CoRTOS - The World's Simplest RTOS

*Truth in advertising: One of the World's Simplest
Real Time Operating Systems.

Version 1.10 - 12 December 2018

Introduction & Overview

CoRTOS is an open-source Cooperative Real Time Operating System for bare-metal applications. It was developed and released by Cleveland Engineering Design, LLC. CoRTOS's advantages are simplicity, size and speed. To get an idea of its size you can turn to the listing on page 6 and see that the kernel takes only 16 lines of C code.

CoRTOS is a full-featured multitasking operating system with independent tasks; it is not a task scheduler. Features include:

- Task delays
- Signals
- Periodic signals
- Messages
- Semaphores
- Timeouts and timers

System responsiveness is maximized with a design that allows interrupts to always be enabled. Nested interrupts can make signaling calls to the OS.

CoRTOS is intended for smaller microprocessors, such as the AVR, Cortex M0/+, MSP430 and PIC24 and smaller software systems of maybe a dozen tasks.

The intended audience includes:

- Those needing a small footprint RTOS for moderately complex products such as IOT, appliances, industrial controls, sensors, and toys;
- Students learning about Real Time techniques;
- Makers wanting to program ‘close to the metal.’

CLEVELAND ENGINEERING DESIGN, LLC

Nicholas O. Lindan
Cleveland Engineering Design, LLC
1412 Dorsh Road, Cleveland Ohio 44121 USA
nolindan@ix.netcom.com - 216-691-3980

CoRTOS Manual v110.pdf - 12 December 2018

Contents

Suggested Reading Order

Every attempt has been made to write this booklet so it gives a logical presentation of CoRTOS. As the audience is so varied there is the danger of this document being trivial to experienced engineers and impenetrable to students.

If you are experienced with real time systems, and are looking for a quick overview of CoRTOS, then you might want to look at:

If you are starting out then the following should get you going:

The Basics of RTOS

"Real Time" software doesn't have much to do with time, real or virtual. Real time software is all about doing things in the real world and getting them done on time.

This isn't all that hard for simple embedded systems as the real world is much, much slower than a μP. A typical 16MHz processor executes 160,000 instructions in the typical 10mS RTOS 'tick'. Tasks don't normally get pre-empted by a "your time is up" tick - they have pre-empted themselves well before then.

As a rule, real time software in small systems doesn't do much that is cyber-cerebral. Typical calculations include sensor linearization, simple filtering, PID algorithms and calculating acceleration profiles - usually performed at leisure while waiting for worldly things to happen.

As a result, real time software spends the great majority of its time waiting for something in the real world, be it a motor, a display or the operator, to get its job done. When a signal comes into the μP, telling it of an event, the software springs into a flurry of activity that ends with the μP sending out signals telling the real world what to do next, after which the software goes back to waiting.

When the software is waiting for something it is said to be "blocked." Blocking is a fundamental concept in real time software. The software may be blocked by a multitude of factors: waiting for time to pass; a signal from an interrupt routine; an incoming communications message; a position sensor . . . When one task is blocked another task can execute - juggling all this is the job of the RTOS.

A taxonomy of real time software organization shows an evolution of ideas:

- Spaghetti all covered with gotos sends the software flow hither and yon in response to inputs. For very, very simple systems this works. For anything bigger than a few pages of code the goto's can be a real problem as the code can be close to impossible to understand and verify. But even in sophisticated structured systems a bit of spaghetti can be a help.

- Big loop programs are appropriate for systems that respond to one input at a time, with maybe an interrupt or two to run an A/D or timer. When it is blocked, the code may sit in a small loop calling a background process. To get around the limitation of only one block the big loop may call a series of little loops which, when blocked, each remember where they are and then return to the big loop. When called again by the big loop they pick up where they left off. As pop singers tell us, life is but a succession of circles within circles.

- Scheduler programs call functions or pseudo-tasks. The tasks run to completion (or as far as they can) and then return to the scheduler. Returning tasks are marked inactive, to be made active again by an ISR or another task. The scheduler then runs the highest priority ready-to-run task. The scheduler may run some tasks at periodic intervals. Alternatively, tasks can be equal

priority and rotate to the end of a list when they return. This scheme is an unrolling of the big loop - the big loop becomes the scheduler and the little loops become the pseudo-tasks called by the scheduler.

- Multitasking Real Time Operating Systems bring autonomy to the tasks. Each task has its own stack and is often called a 'thread' of execution - the thread running through the return addresses on each task's stack. Tasks block when they need to, relinquishing control to the next unblocked task (though preemptive systems can give control to another task without a "by your leave"). When the block clears, the task picks up where it left off. This makes task design easier - each task becomes a small "big loop" in its own right, responsible for the thing(s) it owns and is in control of. The operating system is responsible for applying and removing the blocks. An RTOS typically offers many different blocks. CoRTOS offers the following:

Call to RTOS	Task waits for
`delay()`	A specified time;
`wait_for_message()`	A message from another task
`acquire_semaphore()`	Sole ownership of a shared resource (printer, etc.) or synchronizing multiple tasks;
`wait_for_signal()`	A signal from an ISR or task.

Multitasking Real Time Operating Systems come in two varieties:

- Cooperative RTOS systems, such as CoRTOS, work on the honor system. Each task executes until it is blocked and then relinquishes control to the next task. When a task is no longer blocked it continues execution and again sets up the I/O for the next action the real world is to take. Tasks can have equal priority and execute round-robin style or they can be prioritized. Cooperative systems are very fast. Although there is a processing burden with each blocked task, this per task overhead can be as little as 1 μSecond with each system call. Cooperative RTOS's are easy for the user to expand with new features, in this they are unlike preemptive RTOS kernels that require in depth knowledge of the kernel's subtleties before attempting to make any change to its operation.

- Prioritized Preemptive RTOS systems can take control from a low priority task and give it to a higher priority task in response to an unblocking event. The lower priority task does not voluntarily relinquish control first. Preemptive systems are far more complex than cooperative systems. Preemptive kernels can run to 16K of code while cooperative systems rarely exceed a few hundred instructions. Preemptive systems impose zero overhead on blocked and preempted non-executing tasks and can control systems with hundreds of tasks. They find their place in x86 and large ARM microprocessors acting as master system controllers. Preemptive schedulers are at the innermost kernel of computer operating systems like Linux. All software is, after all, real-time in some sense.

