include code for OSSemDel()
#define OS_SEM_QUERY_EN       0    // Include code for OSSemQuery()
/* --------------------- TASK MANAGEMENT ---------------------- */
#define OS_TASK_CHANGE_PRIO_EN0    // Include code for OSTaskChangePrio()
#define OS_TASK_CREATE_EN     1    // Include code for OSTaskCreate()
#define OS_TASK_CREATE_EXT_EN 0    // Include code for OSTaskCreateExt()
#define OS_TASK_DEL_EN        1    // Include code for OSTaskDel()
#define OS_TASK_SUSPEND_EN    0    // Include code for OSTaskSuspend() and
                                   // OSTaskResume()
#define OS_TASK_QUERY_EN      0    // Include code for OSTaskQuery()
/* --------------------- TIME MANAGEMENT ---------------------- */
#define OS_TIME_DLY_HMSM_EN   0    // Include code for OSTimeDlyHMSM()
#define OS_TIME_DLY_RESUME_EN 1    // Include code for OSTimeDlyResume()
#define OS_TIME_GET_SET_EN    1    // Include code for OSTimeGet() and OSTimeSet()
/* ---------------------- MISCELLANEOUS ----------------------- */
#define OS_SCHED_LOCK_EN      1    // Include code for OSSchedLock() and
                                   // OSSchedUnlock()
#define OS_TICKS_PER_SEC      200  // Set the number of ticks in one second
```

10.12 Call µC/OS-II's Initialization Routine `OSInit()`

This OS routine initializes all of the kernel's linked lists and required parameters.

10.13 Create at Least One Task

The first task should usually be created from within your **`main()`** function. This first task would generally then create any additional tasks (obviously, many variations on this theme are possible). µC/OS-II allows tasks to be created using two slightly different methods. The first is backward compatible with µC/OS and uses its original **`OSTaskCreate()`** routine with the following arguments:

```
UBYTE  OSTaskCreate   (void  (*task)(void *pd), // ptr to task entry
                      void   *pdata,           // ptr to passed data (simulates CALL)
                      void   *pstk,            // ptr to task's top of user stack
                      UBYTE  prio);            // task's priority
```

The new, expanded task create routine allows for the inclusion of optional task-specific data and extensions:

```
#if OS_TASK_CREATE_EXT_EN
INT8U  OSTaskCreateExt(void  (*task)(void *pd), // ptr to tasks entry
                      void   *pdata,           // ptr to optional environment data
                      OS_STK *ptos,            // ptr to task's Top Of Stack
                      INT8U  prio,             // task's priority
                      INT16U id,               // task's ID
                      OS_STK *pbos,            // ptr to task's Bottom Of Stack
                      INT32U stk_size,         // task's stack size
                      void   *pext,            // pointer to optional TCB extension
                      INT16U opt)              // pointer to optional task-specific info
#endif
```

10.14 Finally, Call `OSStart()` to Begin Multitasking

Now, you simply pray that you've done everything correctly — simple!

11.0 Performance Numbers for µC/OS-II on the CR16C

The performance of any RTOS is affected by a number of different factors, not the least of which is the architecture of the machine on which it is running. As described earlier, the CR16C's instruction set architecture is tuned to achieve both high performance and good code density. Although these are most often *conflicting* demands, on the CR16C they reach a well-balanced compromise. The cycle timing of most of µC/OS-II's services are naturally compiler and compiler-option dependent, and they may vary slightly. Because the CR16C's instruction cycle is equivalent to the system clock (this would be your oscillator frequency if operating without the internal PLL, or, if using the PLL, this is your derived system clock rate), the cycle count can be easily translated into real time by multiplying with your oscillator's period. In a typical application, the CR16C achieves average CPI figures (Cycles Per Instruction) in the 1.6 to 1.8 range. Altogether, this results in extremely low OS overhead and plenty of CPU time for user tasks.

Figure 9 illustrates the *nominal* OS overhead associated with a context switch from the task level. Task A calls the µC/OS-II service `OSTimeDly()`, at which point the task is suspended until the desired delay expires. As shown in the diagram, the total nominal delay incurred by the application when using this kernel service is about 250 cycles for Supervisor mode or 313 cycles for User mode. This means that once Task A calls the OS, the highest priority task ready to run (Task B in the diagram) gains control of the CPU in about 10.4 or 13.1 microseconds at 24 MHz.

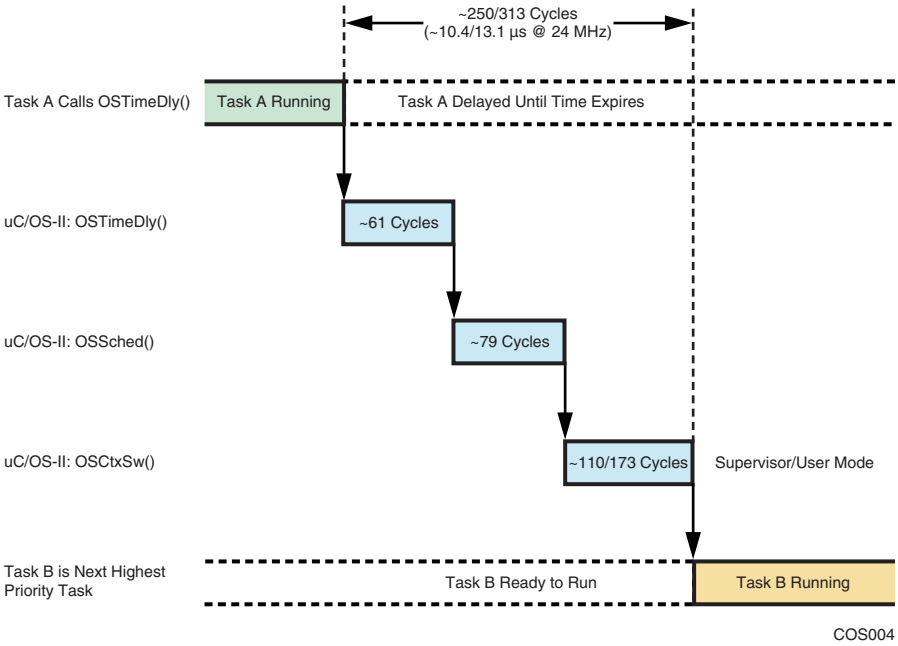

Figure 4. Context Switch Overhead

Another important RTOS performance index is that of interrupt response. What interrupt latency, response, and recovery times will I have to allow for under µC/OS-II? As you can see in Figure 5, interrupt response suffers by 75 to 125 additional cycles under µC/OS-II, depending on protection mode and interrupt nesting.

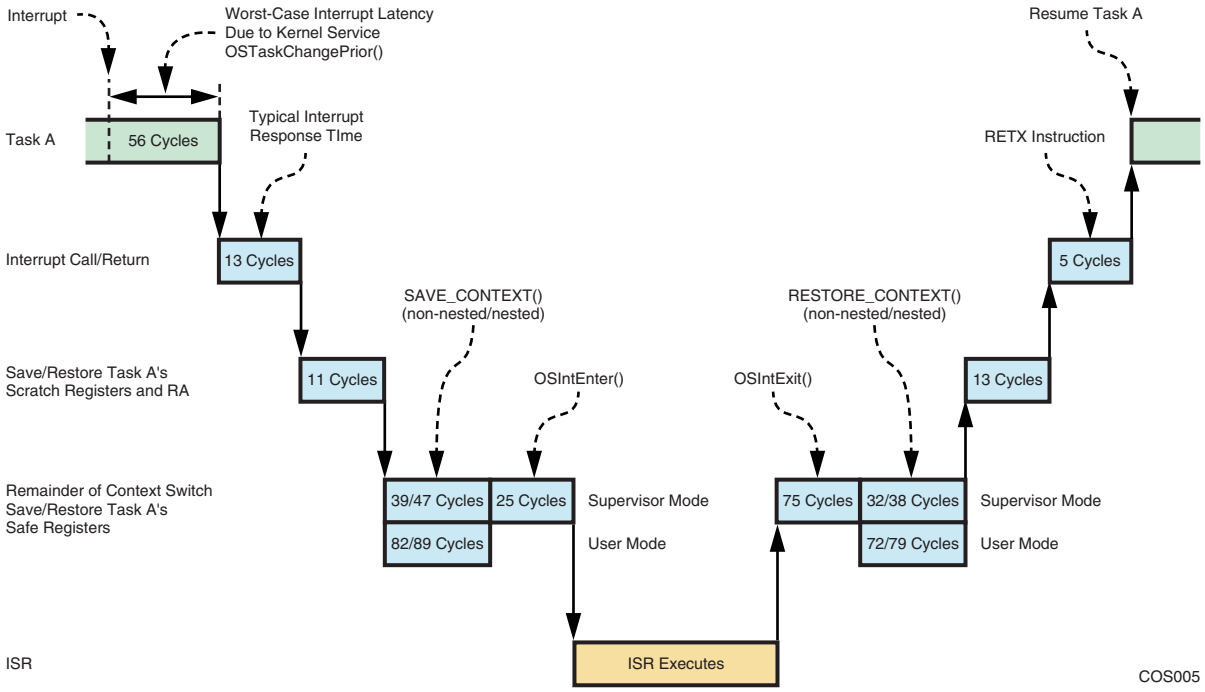

Figure 5. Interrupt Latency, Response, and Recovery Overhead Due to µC/OS-II

12.0 Porting µC/OS-II for the CR16C

µC/OS-II V2.70 was ported for use on the CR16C using both National Semiconductor's CR16C CompactRISC Toolset Version 3.1 and IAR's C/EC++ Compiler for CR16C V1.13A/W32 (1.13.1.3). National's compiler was invoked using the following options:

```
-Os -g -z -n -S
```

The complete source code for the CR16C port of µC/OS-II is available on National Semiconductor's web site.

13.0 Summary

µC/OS-II can be tuned to have both the smallest footprint and the lowest RAM utilization. This makes µC/OS-II a very attractive choice for many embedded applications. In addition, µC/OS-II's performance compares quite favorably with other RTOSs. Given these two factors, you may want to consider including µC/OS-II in your next design.

14.0 Resources

MicroC/OS-II, The Real-Time Kernel
Jean J. Labrosse
CMP Books, 2002
ISBN 1-57820-103-9

CompactRISC CR16C Programmers Reference Manual
National Semiconductor Corporation
Part Number: 424521772-101
December 2000

CompactRISC C Compiler Reference Manual
National Semiconductor Corporation
Part Number: 424521772-002
December 2000

Jean J. Labrosse
M I C R I U M , I N C .
949 Crestview Circle
Weston, FL 33327
U.S.A.

phone: +1 954 217 2036
fax: +1 954 217 2037
web: www.Micrium.com
www.uCOS-II.com
e-mail: Jean.Labrosse@Micrium.com

Appendix A: Context Switch Invoked From Task Level