The CoRTOS Kernel and Basic Task Switching

The kernel code is presented here without comments. The entirety of the CoRTOS task management kernel is the function **relinquish()**, lines 9-28, implemented in 16 lines of executable C. A commented version is provided in the distribution files. The code below is generic/AVR; the distribution files accommodate processor differences.

```
 1 #include "common_defs.h"
 2 #include "CoRTOSkernel.h"
 3 #include "CoRTOStask.h"

 4 uint8_t current_task;

 5 static uint16_t sp_save [number_of_tasks];
 6 static uint16_t starting_stack [number_of_tasks];
 7 static boolean start_from_beginning [number_of_tasks];
 8 static boolean suspended [number_of_tasks];

 9 void relinquish (void) {
10    asm volatile ("NOP":::"r2","r3","r4","r5","r6","r7","r8","r9","r10",\
11       "r11","r12","r13","r14","r15","r16","r17", "r28", "r29");
12    sp_save[current_task] = _SP;
13    while (true) {
14       do {
15          if (++current_task == number_of_tasks) current_task = 0;
16       } while (suspended[current_task] == true);
17       if (start_from_beginning[current_task] == true) {
18          start_from_beginning[current_task] = false;
19          _SP = starting_stack[current_task];
20          start_addresses[current_task] ();
21          suspended[current_task] = true;
22          start_from_beginning[current_task] = true;
23       } else {
24          _SP = sp_save[current_task];
25          return;
26       }
27    }
28 }

29 void suspend (void) {
30    suspended[current_task] = true;
31 }

32 void resume_task (uint8_t tn) {
33    suspended[tn] = false;
34 }

35 void start_CoRTOS (void) {
36    uint8_t tn;
37    uint16_t spv;

38    spv = _SP;
39    for (tn = 0; tn < number_of_tasks; tn++) {
40       starting_stack[tn] = spv;
41       spv -= task_stack_size[tn];
42       start_from_beginning[tn] = true;
43       suspended[tn] = false;
44    }
45    start_from_beginning[0] = false;
46    current_task = 0;
47    start_addresses[0] ();
48 }
```

That's it.

CoRTOS knows about the tasks from the following lines in `CoRTOStask.h` and `CoRTOStask.c` (or other files as you deem appropriate):

```
#define number_of_tasks 3
typedef void (* task_address_type) (void);
task_address_type start_addresses [number_of_tasks] = {task0, task1, task2};
uint16_t task_stack_size [number_of_tasks] = {60, 60, 60};
```

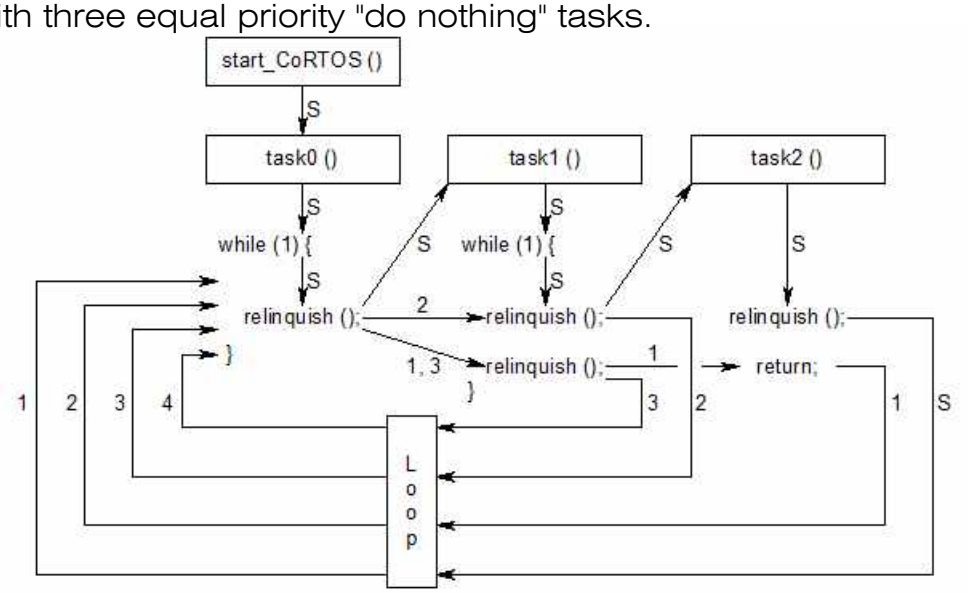
Tasks are void functions taking no parameters.

To illustrate the control flow, and the action of `relinquish()`, here is a simple round-robbin system with three equal priority "do nothing" tasks.

```
void task0 (void) {
   while (true)
      relinquish ();
}

void task1 (void) {
   while (true) {
      relinquish ();
      relinquish ();
   }
}

void task2 (void) {
   relinquish ();
}
```



In the following rather excruciatingly pedantic explanation of the program flow in the above example please refer to the line numbers in the kernel listing opposite and the paths in the diagram above.

Path S

`start_CoRTOS()` is called by `main()`.

40-45 Stack space for the tasks is carved from the stack allocated by the linker - be sure to reserve enough stack space. `start_CoRTOS()` sets the tasks to active (more about that later) and flags relinquish to start the tasks from their entry point. Tasks always start from the function entry point at startup.

Tasks are identified to CoRTOS by task numbers. The numbers are related to the tasks by their position in the `start_addresses` array. The much used variable `current_task` always refers to the task number of the executing task.

46-48 `task0()` is started from the beginning by `start_CoRTOS()` to get things rolling and so begins `Path S` in the flow chart above.

`task0`, is a very simple task that sits in an infinite loop relinquishing control to the next task. It is, in a sense, a prototype for all tasks - after some initialization code

the tasks normally sit in their own little "big loops," giving up control to the next task when they are blocked (i.e., waiting for the real world to catch up). Following `Path S`, `task0` calls `relinquish()`, and it is here that the CoRTOS kernel first starts.

10-12 `relinquish()` first saves the registers that the C compiler expects to be conserved across function calls and remembers `task0()`'s stack pointer. The assembler statement's clobber list informs the compiler of the registers that need to be saved across the upcoming context switch. The compiler inserts code to restore all the registers when `relinquish()` executes a return.

14-16 `current_task` is incremented, skipping any inactive tasks (in this case all tasks are active) and `task1()` becomes the `current_task`. The if() construct is faster and generates less code than the mod % operator.

17-20 `Path S` continues. The `start_from_beginning` flag was set for `task1()` when the system started, it is promptly cleared as a task is, normally, only started once from its entry point. The stack pointer is set to point to `task1()`'s stack and `task1()` is called via the `start_addresses[]` array.

`task1()` enters its own infinite loop and makes its first call to `relinquish()`.
`task2()` acquires initial control by the same path as `task1()`, above.

`task2()` calls `relinquish()` and path `S` ends and `Path 1` begins on line 16 when `current_task` is set back to 0 / `task0()`.

Path 1

17 & 25-26 `task0()` is given back control where it left off by switching to its saved task pointer, popping off all the saved registers, and returning to `task0()` - at the instruction right after `task0()` called `relinquish()`.

`task0()` loops around and calls `relinquish()` again.

`relinquish()` now saves `task0()`'s stack pointer, restores `task1()`'s stack pointer and gives control to `task1()`.

`task1()` makes its second call to `relinquish()`, passing control to `task2()`.

`task2()` is a once-through task, and when control is given to `task2()` it executes a `return` instruction.

21-22 `task2()` returns to here, the place right after `task2()` was started with a call to its entry point. `task2()` is marked as inactive so that it is skipped at line 17. `task2()` is also marked to start again at its entry point if it is ever made active again.

13 The reason for the `while(true)` loop becomes apparent as the code now loops around to advance current_task, in this case it goes back to `task0()`, beginning `Path 2`.

Path 2

Things go as before, `task0()` relinquishing control to `task1()`, which reaches the bottom of its `while(true)` loop and repeats its first call to `relinquish()`.

15-16 `current_task` advances to `task2()`, `task2()` is found to be inactive and is skipped, and as a result `current_task` is reset to 0 and `task0()`.

Paths 3 - ∞

`task0()` relinquishes control to `task1()` which alternates between its two calls to `relinquish()`, skipping `task2()` and giving control back to `task0()`.

Exiting CoRTOS

A return executed by `task0()`, which doesn't happen in this example, will stop CoRTOS. The return will take the code to line 49, and the subsequent return from `start_CoRTOS` will take the code back to `main()`.

Suspend / Resume & Blocking

The example above, where the tasks are always active, is not encountered in real systems. Instead, tasks go dormant while they are waiting for an event: a communications task waiting for an incoming message; a measurement task waiting for an A/D cycle to complete; or a robotics task waiting for a motor to reach a target position.

When a task is waiting for something to happen it is said to be blocked and it can't proceed until that something happens. Blocking, and all of the ways a task can block, is a major facet of all real time systems. Indeed, a task will spend most of its time blocked.

There are two techniques for blocking in CoRTOS:

- Directly sensed blocking The task sits in a `while()` loop, relinquishing control to the next task if it senses the block is still present. The task continues on its way when it senses the block has been removed.

- Suspended blocking The task calls `suspend()` (lines 30-32 in the kernel listing on page 6) to set itself to an inactive state and then calls `relinquish()`. This allows `relinquish()` to quickly skip over the blocked task. An ISR or another task is responsible for removing the block by calling `resume_task()` (lines 33-35). Obviously the blocked task first needs to tell its unblocker that it is waiting.

Directly sensed blocking is the simplest form:

```
while ((PORTB & bit2) == 0)
    relinquish ();
```

It is a primitive technique that will make sophisticates gag, but it has its place in small simple systems.

The drawbacks to direct blocking are:

- It is not suited to ultra-low-power (ULP) applications because the processor is always active even though it is doing nothing productive;
- It slows the software response time when the blocking condition clears - all tasks are normally blocked and so the added worst case response time is the sum of the time all the tasks take to test their current blocking condition.

The advantages to direct blocking are:

- Any condition can be sensed easily;
- Less code to write as there is no task<->ISR or task<->task coordination;
- Simplicity equates to code reliability.

Suspended blocking addresses the drawbacks of directly sensed blocking.

It is a bit more complicated as two independent bodies of code are involved:

```
void task0 (void)
{
  ...
  suspend ();
  task_waiting_for_PortB_2 = current_task;
  relinquish ();
  /* Block has just been removed by ISR,
  continue */
  ...
}
```

```
ISR (PortB_2_vector)
{
  /* Invoked when Port-B/Bit-2 goes high */
  ...
  if (task_waiting_for_PortB_2 != 0xff)
  {
    resume_task (task_waiting_for_PortB_2);
    task_waiting_for_PortB2 = 0xff;
  }
  ...
}
```

Note that the task calls `suspend()` before indicating it is waiting. If done in the reverse order there is the possibility the ISR is triggered in the interval in the task code between setting the flag and calling `suspend()` - this would cause the ISR to see the flag set and so resume the task (with no effect) - and then the task suspends itself, with the result that no resumption of the task ever comes and the task sits suspended forever. Subtleties like these are a given when interrupts get involved.

The `suspend()/resume_task()` interaction can also be task->task. This is common when one task sends a message to another task asking for some action (moving a robotic arm as an example) or informing on status (motion completed).

The advantages of suspended blocking are:

- Suitable for ULP: `relinquish()` can sense that all tasks are inactive and place the processor in a low power state. This topic is covered in Appendix A;

- Response time is at a minimum: `relinquish()` is able to go through the tasks very quickly to find an active task and give it control. Time is not wasted continuously checking blocking conditions.
- Allows prioritization: as described in Appendix A.

The disadvantage of suspended blocking is that it complicates the system:

- ISRs need to be written that sense block removal and resume the blocked tasks;
- Software reliability is lowered as more code, more complicated code, leads to the inclusion of the hard-to-root-out subtle bugs associated with real time operating system code, especially interrupts that very occasionally come at just the wrong time between two machine instructions.

CoRTOS provides standard functions, all using suspended blocking, that take care of most blocking situations, thus minimizing interrupt timing hazards.