```
// ********************************************************************************
// NAME:            OSCtxSw             Performs a context switch from the task level
//
// Notes:           1)   To minimize Task switching overhead, three distinct stack frame
//                       types are utilized. This technique leverages the fact that
//                       only the SAFE regs need be saved prior to a context switch
//                       invoked from the task level, because the calling task will have
//                       necessarily saved any/all scratch registers it is using. This
//                       is in contrast with the stack frame used for interrupts, since
//                       we return via the ISR.
//
//                  2)   Upon entry...
//
//                       OSTCBCur       points to the current task's OS_TCB
//                       OSTCBHighRdy   points to the OS_TCB of the task to resume
//
//                  3)   The Interrupt stack (@ISP) looks as follows:
//
//                        ISP->PSR      of current task
//                              +2      Address of OS_TASK_SW() instruction
//
// ********************************************************************************

OSCtxSw:                                //     This is the svc trap handler used to
                                        //     access the OS
#if USER_MODE > 0
          _SAVESAFE_                    // 27  Only the SAFE registers need saved here
                                        //     since we've arrived here via a call to an
                                        //     OS service.
#else
          _PUSHSAFE_                    // 13
#endif

#if USER_MODE > 0

          sprd      usp,(r7,r6)         // 1   Get user's TOS
          addd      $-8,(r7,r6)         // 1   Adjust for saving user's PC and PSR
          lprd      (r7,r6),usp         // 4   Update user's TOS
          movd      (r7,r6),(r12)       // 1   Save new TOS for writing to task's TCB

          sprd      isp,(r9,r8)         // 1   point to system stack area
          loadd     0(r9,r8),(r4,r3)    // 3   get user's PC
          addd      $1,(r4,r3)          // 1   adjust PC to return to next instruction
          loadw     4(r9,r8),r5         // 2   get user's PSW

          movw      $FRAME_TASK,r2      // 1   Indicate Task-level type of stack frame
```

```
          stormp    $4                  // 5   put user's PC and PSW on its user stack

          loadd     OSTCBCur,(r1,r0)    // 3   save program stack pointer in task's TCB
          stord     (r12),0(r1,r0)      // 3

#else

          sprd      isp,(r9,r8)         // 1   point to system stack area

          loadd     0(r9,r8),(r1,r0)    // 3   get user's PC
          addd      $1,(r1,r0)          // 1   adjust PC to return to next instruction
          loadw     4(r9,r8),r2         // 2   get user's PSW
          push      $3,r0               // 4   put user's PC and PSW on its user stack

          movw      $FRAME_TASK,r0      // 1   Indicate Task-level type of stack frame
          push      $1,r0               // 2

          loadd     OSTCBCur,(r1,r0)    // 3   save program stack pointer in task's TCB
          stord     (sp),0(r1,r0)       // 3

          loadd     OSStkPtr,(sp)       // 3   In SUPERVISOR mode, switch back to System
                                        //     stack for any stack usage by the kernel
#endif

          bal       (ra), OSTaskSwHook  // 6

          loadb     OSPrioHighRdy,r0    // 2   Update Current Priority
          storb     r0,OSPrioCur        // 2

          loadd     OSTCBHighRdy,(r1,r0)// 3   Get TCB of highest priority task ready to
                                        //     run
          stord     (r1,r0),OSTCBCur    // 3

#if USER_MODE > 0
          loadd     0(r1,r0),(r1,r0)    // 3   load new pgm stack pointer
                                        //     SP = OSTCBHighRdy->OSTCBStkPtr

          loadmp    $4                  // 6   Get the frame type, PC and PSW from
                                        //     user's stack
          stord     (r4,r3),0(r9,r8)    // 3   put task's PC on Interrupt Stack
          storw     r5,4(r9,r8)         // 2   put user's PSW on Interrupt Stack
                                        //     Determine the restore method commensurate
                                        //     with frame type
          cmpw      $FRAME_TASK,r2      // 1   Task type? - only safe restore
          beq       _TFrameTASK         // 1,3
#else
                                        //     Method0 has switched in the user's stack
                                        //     for push/pop
          loadd     0(r1,r0),(sp)       // 3   load new pgm stack pointer
```

```
                                        //     SP = OSTCBHighRdy->OSTCBStkPtr

          pop       $1,r3               // 3   Get the frame type
          pop       $3,r0               // 5   pop the PC, PSW, and frame type from
                                        //     user's stack
          stord     (r1,r0),0(r9,r8)    // 3   put task's PC into Interrupt Stack
          storw     r2,4(r9,r8)         // 2   put user's PSW into Interrupt Stack
                                        //     Jump to restore method commensurate with
                                        //     frame type
          cmpw      $FRAME_TASK,r3      // 1   Task type? - only safe restore needed...
          beq       _TFrameTASK         // 1,3
          cmpw      $FRAME_INIT,r3      // 1   INIT type? - a complete restore is
                                        //     indicated.
          beq       _TFrameINIT         // 1,3
#endif

_TFrameINT:                             //     Frame type INTerrupt (this task was
                                        //     preempted during an OS-aware ISR)
#if USER_MODE > 0

          _RESTOREALL_                  // 42  In USER mode, both types INT and INIT
                                        //     come here. We must always do a complete
          loadd     OSStkPtr,(sp)       // 3   restore. Reset System SP
          retx                          // 5   return to the task
#else
          popret    RA                  // 5?  In Supervisor mode, we return to the ISR
#endif
                                        //     (see INT type above)
#if USER_MODE > 0
#else
_TFrameINIT:                            //     Frame Type initial

          _POPALL_                      // 24
          retx                          // 5   return to task
#endif

_TFrameTASK:                            //     Frame type TASK (this task was preempted
                                        //     following a call to an OS service)
#if USER_MODE > 0

          _RESTORESAFE_                 // 26  Restore safe registers only and return
                                        //     to task

#else
          _POPSAFE_                     // 15  Pop the safe regs and return to the task
#endif

          retx                          // 5
```

Appendix B: Context Switch Invoked From Interrupt Level

```
// ********************************************************************************
// NAME:            OSIntCtxSw          Performs a context switch from within an ISR
//
// Notes:           1)   Previous versions relied upon a known, consistent stack frame
//                       upon entry. Due to uncertainties arising from various factors
//                       (compiler optimization, volatile local var's, etc.), I decided
//                       to abandon this technique and always return via the ISR. While
//                       this does introduce additional, variable delays, it is the only
//                       method guaranteed to avoid these potential sources of error...
//
//                  2)   Upon entry...
//
//                  OSTCBCur            points to the current task's OS_TCB
//                  OSTCBHighRdy        points to the OS_TCB of the task to resume
//
// ********************************************************************************

OSIntCtxSw:

#if USER_MODE > 0

          sprd      usp,(r7,r6)         // 1   Get user's TOS
          addd      $-8,(r7,r6)         // 1   Adjust for saving user's PC and PSR
          lprd      (r7,r6),usp         // 4   Save new user's TOS
          movd      (r7,r6),(r12)       // 1   Save this for later update of user's TCB

          sprd      isp,(r9,r8)         // 1   point to system stack area
          loadd     0(r9,r8),(r4,r3)    // 3   get user's PC
          loadw     4(r9,r8),r5         // 2   get user's PSW
          movw      $FRAME_INT,r2       // 1   Indicate Interrupt-level type of stack
                                        //     frame
          stormp    $4                  // 5   put user's PC and PSW on its user stack

          loadd     OSTCBCur,(r1,r0)    // 3   save program stack pointer in task's TCB
          stord     (r12),0(r1,r0)      // 3
#else
          sprd      isp,(r9,r8)         // 1   point to system stack area

          loadd     0(r9,r8),(r1,r0)    // 3   get user's PC
          loadw     4(r9,r8),r2         // 2   get user's PSW
          push      $3,r0               // 4   put user's PC and PSW on its user stack

          movw      $FRAME_INT,r0       // 1   Indicate Interrupt-level type of stack
                                        //     frame
          push      $1,r0               // 2
```

```
          loadd     OSTCBCur,(r1,r0)     // 3   SP = OSTCBCur->OSTCBStkPtr
          stord     (sp),0(r1,r0)        // 3

          loadd     OSStkPtr,(sp)        // 3   In SUPERVISOR mode, switch back to System
                                         //     stack for any stack usage by the kernel
#endif

          bal       (ra), OSTaskSwHook   // 6

          loadb     OSPrioHighRdy,r0     // 2   OSPrioCur = OSPrioHighRdy
          storb     r0,OSPrioCur         // 2

          loadd     OSTCBHighRdy,(r1,r0)// 3   Get TCB of highest priority task ready
                                         //     to run
          stord     (r1,r0),OSTCBCur     // 3

#if USER_MODE > 0
                                         //     Method1 must use load/store instructions
                                         //     to avoid using the user's stack (in case
                                         //     of NMI)
          loadd     0(r1,r0),(r1,r0)     // 3   load new pgm stack pointer
                                         //     SP = OSTCBHighRdy->OSTCBStkPtr//

          loadmp    $4                   // 6   Get the frame type, PC and PSW from
                                         //     user's stack
          stord     (r4,r3),0(r9,r8)     // 3   put task's PC into Interrupt Stack
          storw     r5,4(r9,r8)          // 2   put user's PSW into Interrupt Stack
                                         //     Jump to restore method commensurate with
                                         //     frame type
          cmpw      $FRAME_TASK,r2       // 1   Task Type - only safe restore
          beq       _IFrameTASK          // 1, 3

#else
          loadd     0(r1,r0),(sp)        // 3   load new pgm stack pointer
                                         //     SP = OSTCBHighRdy->OSTCBStkPtr

          pop       $1,r3                // 3   Get the frame type

          pop       $3,r0                // 5   pop the PC and PSW from user's stack
          stord     (r1,r0),0(r9,r8)     // 3   put task's PC into Interrupt Stack
          storw     r2,4(r9,r8)          // 2   put user's PSW into Interrupt Stack

          cmpw      $FRAME_TASK,r3       // 1   Task type - only safe restore
          beq       _IFrameTASK          // 1, 3
          cmpw      $FRAME_INIT,r3       // 1   Init Type - complete restore
          beq       _IFrameINIT          // 1, 3
#endif
```

```
_IFrameINT:                             //    Interrupt Frame Type...

#if USER_MODE > 0

          _RESTOREALL_                  // 42  macro - restore all registers
          loadd     OSStkPtr,(sp)       // 3   Reset System SP
          retx                          // 5   return to highest priority task

#else
          popret    RA                  // 5?  return via the ISR
#endif

#if USER_MODE > 0
#else
_IFrameINIT:                            //    Initial Frame Type

          _POPALL_                      // 24  macro - restore all registers
          retx                          // 5   return to highest priority task
#endif

_IFrameTASK:                            //    Task Frame type...