Anatomy of a Task

If only the `relinquish()` function and direct blocking are used:

```
void taskx (void)
{
    /*  Do any initialization */
    ...
    /*  In general, each task is a forever loop. */
    while (true)
    {
        /*  Preform work until blocked by either lack of input
            or the requirement to wait for some time.  When we
            can't execute we pass control to the next task. */
        while (something == not_ready)
            /*  Give up control to the next task and let it see
                if it can do some work. */
            relinquish ();
        /*  We continue when that 'something' becomes ready. */
        ...
        /*  There can be many different places where we pass on
            control */
        while (something_else == has_not_happened_yet)
            relinquish ();
        ...
        /*  And if we are doing something long and tedious we
            might want to relinquish control at suitable points
            so the other tasks can get a turn. */
        b = -((2.0*pi*h)/lambda0))*sqrt((n*sin(theta1))^2.0-n2^2.0));
        relinquish ();
        r = (r12+r23*exp(-2.0*b))/(1.0+r12*r23*exp(-2.0*b);
        ...
        /*  If it is time to stop we execute a return, For task0
            executing the return stops the system and returns control
            to main () */
        if (we_are_finished == true)
            return;
    }
}
```

That all looks a bit tedious, so CoRTOS provides extensions in its API that make task design a more civilized affair.

A typical task using CoRTOS's extensions might look like:

```
void greetings (void)
{
    acquire_semaphore (human_machine_interface);
    send_message (printer_driver, "Hello World!\n");
    send_message (display_driver, "Hit any key to continue...");
    wait_for_message (keyboard_message_queue);
    send_message (display_driver, "\x1b[2K");
    release_semaphore (human_machine_interface);
}
```

if `greetings()` did not acquire exclusive control of the human/machine interface then several tasks might overwrite each other's messages.

Simple Examples: Blinky & Co.

Blinky

Blinky demonstrates the delay feature in CoRTOS.

Like all blinky programs it doesn't do much - but it does demonstrate a complete application of CoRTOS.

The files CoRTOSblinky???.c contain the tasks, the kernel, the interrupt code and the delay code all concatenated into one - needing only the processor specific header files that are supplied by the development environment.

Versions in the distribution are provided for AVR, MSP430 and PIC24. These are the only files to include in the project (or include in the directory for MSP/TI CCS):

Processor Family	File	Tested on Platform
AVR	CoRTOSblinkyAVR.c	ATmega328PB-XMINI, STK500
MSP	CoRTOSBlinkyMPS.c	MSP-EXP430FR6989
PIC	CoRTOSblinkyPIC.c	DM240004 - PIC24FJ128GA204
	CoRTOSuP.h	* Edit to identify the right micro
	CoRTOScomdefs.h	The usual #defines...

```
 1 void LED_on_task (void) {
 2    while (true) {
 3       /* Light, followed by one second of come what may. */
 4       led_on (0);
 5       delay (0, 100);
 6    }
 7 }

 8 void LED_off_task (void) {
 9    while (true) {
10       /* Wait 0.5 seconds with the LED on. */
11       delay (50)
12       /* and 0.5 seconds with the LED off. */
13       led_off (0);
14       delay (1, 50);
15    }
16 }
```

The two tasks use the `delay()` function in CoRTOS, which is comprised of 3 parts: a periodic interrupt service routine (ISR) [not shown]; a `service_timers()` routine called by the ISR, shown below; and the `delay()` function called by the task.

The delay code below is simplified, but shows how CoRTOS extensions are implemented.

```
 1 static uint16_t timer [number_of_timers];
 2 static boolean timer_active [number_of_timers] = {false, false, false};
 3 static uint8_t delayed_task_number [number_of_timers];

 4 /* service_timers() is called by a periodic interrupt service routine.
 5    The period determines the time value of a 'tick'. */
 6 void service_timers (void) {
 7    uint8_t tin;
 8    for (tin = 0; tin < number_of_timers; tin++) {
 9       if (timer_active[tin] == true) {
10          if (timer[tin] == 0) {
11             resume_task (delayed_task_number[tin]);
12             timer_active[tin] = false;
13          }
14          else
15             --timer[tin];
16       }
17    }
18 }

19 void delay (uint8_t tin, uint16_t ticks) {
20    timer_active[tin] = false;
21    timer[tin] = ticks;
22    delayed_task_number[tin] = current_task;
23    suspend ();
24    timer_active[tin] = true;
25    relinquish ();
26 }
```

The task side of `delay()`:

20-22 The task first makes the timer inactive so that the ISR won't be decrementing the timer while its value is being set. It then stores the number of ticks for the delay and the task number of the delayed task, using the convenient `current_task` global variable maintained by `relinquish()`.

23 The task first calls `suspend()` to set its inactive flag - this eliminates the possibility of the ISR resuming the task and resetting the flag prematurely.

24-25 The timer is made active; its value will be decremented by `service_timers()`. The task calls `relinquish()` to make itself inactive until the delay time expires.

The ISR side:

8-9 The ISR scans all the timers looking for any active ones.

10-12 When the ISR finds an active timer it decrements it. If the timer goes to zero then the ISR resumes the delayed task and sets the timer back to inactive.

The whole of a delay feature for a cooperative RTOS is implemented in 26 lines of executable code.

Super Blinky

Super Blinky uses most of the features of CoRTOS to blink a set of up to 8 LEDs. Super blinky accommodates AVR, PIC24, MSP430 and SAMD10/Cortex M0+ processors. The PIC24 and MSP versions have to make do with only a few LEDs. The AVR version uses Port D to flash all 8, using an Atmel STK500 or an Explained Mini evaluation module with an Arduino-style LED indicator shield. The Cortex version flashes 16 LEDs on an LED shield.

Rather than being a stripped down all-in-one file application, SuperBlinky links with the same CoRTOS files that you would use in your application.

The distribution files CoRTOSsuperblinky.c holds `main()` and the task code. The distribution file CoRTOStask.h is customized to define the SuperBlinky tasks to CoRTOS, though in this case the stacks reside in CoRTOSsuperblinky.c; you would edit this file to define your tasks when configuring your own CoRTOS application. CoRTOStask.c is a placeholder file that you would use a prototype for your tasks.