#if USER_MODE > 0

          _RESTORESAFE_                 // 26  macro - restore safe registers

#else

          _POPSAFE_                     // 15  macro - restore safe registers

#endif

          retx                          // 5   return to highest priority task
```

Appendix C: Save Task's Context from within an ISR

```
// ********************************************************************************
// NAME:            OSSaveContext       Saves task's context from within an ISR
//
// The task's partially saved (by the isr) stack frame will typically look as follows
// upon entry:
//
//        Offset    SUPERVISOR MODE     USER MODE
//        ------    ---------------     ---------------
//        +16       RA(hi)              N/A (these regs were pushed to the system stack)
//        +14       RA(lo)
//        +12       r6
//        +10       r5
//        +8        r4
//        +6        r3
//        +4        r2
//        +2        r1
//        (sp)+0    r0
//
// HOWEVER, due to uncertainties accompanying compiler optimization levels, ISR
// construction, etc., the context may appear different than above.
//
// After the balance of the context is saved, the task's stack frame will look as follows
// (typical):
//
//        Offset    SUPERVISOR MODE     USER MODE
//        ------    ---------------     ---------------
//
//        +42       RA(hi)              r7     (High memory)
//        +40       RA(lo)              r6
//        +38       r6                  r1
//        +36       r5                  r0
//        +34       r4                  RA(hi)
//        +32       r3                  RA(lo)
//        +30       r2                  r13(hi)
//        +28       r1                  r13(lo)
//        +26       r0                  r12(hi)
//        +24       r7                  r12(lo)
//        +22       r13(hi)             r11
//        +20       r13(lo)             r10
//        +18       r12(hi)             r9
//        +16       r12(lo)             r8
//        +14       r11                 r5
//        +12       r10                 r4
//        +10       r9                  r3
//        +8        r8                  r2
//        +6        PSR                 PSR
//        +4        task address(hi)    task address(hi)
```

```
//        +2        task address(lo)    task address(lo)
//        (sp)+0    Frame Type = INT    Frame type = INT (Low memory)
//
// ********************************************************************************

OSSaveContext:

#if OS_NEST_EN > 0
          push      $1,r0               // 2   If nesting is supported, we must always
                                        //     increment the nest counter upon entry to
                                        //     an OS-aware ISR, as well as in ALL
          loadb     OSIntNesting,r0     // 4   non OS-aware ISR's that allow nesting
                                        //     (since another interrupting "OS-aware" ISR
          addb      $1,r0               // 1   may invoke a ctx sw)
          storb     r0,OSIntNesting     // 5

          cmpb      $1,r0               // 1   Nested interrupt??
          bne       _ISaveExit1         // 1   YES - no need to save context..goto EXIT
          pop       $1,r0               // 3   NO, we must save context.
#endif

#if USER_MODE > 0
          push      RA                  // 3   Save the return address (ra) from
                                        //     interrupted task's context. In USER mode,
                                        //     all regs must be written to the
          _SAVEALL_                     // 46  the task's stack, despite the fact that
                                        //     they've already been pushed to the
                                        //     supervisor's stack!!
#if OS_NEST_EN == 0
          loadb     OSIntNesting,r0     // 5   If nesting is disabled, increment the nest
                                        //     counter
          addb      $1,r0               // 1   (Even if nesting is disabled, the nest
                                        //     counter is decremented in OSIntExit).
          storb     r0,OSIntNesting     // 4
#endif

          popret    RA                  //     return to ISR

#else

          push      $1,r7               //     In SUPERVISOR mode, save remaining SAFE
                                        //     registers (excluding RA, since it was
          push      $8,r8               //     saved at entry to ISR)

#if OS_NEST_EN == 0
          loadb     OSIntNesting,r0     // 5   Increment the nesting counter..
          addb      $1,r0               // 1   (Even if nesting is disabled, the nesting
          storb     r0,OSIntNesting     //     counter is decremented in OSIntExit).
#endif
```

```
          jump      (RA)                //    Return to the ISR

#endif

_ISaveExit1:
          pop       $1,r0               //    No context save required..
          jump      (RA)
```

Appendix D: Context Initialization

```
/*
*********************************************************************************
*                    INITIALIZE A TASK'S STACK
*
* Description:       This function is called by either OSTaskCreate() or OSTaskCreateExt()
*                    to initialize the stack frame of the task being created. This
*                    function is highly processor specific.
*
* Arguments:         task                is a pointer to the task code
*
*                    pdata               is a pointer to a user supplied data area that
*                                        will be passed to the task when the task first
*                                        executes.
*
*                    ptos                is a pointer to the top of stack. It is assumed
*                                        that 'ptos' points to a 'free' entry on the task
*                                        stack. Since stack growth is from High to Low
*                                        Mem on the CR16B, 'ptos' must contain the HIGHEST
*                                        valid address of the stack.
*
*                    opt                 specifies options that can be used to alter the
*                                        behavior of OSTaskStkInit().
*                                        (See ucos_ii.h for task options.)
*
* Returns:           Always returns the location of the new top-of-stack' once the
*                    processor registers have been placed on the stack in the proper
*                    order.
*
* Note(s):           Interrupts are enabled when your task starts executing. You can
*                    change this by setting the PSR to 0x0800 instead. In this case,
*                    interrupts would be disabled upon task startup. The application code
*                    would be responsible for enabling interrupts at the beginning of the
*                    task code. ** NOTE THAT IN THIS CASE, YOU WILL ALSO NEED TO modify
*                    OSTaskIdle() and OSTaskStat() so that they enable interrupts.
*                    Failure to do this will make your system crash!
*
*********************************************************************************
*/