The necessary files for the SuperBlinky demonstration are:

CoRTOSsuperblinky.c	CoRTOStask.h
CoRTOSkernel.c	CoRTOSkernel.h
CoRTOSioint.c	CoRTOSioint.h
CoRTOStimer.c	CoRTOStimer.h
CoRTOSsignal.c	CoRTOSsignal.h
CoRTOSmessage.c	CoRTOSmessage.h

Needless to say, Super Blinky is more complex than the average blinky program.

- One task is in control of the LEDs;
- The other tasks send messages to it to toggle the LEDs on and off;
- One task sets up a periodic signal for its blinking delay;
- The other tasks use the delay function as in Blinky.

```
 1 uint8_t led [3] = {0, 1, 2};
 2
 3 void task0 (void) {
 4    start_periodic_signal (0, 0, 43);
 5    while (true) {
 6       led[0] += 3;
 7       send_message (0, (void *)&led[0]);
 8       wait_for_signal (0);
 9    }
10 }

11 void task1 (void) {
12    while (true) {
13       led[1] += 5;
14       send_message (0, (void *)&led[1]);
15       delay (1, 47);
16    }
17 }
```

```
18 void task2 (void) {
19    while (true) {
20       led[2] += 7;
21       send_message (0, (void *)&led[2]);
22       delay (2, 51);
23    }
24 }

25 void task3 (void) {
26    uint8_t * ptr;
27    while (true) {
28       ptr = (uint8_t *)wait_for_message (0);
29       led_toggle (*ptr);
30    }
31 }
```

The line-by-line commentary:

4 `task0()` sets up a periodic signal. It uses periodic timer #0 and signal counter #0 with a time between signals of 47 ticks. All the tick values and the LED increments are prime numbers, resulting in a long repeat pattern.

6-7 The number of the LED to be toggled on/off is incremented and a message is sent using que #0.

8 `task0()` then waits for the periodic signal, sent every 47 ticks.

11-24 `task1()` & `task2()` do likewise, but they call `delay()` every time through their loops.

28-29 `task3()` waits for a message from any of the three tasks using message queue #0 and calls the LED toggle routine in the I/O module.

Signaling and messaging are covered in the CoRTOS API, below.

The CoRTOS API

Common parameters in the API functions are:

`tn`	task number
`tin`	timer number
`ticks`	number of timer ISR periods defining the time interval
`sin`	signal number
`mqn`	message queue number
`rn`	semaphore number
`semn`	semaphore number

Kernel - CoRTOSkernel.c

```
void start_CoRTOS (void);
void relinquish (void);
void suspend (void);
void resume_task (uint8_t tn);
void restart_task (uint8_t tn);
void reset_task (uint8_t tn);
```

Timer - CoRTOStimer.c

```
void initialize_timer_module (void);
void service_timers (void);
void delay (uint8_t tin, uint16_t ticks);
void start_periodic_signal (uint8_t tin, uint8_t sin, uint8_t ticks);
void stop_periodic_signal (uint8_t tin);
void start_countdown_timer (uint8_t tin, uint16_t ticks);
void start_timeout (uint8_t tin, uint16_t ticks);
uint16_t stop_timeout (uint8_t tin);
uint16_t check_timer (uint8_t tin);
void pause_timer (uint8_t tin);
void resume_timer (uint8_t tin);
```

Signals - CoRTOSsignal.c

```
void initialize_signal_module (void);
status_t send_signal (uint8_t sin);
uint8_t wait_for_signal (uint8_t sin);
void clear_signals (uint8_t sin);
uint8_t check_signals (uint8_t sin);
```

Messages - CoRTOSmessage.c

```
void initialize_message_module (void);
status_t send_message (uint8_t mqn, void * message);
void * wait_for_message (uint8_t mqn);
uint8_t check_messages (uint8_t mqn);
```

Semaphores - CoRTOSsem.c

```
status_t acquire_sem (uint8_t semn, boolean wait);
void release_sem (uint8_t semn);
void set_sem_count (uint8_t semn, uint8_t count);
uint8_t query_sem_count (uint8_t semn);
```

Kernel - CoRTOSkernel.c

```
void start_CoRTOS (void);
```

Called from `main()`. `main()` is the user's responsibility.

Before start_CoRTOS is called the `start_address[]` and `task_stack_size[]` arrays must be initialized by the user before `start_CoRTOS()` is called. Task numbers correspond to the order of the tasks in the array.

`start_CoRTOS()` initializes CoRTOS's variables and carves off stack space for each of the tasks. Stack is allocated by growing the stack from the current stack pointer value.

It then passes control to the task associated with task number 0x00 - '`task0`' for want of a better name. CoRTOS starts when `task0` calls `relinquish()`.

If `task0` executes a return then the system will stop and control will pass back to `start_CoRTOS()` and thence back to `main()`.

```
void relinquish (void);
```

Called from a task when it is blocked, `relinquish()` gives control to the next active task, as ordered by task number.

`relinquish()` can be modified to prioritize tasks, see Appendix A. If prioritized then `relinquish()` will give control to the active task with the lowest task number.

If a task is involved in long onerous calculations lasting for more than a few milliseconds there should be a sprinkling of `relinquish()` calls in the code in order to prevent the one calculation task from hogging the processor.

The following two functions are used by CoRTOS extensions to implement suspended blocking.

```
void suspend (void);
```

If a task suspends itself then it will be flagged as inactive and it will be skipped over by `relinquish()`. The software that detects the clearing of the block would resume the task.

```
void resume_task (uint8_t tn);
```

A task may be resumed by an ISR. There is no restriction on the priority or number of ISRs that make calls to `resume()`. Calling `resume_task()` has no effect if the task is not suspended. These functions should be used if the

extension set is expanded. These two functions are required to be used for blocking if the tasks are prioritized, see Appendix A.

The following two functions implement brute-force task control and are useful when a machine function needs to be restarted. The functions need to be used with care: If task `tn` is reset/restarted when it was queued to a semaphore then the semaphore will stall when `tn`'s turn comes up for ownership; If called when tn is delayed then the task will be resumed when the delay is up, causing a possible malfunction; If the task was waiting for a signal then extra signals may accumulate to the task, ditto for messages.

```
void restart_task (uint8_t tn);
```

This function sets the start_from_begining flag for the task and makes it active. As a result the task will execute from its entry point when its turn comes up in the task rotation.

```
void reset_task (uint8_t tn);
```

This function sets the `start_from_begining` flag for the task and marks it as suspended/inactive. The task will no longer execute. If `tn` is subsequently made active by a call to `resume_task()` then it will execute from its entry point when its turn comes up in the task rotation.

Delays & Timers - CoRTOStimer.c

The number of timers is variable and is defined in CoRTOStask.h. A safe strategy, if RAM is not at a premium, is to have one timer per task and use the task number for the `tin` (timer number) parameter. The number of timers is defined in CoRTOStask.h.

`ticks` is used throughout and is in units of time, defined by the rate of a periodic interrupt routine. Generally a tick period of 2mSec to 10mSec has been found to be near optimum for most systems. At 10mSec/tick the maximum timer value is `0xffff` = 655.35 seconds. The actual delay can vary by -0/+1 tick. `delay(0)` results in a delay of less than one tick.

If periodic signaling is not used then the function `service_timers()` in the file CoRTOStimer.c must be edited as noted in the code comments. This removes the call to the periodic signaling software. If this is not done a linker error will result.

The timer module contains several functional sections:

- Task delays;
- General purpose count-down timers to measure how much time has passed;
- Timeouts used in conjunction with waiting for signals, messages or semaphores;

- Periodic signals that can be used to run tasks at regular intervals.

```
void initialize_timer_module (void);
```

If the timer module is used, then `initialize_timer_module()` must be called from `main()` before CoRTOS is started.

The user must supply a periodic ISR as a time base:

```
void service_timers (void);
```

If the timer module is used, then a periodic ISR must call the function `service_timers()`.

The rate at which `service_timers()` is called determines the value of a tick, not the ISR's execution rate per se.

Task Delays

```
void delay (uint8_t tin, uint16_t ticks);
```

Causes the current task to be suspended for the specified time.

Periodic Signals

If periodic signaling is used, then the signaling module CoRTOSsignal.c must be included in the system.

```
void start_periodic_signal (uint8_t tin, uint8_t sin,
   uint8_t ticks);
```

Causes a signal every `ticks` units of time - if `service_timers()` is called every 10mSec and `ticks` is 20 then the task will receive a signal every 200mSec.

The signal will be sent via signal `sin`. If the task does not pick up the signals, they will accumulate to a value of 0xff. See the section on signaling, below.

```
void stop_periodic_signal (uint8_t tin);
```

Stops the signal. It does not clear accumulated signals - `clear_signals()` can be used for that purpose (see the signaling module).

General purpose count-down timers

```
void start_countdown_timer (uint8_t tin, uint16_t ticks);
```

Starts timer `tin` with the value `ticks` and immediately returns. The timer counts down to zero and stops. Its value can be read with `check_timer()`.

```
uint16_t check_timer (uint8_t tin);
```

Returns the number of remaining ticks in timer `tin`. It can be used with any timer be it count-down, periodic signaling or delaying.

Timeouts used with `wait_for_xxx()`

Timeouts can be used in conjunction with waiting for signals, messages and semaphores. A timeout is first started and the task then makes the call to wait for the signal or message. When the task is made active, it first stops the timeout and then checks if the wait for a signal, message or semaphore was successful.

It is very important to call `stop_timeout()` after successfully acquiring a message, signal or semaphore, otherwise a spurious `resume()` call may be made by `service_timers()`.

```
void start_timeout (uint8_t tin, uint16_t ticks);
```

Starts a timeout timer. If the timer expires then the calling task (i.e., `current_task`) will be made active by the timer ISR/`service_timers()` with a call to `resume_task()`.

```
uint16_t stop_timeout (uint8_t tin);
```

Stops the timeout timer. The function returns the number of ticks remaining in the timeout.

An example of using a timeout:

```
char * msg_ptr;
...
/* Wait for a message with a 1 second timeout. */
start_timeout (my_timeout_timer, 100);
msg_ptr = wait_for_message (my_message_que);
stop_timeout (my_timeout_timer);
if (msg_ptr == NULL) {
    /* timeout, no message */
    ...
}
```

Miscellaneous timer functions:

```
uint16_t check_timer (uint8_t tin);
```

Returns the number of ticks remaining in timer `tin`. `0x0000` is returned if the timer has expired.

```
void pause_timer (uint8_t tin);
void resume_timer (uint8_t tin);
```

These functions can be used with any active timer. If the timer is inactive, they have no effect.

Signals - CoRTOSsignal.c

Signals carry no extra information. They accumulate in 8-bit counters. They are used for isr->task and task->task signaling. If data is to accompany a signal then using `send_message()` may be a better alternative. Signaling is one-on-one; broadcast signals must use multiple signal counters, one for each receiving task. The number of signal counters is set it `CoRTOStask.h`.

Signaling, like the rest of CoRTOS, normally runs with interrupts enabled - however:

A caution for RISC processors: Most RISC processors cannot atomically manipulate memory. As a result, if two ISRs of different priorities, where one can interrupt the other, are both signaling with the same signal counter then a short disable/enable interrupts may need to be added to the function `send_signal()`, see the comments in the `CoRTOSsignal.c` file. This caution does not apply to CISC processors such as the MSP430, but does apply to the AVR, PIC24, and ARM families.

```
void initialize_signal_module (void);
```

Needs to be called from `main()` if the signaling module is used.

```
status_t send_signal (uint8_t sin);
```

This function is called from an ISR or task. The signal counter is incremented; if a task is waiting for the signal then the task is resumed. If a task is not waiting then the signal will accumulate in the signal counter. A return of `status_busy` - `0x01` will be made if the signal counter is at its maximum of `0xff`, otherwise the return value will be `status_ok` - `0x00`. `status_t` values are defined in CoRTOScomdefs.h.

```
uint8_t wait_for_signal (uint8_t sin);
```

The calling task will wait until the signal counter is greater than zero. When the signal counter is positive then the signal counter will be decremented by one and the task will continue.

Normally signals are consumed at the same rate as they are generated. The function returns the number of received signals, normally this value would be 1, but will be greater if signals are being generated faster then they are being consumed.

If a timeout is started in conjunction with `wait_for_signal()` and the return value is 0 then the timeout has expired.

```
void clear_signals (uint8_t sin);
```

The signal counter is set to zero.

```
uint8_t check_signals (uint8_t sin);
```

Returns the value of the signal counter.