OS_STK *OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt)
{
          ULONG     *stk;
          OS_STK    *ptr;
```

```
#ifdef __ICCCR16C__
                              /* We must be specific with IAR since function pointers*/
     typedef void (** PTASK)(void *);   /* may not be cast to ULONG without a shift!!*/
          PTASK     ptask;

#endif

          opt =     opt;                /* 'opt' is not used, prevent warning       */
          stk =     (ULONG *)ptos;      /* Get Top of Stack word pointer            */

#if USER_MODE > 0

          *--stk =  (ULONG)0x00070006;  /* R7,R6 = 7,6                              */
          *--stk =  (ULONG)0x00010000;  /* R1,R0 = 1,0                              */
          *--stk =  (ULONG)0x000E000E;  /* R14H/L (RA) = 14,14                      */
          *--stk =  (ULONG)0x000D000D;  /* R13 = 13,13                              */
          *--stk =  (ULONG)0x000C000C;  /* R12 = 12,12                              */
          *--stk =  (ULONG)0x000B000A;  /* R11,R10 = 11,10                          */
          *--stk =  (ULONG)0x00090008;  /* R9,R8 = 9,8                              */
          *--stk =  (ULONG)0x00050004;  /* R5,R4 = 5,4                              */
          *--stk =  (ULONG)pdata;       /* R3,R2 = passed pointer                   */

#else

          *--stk =  (ULONG)0x000E000E;  /* R14H/L (RA) = 14,14                      */
          *--stk =  (ULONG)0x00060005;  /* R6,R5 = 6,5                              */

          ptr =     (OS_STK*)(void*)stk;/* Set WORD pointer to next slot            */
          *--ptr =  (UWORD)0x0004;      /* R4 = 4                                   */
          stk =     (ULONG*)(void*)ptr; /* Set ULONG pointer to next spot           */

          *--stk =  (ULONG)pdata;       /* R3,R2 = passed pointer                   */
          *--stk =  (ULONG)0x00010000;  /* R1,R0 = 1,0                              */

          ptr =     (OS_STK*)(void*)stk;/* Set WORD pointer to next slot            */
          *--ptr =  (UWORD)0x0007;      /* R4 = 4                                   */
          stk =     (ULONG*)(void*)ptr; /* Set ULONG pointer to next spot           */

          *--stk =  (ULONG)0x000D000D;  /* R13 = 0                                  */
          *--stk =  (ULONG)0x000C000C;  /* R12 = 0                                  */
          *--stk =  (ULONG)0x000B000A;  /* R11,R10 = 0                              */
          *--stk =  (ULONG)0x00090008;  /* R9,R8 = 0                                */

#endif

          ptr =     (OS_STK*)(void*)stk;
```

```
#if USER_MODE > 0
          *--ptr =  (UWORD)0x0a08;      /* Initialize PSR with ints enabled and      */
                                        /* USER mode                                 */
#else
          *--ptr =  (UWORD)0x0a00;      /* Initialize PSR with ints enabled and      */
                                        /* SUPERVISOR mode                           */
#endif

#ifdef __ICCCR16C__

          ptask =   (PTASK)ptr;         /* Set pointer to next spot                  */
          *--ptask = task;              /* Put Task's address on top of stack        */
          ptr =     (OS_STK*)ptask;
#else

          stk =     (ULONG*)(void*)ptr; /* Set pointer to next spot                  */
          *--stk =  (ULONG)task;        /* Put Task's address on top of stack        */
          ptr =     (OS_STK*)(void*)stk;

#endif

          *--ptr =  FRAME_INIT;         /* Define the initial Frame type as Interrupt*/
                                        /* to ensure a full restore...               */
          return    ((OS_STK *)ptr);    /* This will be the Task's initial SP value  */
}
```

Appendix E: µC/OS-II Macros

```
/*********************************************************************************************
*                   uC/OS-II
*                   CR16C-Specific Kernel macros
* File:             os_cpu.h
* By:               Jeffrey Wright
*********************************************************************************************/
#ifndef   CR16CCH
#define   CR16CCH

#ifdef    OS_CPU_GLOBALS
#define   OS_CPU_EXT
#else
#define   OS_CPU_EXT extern
#endif
                                        /* Task's Stack frames may come in these     */
                                        /* flavors...                                */
#define   FRAME_INIT     1              /* Initial frame type                        */
#define   FRAME_TASK     2              /* After task calls OS service               */
#define   FRAME_INT      3              /* After task is interrupted                 */

#ifdef    __ICCCR16C__
#include  <intrinsics.h>
#include  <string.h>
#else
#include  <asm.h>

#endif

/*
*********************************************************************************************
*                   FUNCTION PROTOTYPES
*                   (Compiler Specific ISR prototypes)
*********************************************************************************************
*/

#ifdef    __ICCCR16C__
__trap void OSCtxSw(void);
#else
void      OSCtxSw(void);
#endif

void      OSTickISRHandler(void);       /* This is the generic tick handler called by*/
                                        /* the app                                   */
void      OSTIMER_EN(void);             /* This is an extern declaration, filled by  */
                                        /* the app                                   */
void      OSSaveContext(void);          /* These are required since IAR's inline     */
void      OSRestoreContext(void);       /* assbly support is inadequate..            */
```

```
void      OSUseSysStk(void);
void      OSUseTaskStk(void);

void      OSStartHighRdy(void);
void      OSIntCtxSw(void);

/*****************************************************************************
*                  DATA TYPES
*                  (Compiler Specific)
*****************************************************************************/
typedef unsigned    char      BOOLEAN;
typedef unsigned    char      INT8U;    /* Unsigned 8 bit quantity                */
typedef signed      char      INT8S;    /* Signed 8 bit quantity                  */
typedef unsigned    short     INT16U;   /* Unsigned 16 bit quantity               */
typedef             short     INT16S;   /* Signed 16 bit quantity                 */
typedef unsigned    long      INT32U;   /* Unsigned 32 bit quantity               */
typedef signed      long      INT32S;   /* Signed 32 bit quantity                 */
typedef             float     FP32;     /* Single precision floating point        */
typedef             double    FP64;     /* Double precision floating point        */

/*****************************************************************************
*                  Legacy DATA TYPES
*                  (Included for legacy code)
*****************************************************************************/
#define             BYTE      INT8S     /* Define data types for backward         */
#define             UBYTE     INT8U     /* compatibility to uC/OS V1.xx.          */
#define             WORD      INT16S    /* Not actually needed for uC/OS-II.      */
#define             UWORD     INT16U
#define             LONG      INT32S
#define             ULONG     INT32U
#define             UCHAR     char
typedef volatile    INT8U     VUBYTE;   /* Volatile Unsigned 8 bit quantity       */
typedef volatile    INT16U    VUWORD;   /* Volatile Unsigned 16 bit quantity      */
typedef volatile    INT32U    VULONG;   /* Volatile Unsigned 32 bit quantity      */

#define   OS_STK_GROWTH  1              /* Stack grows from HIGH to LOW memory    */

typedef             INT16U    OS_STK;   /* Each stack entry is 16-bit wide        */

OS_CPU_EXT          ULONG     OSStkPtr; /* Used in SUPERVISOR MODE on the CR16C   */

/*****************************************************************************
*                  uC/OS-II allows two methods of Disabling/Enabling interrupts:
*
* Method #1:        Disable/Enable interrupts using simple instructions. After critical
*                   section, interrupts will be enabled even if they were disabled before
*                   entering the critical section. You MUST change the constant in
*                   OS_CPU_A.ASM, function OSIntCtxSw() from 10 to 8.
*
```

```
* Method #2:        Disable/Enable interrupts by preserving the state of interrupts. In
*                   other words, if interrupts were disabled before entering the critical
*                   section, they will be disabled when leaving the critical section.
*                   You MUST change the constant in OS_CPU_A.ASM, function OSIntCtxSw()
*                   from 8 to 10.
*********************************************************************************/

#ifdef    __ICCCR16C__                  /* IAR version...                          */

#define OS_ENTER_CRITICAL()   __disable_interrupt()
#define OS_EXIT_CRITICAL()    __enable_interrupt()
#define ENABLE_INTERRUPTS()   __set_PSR_I_bit()
#define DISABLE_INTERRUPTS()  __clear_PSR_I_bit()
#define OS_TASK_SW()          __raise_exception(5)
                                        /* This invokes the svc trap as a system call*/
#define __ASS__               asm

#else                                   /* NSC                                     */

#define OS_ENTER_CRITICAL()   _di_()    /* Disable interrupts                      */
#define OS_EXIT_CRITICAL()    _ei_()    /* Enable interrupts                       */
#define ENABLE_INTERRUPTS()   set_i_bit()
#define DISABLE_INTERRUPTS()  clear_i_bit()
#define OS_TASK_SW()          _excp_(5) /* This uses the svc trap as a system call   */
#define __ASS__               __asm__

#endif

/*
*********************************************************************************
*                   CONTEXT SAVE/RESTORE MACROS
*********************************************************************************
*/

/*
* NOTE:
* Due to the IAR comiler's inadequate inline assembler support, assembler subroutines
* are now utilized inplace of the previous version's use of inline assembler macros.
* This increases overhead by a few additional cycles, but is nonetheless unavoidable.
* Furthermore, to maintain consistency between the two, the NSC port now also utilizes
* this method.
*/

#define USE_SYSTEM_STK()      OSUseSysStk();
#define USE_TASK_STK()        OSUseTaskStk();
#define SAVE_CONTEXT()        OSSaveContext();
#define RESTORE_CONTEXT()     OSRestoreContext();

#endif
```

Appendix F: Assembly Macros

```
// Task stack frames come in these flavors...
#define FRAME_INIT       1              // Initial
#define FRAME_TASK       2              // After task calls an OS service
#define FRAME_INT        3              // After task is interrupted

// ************** For use with SUPERVISOR mode kernels ****************

#ifdef    __ACR16C__
_USE_USER_STACK_    MACRO
#else
.macro    _USE_TASK_STACK_
#endif
          stord     (sp),OSStkPtr       // Save System SP for use by nested interrupts
          loadd     OSTCBCur:_SIZE,(r1,r0)
          loadd     0(r1,r0),(sp)
#ifdef    __ACR16C__
          ENDM
#else
.endm
#endif

#ifdef    __ACR16C__
_USE_SYSTEM_STACK_  MACRO
#else
.macro    _USE_SYSTEM_STACK_
#endif
          loadd     OSTCBCur,(r1,r0)
          stord     (sp),0(r1,r0)
          loadd     OSStkPtr,(sp)
#ifdef    __ACR16C__
          ENDM
#else
.endm
#endif

// ************** ALTERNATE METHOD USING STORMP/LOADMP INSTRUCTIONS...