Messages - CoRTOSmessage.c

The user can specify the number of message queues and their length, the values are set in CoRTOStask.h. As with signaling, it is important the sender and receiver tasks agree on which queue they are using. If RAM is not at a premium, and there is a lot of messaging going on, then having one queue per task can make things simple - the `mqn` parameter is then the task number of the receiving task. Tasks may use as many queues as they please but can only wait on one queue at a time.

Messaging is not designed for ISR->task use. Use the signal module for this, using your own message ques. If a lot of ISR->task messaging is required in your application then you may want to rewrite the message module along the lines of the signal module.

```
void initialize_message_module (void);
```

Only needs to be called if the message module is used. It is called once from `main()` in the user code.

```
status_t send_message (uint8_t mqn, void * message);
```

Queues a message that can be retrieved by another task. If the message que is full then `status_full` - 0x02 is returned, otherwise the return value is `status_ok` - 0x00. `status_t` values are defined in CoRTOScomdefs.h.

```
void * wait_for_message (uint8_t mqn);
```

The calling task waits until a message is available. If a message is already in the queue then the task will pick it up immediately. The return value is the `message` pointer passed in `send_message()`.

If a timeout was started, and the return value is `NULL`, then the timeout has expired.

```
uint8_t check_messages (uint8_t mqn);
```

Returns the number of waiting messages.

Semaphores - CoRTOSsem.c

The word "semaphore" has several meanings. The first meaning is for a visual signaling system using waving flags to send text messages from one hilltop to another. The second meaning is a visual signaling system used by railroads to

insure only one train at a time had access to a shared section of track. Software semaphores conflate both meanings and they are used to both send signals and control access.

Semaphores were an early construct in the design of the first real time operating systems. Semaphores have since evolved and CoRTOS supports both message queues and signals. These other methods should be examined first to see if they are a better fit for the application than the rather primitive semaphores.

CoRTOS uses binary semaphores for access control to shared resources.

In this manual the word "semaphore" refers to the logical construct and not the signal (count) itself. Each semaphore has an initial count that can range from 0 to 255. A task normally requests one count from a given semaphore. There is nothing that precludes a task from making multiple requests to acquire more than one count from a semaphore. Multiple tasks can each acquire counts from the same semaphore. Additionally a task can acquire counts from multiple semaphores.

A task that requests a count from a semaphore that has a count of 0x00 will wait in a queue.

The initial count of a semaphore is 0x00.

Semaphore usage in CoRTOS can be classified into:

- Binary Semaphores are used to control access to a shared hardware or software element. They are also used to coordinate and synchronize tasks. At system start up, after the element is initialized and available, the count should be set to 0x01. After a task has acquired a count and gained access to the resource it needs to release the count back to the semaphore when the task has finished using the resource.

- Counting Semaphores are used to give multiple permissions to a collection of elements. Counting semaphores can also be used to synchronize multiple subordinate tasks to a master task. If used for access control then the count needs to be set after the elements are initialized and available. If used for access control the semaphore counts need to be released back to the semaphore. If used for synchronization then the counts are discarded (i.e. not returned) by the acquiring tasks.

- Signaling Semaphores are an obsolete use of semaphores to synchronize tasks. The initial count should be set to 0x00 (the default value). Signaling semaphores CAN NOT be used from an interrupt; use signals instead. Like counting semaphores the counts are discarded after they have been acquired.

Binary semaphores are primitive forms of "mutexes." Mutexes come with more features useful in controlling access to a shared resource. CoRTOS, being a basic system, does not offer the full suite of mutex features.

The semaphore functions are included by linking in the `CoRTOSsem.c` file.

The number of semaphores is set in `CoRTOStasks.h`. This value is used when CoRTOS is compiled. Semaphores are static and cannot be dynamically added or removed.

Semaphore structures are referred to CoRTOS by a semaphore number `semn`, starting with 0.

Semaphore count 'ownership' is not tracked by the system and the count acquired by a task is not returned to the semaphore if the task is initialized. It would be the job of the initializing task to return the acquired semaphore count.

Setting the Semaphore Count

At startup a semaphore's count is `0x00`. The count can be set to any number in the range `0x00-0xff` by a call to:

```
void set_sem_count (uint8_t semn, uint8_t count);
```

This function is used with counting and binary semaphores.

If the new count is >`0x00` and one or more tasks are queuing at the semaphore then the count value is adjusted and the queuing tasks are resumed.

- if used for access control then the count is set after the element has been initialized and is ready for use. Changing the value when tasks 'own' the element can be problematic.

- if used for synchronizing multiple tasks from a master task then the count is set when it is time to signal the subordinate tasks. Alternatively the master task can just release multiple counts one after the other.

Semaphore Acquisition, Release and Query

```
status_t acquire_sem (uint8_t semn, boolean wait);
```

If the semaphore's count is >`0x00` then the count is decremented by one and the task continues in the ready state.

If the count is `0x00` when acquisition is attempted then the task stops execution and enters a FIFO queue of tasks waiting for the semaphore count to be incremented by the release of a semaphore.

A timeout on acquisition can be set, see the previous Delays & Timers section.

The return value from `acquire_sem()` can be:

`case status_ok`
Semaphore count was available.
`case status_unavail`
The `wait` parameter is false and no semaphore count is available.
`case status_timeout`
The timeout has expired while waiting for a count

```
void release_sem (uint8_t semn);
```

If a task is queuing for a count from `semn` then that task is resumed. If no task is queuing then the semaphore's count is incremented.

```
uint8_t query_sem_count (uint8_t semn);
```

The count value of semaphore `semn` is returned.

Why Use Semaphores?

If signals and message queues are better at doing the specific jobs that semaphores are often forced to do then why bother with semaphores?

Using semaphores can reduce code size. As semaphores are multi-purpose objects a system with modest requirements for task<->task signaling and messaging can use only the semaphore feature and not need to include the signaling and messaging modules.

Multiple tasks can que at a semaphore for a signal. A signal can only have one task queuing at a time. Though the wisdom of having multiple tasks queuing for the same signal might be questionable.

Warnings When Using Semaphores as Mutexes

If tasks are prioritized, as described in Appendix A, there is the possibility of "priority inversion." See the web for a discussion of this hazard and mitigation strategies. In most cases a short period of inversion is not a problem.