#ifdef    __ACR16C__
_SAVESAFE_          MACRO
#else
.macro    _SAVESAFE_
#endif
          push      $1,r7               // 2   Save R7
          sprd      usp,(r7,r6)         // 1   Get user's TOS
          addd      $-22,(r7,r6)        // 1   Adjust ptr to save
          lprd      (r7,r6),usp         // 4   Load new user TOS
          movd      (r12),(r3,r2)       // 1   Copy remaining regs to
```

```
          movd      (r13),(r5,r4)       // 1
          stormp    $8                  // 9   Save R12,13,8,9,10,11
          pop       $1,r2               // 3   Reload R7
          movd      (ra),(r4,r3)        // 1   Copy r14
          stormp    $3                  // 4   Save R7,r14
#ifdef    __ACR16C__
          ENDM                          // 27  cycles
#else
.endm
#endif

#ifdef    __ACR16C__
_RESTORESAFE_       MACRO
#else
.macro    _RESTORESAFE_
#endif
          loadmp    $8                  // 10  Load R12,13,8,9,10,11
          movd      (r3,r2),(r12)       // 1   Copy remaining regs to
          movd      (r5,r4),(r13)       // 1   default for saving
          loadmp    $3                  // 5   Load R7,r14
          movw      r2,r7               // 1   Copy R7
          movd      (r4,r3),(ra)        // 1   Copy Ra
          lprd      (r1,r0),usp         // 4   Load new user TOS
          loadd     OSStkPtr,(sp)       // 3   Reset System SP
#ifdef    __ACR16C__
          ENDM                          // 26  cycles
#else
.endm
#endif

#ifdef    __ACR16C__
_SAVEALL_           MACRO
#else
.macro    _SAVEALL_
#endif
          push      $2,r6               // 4   Save user's R6,7
          loadd     OSStkPtr,(r7,r6)    // 3   context...we must save the ra of the
                                        //     isr first!!
          addd      $-4,(r7,r6)         // 1
          loadd     0(r7,r6),(ra)       // 3
          sprd      usp,(r7,r6)         // 1   Get user's TOS
          addd      $-36,(r7,r6)        // 1   Adjust ptr to new TOS
          lprd      (r7,r6),usp         // 4   Load new user TOS
          stormp    $8                  // 9   Save R2,3,4,5,8,9,10,11
          movd      (r12),(r3,r2)       // 1   Copy R12,13,14,0,1
          movd      (r13),(r5,r4)       // 1   to permit storing
          movd      (ra),(r9,r8)        // 1
          movd      (r1,r0),(r11,r10)   // 1
          stormp    $8                  // 9   Save R12,13,14,0,1
```

```
          pop       $2,r2               // 4   Restore user's R6,7
          stormp    $2                  // 3   Save R6,7
#ifdef    __ACR16C__
          ENDM                          // 46  cycles
#else
.endm
#endif

#ifdef    __ACR16C__
_RESTOREALL_        MACRO
#else
.macro    _RESTOREALL_
#endif
          addd      $32,(r1,r0)         // 1   Adjust ptr for R6,7
          loadmp    $2                  // 4   Load R6 and R7
          movd      (r3,r2),(r7,r6)     // 1   Copy R6,R7
          lprd      (r1,r0),usp         // 4   Load new user TOS
          addd      $-20,(r1,r0)        // 1   Adjust pointer for next group
          loadmp    $8                  // 10  Load R12,13,14,0,1
          movd      (r3,r2),(r12)       // 1   Copy remaining regs to
          movd      (r5,r4),(r13)       // 1
          movd      (r9,r8),(ra)        // 1
          push      $2,r10              // 3   Save R1,R0
          addd      $-32,(r1,r0)        // 1   Adjust pointer for next group
          loadmp    $8                  // 10  Load R2,3,4,5,8,9,10,11
          pop       $2,r0               // 4   Get R1,R0
#ifdef__ACR16C__
          ENDM                          // 42  cycles
#else
.endm
#endif
// ************** For use with SUPERVISOR mode kernels ****************

#ifdef    __ACR16C__
_POPALL_            MACRO
#else
.macro    _POPALL_
#endif
          pop       $8,r8               // 10
          pop       $1,r7               // 3
          pop       $7,r0,RA            // 11
#ifdef    __ACR16C__
          ENDM                          // 24  cycles
#else
.endm
#endif
```

```
#ifdef    __ACR16C__
_PUSHSAFE_          MACRO
#else
.macro    _PUSHSAFE_
#endif
          push      $1,r7               // 2
          push      $8,r8,RA            // 11
#ifdef    __ACR16C__
          ENDM                          // 13
#else
.endm
#endif

#ifdef    __ACR16C__
_POPSAFE_           MACRO
#else
.macro    _POPSAFE_
#endif
          pop       $8,r8,RA            // 12
          pop       $1,r7               // 3
#ifdef    __ACR16C__
          ENDM
#else
.endm
#endif
```

Notes

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor
Americas Customer
Support Center
Email: new.feedback@nsc.com
Tel: 1-800-272-9959

www.national.com

National Semiconductor
EuropeCustomer
Support Center
Fax: +49 (0) 180-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 69 9508 6208
English Tel: +44 (0) 870 24 0 2171
Francais Tel: +33 (0) 1 41 91 8790

National Semiconductor
Asia Pacific Customer
Support Center
Email: ap.support@nsc.com

National Semiconductor
Japan Customer
Support Center
Fax: 81-3-5639-7507
Email: jpn.feedback@nsc.com
Tel: 81-3-5639-7560