As it is possible for a task to acquire more than one semaphore there is the possibility of a 'deadly embrace.' This happens when task1 acquires resource r1 and then waits to acquire r2, while task2 already owns r2 and then waits for r1to be released. It is very, very easy to fall into this trap. Approach multiple resource acquisition with trepidation. Always acquire multiple resources in the same order in each task if at all possible.

Interrupts and CoRTOS

CoRTOS is designed to execute with interrupts enabled.

The only exception is for 8 bit processors with 16 bit stack pointers where interrupts have to be disabled very briefly to insure the stack pointer value is set atomically.

There are functions for turning the timer interrupt on and off in the supplied `CoRTOSint.c` files. The timer interrupt can be turned on in `main()` only after the delay module has been initialized.

```
void timer_int_disable (void);
void timer_int_enable (boolean force);
```

The timer interrupt control functions have nesting. If a call to disable is made while the interrupts are already disabled then only the outermost corresponding enable call will have any effect. The interrupts can be forced to enabled and the nesting level reset by setting `force` to `true`.

Distribution Files

The wild card characters ??? indicate the processor, either "AVR" for ATmega 168/328, MSP for MSP430, PIC for PIC24 or M0 for ARM Cortex M0+. All files use 3 space hard tabs.

System files:

Required

`CoRTOSkernel.c` `CoRTOSkernel.h`	CoRTOS itself
`CoRTOSuP.h`	Defines the microprocessor. This is used for conditional compilation of the CoRTOS files. Feel free to instead just edit out the unneeded microprocessor specific code.
`CoRTOScomdefs.h`	Common definitions, included in all CoRTOS files. You may use it in your own files if you wish but it is not necessary.
`CoRTOStask.h`	Defines (as literals) the number and size of CoRTOS objects: the number of tasks, signals, timers, semaphores, message queues and message queue sizes. It also defines the arrays that hold the task start addresses and the stack sizes. It should be included in the file that declares these arrays.

CoRTOStask.c	Is a placeholder module for the task and stack size arrays. You do not have to use this module, but these arrays have to be declared somewhere in your code.

CoRTOS extensions

CoRTOStimer.c CoRTOStimer.h	Only needed for the delay & timer functions
CoRTOSmessage.c CoRTOSmessage.h	Only needed for passing messages
CoRTOSsem.c CoRTOSsem.h	Only needed for semaphores
CoRTOSsignal.c CoRTOSsignal.h	Only needed for task signaling

These demonstration files are also included:

CoRTOSblinkyAVR.c CoRTOSblinkyMSP.c CoRTOSblinkyPIC.c	All-in-one demonstration files, processor specific, CoRTOSuP.h needs to agree with the uP type
CoRTOSSuperBlinky.c	The SuperBlinky demonstration file
CoRTOSsemtest.c	A semaphore demonstration file
CoRTOSio.c CoRTOSio.h CoRTOSint.c CoRTOSint.h	I/O and interrupt modules to work with the SuperBlinky and semtest modules. You can use these as a template for your I/O and interrupt modules.

Appendix A - Adding Features

Adding Prioritization

A one-line change to **relinquish()** changes CoRTOS from round-robin scheduling to prioritized, though not preemptive, scheduling:

```
void relinquish (void) {
   asm volatile ("NOP":::"r2","r3","r4","r5","r6","r7","r8","r9","r10",\
      "r11","r12","r13","r14","r15","r16","r17", "r28", "r29");
   sp_save[current_task] = _SP;
   while (true) {

      /* The following line changes CoRTOS to prioritized scheduling */
      current_task = 0;

      do {
         if (++current_task == number_of_tasks) current_task = 0;
      } while (task_active[current_task] == false);
      if (start_from_beginning[current_task] == true) {
         start_from_beginning[current_task] = false;
         _SP = starting_stack[current_task];
         start_addresses[current_task] ();
         task_active[current_task] = false;
         start_from_beginning[current_task] = true;
      }
      else {
         _SP = sp_save[current_task];
         return;
      }
   }
}
```

As a result of this change, after a task relinquishes control the scan for the next available task always starts at task0 and continues up the task numbers.

task0 is the highest priority task; priorities decline with increasing task number.

This will only work with suspended blocking where tasks are inactive while they are blocked. A task using direct blocking with prioritization will not allow lower priority tasks to execute while it is waiting for the block to clear.

Using CoRTOS in Ultra Low Power Applications

Small systems often have to work at ultra low power levels and CoRTOS can accommodate this with the use of suspended blocking and a small change to CoRTOSkernel.c. The added code is shown in blue:

```
1  static uint8_t number_of_active_tasks = 0;

2  void relinquish (void) {
3     asm volatile ("NOP":::"r2","r3","r4","r5","r6","r7","r8","r9","r10",\
4        "r11","r12","r13","r14","r15","r16","r17", "r28", "r29");
5     sp_save[current_task] = _SP;
6     while (true) {
7        do {
8           if (++current_task == number_of_tasks) {
9              if (number_of_active_tasks == 0)
10                enter_low_power_state ();
11             current_task = 0;
12             number_of_active_tasks = 0;
13          }
14       } while (suspended[current_task] != false);
```

```
15       number_of_active_tasks++;
16       if (start_from_beginning[current_task] == true) {
17          start_from_beginning[current_task] = false;
18          _SP = starting_stack[current_task];
19          start_addresses[current_task] ();
20          suspended[current_task] = true;
21          start_from_beginning[current_task] = true;
22       }
23       else {
24          _SP = sp_save[current_task];
25          return;
26       }
27    }
28 }

29 void resume_task (uint8_t tin) {
30    suspended[tin] = false;
31    /* Generally not needed as ISR trigger exits low power state. */
32    exit_low_power_state ();
33 }
```

9 If relinquish() finds no ready to run tasks in the course of a scan then the processor can go into a low power state until there is an unblocking event. An ISR calling resume would take the processor out of the low power state.

10 `enter_low_power_state()` stops, or slows, the clocks in order to put the processor in a static or close to static state. An interrupt leading to the resumption/unblocking of a task would turn the processor back on and cause `enter_low_power_state()` to execute a return .

13 In any case `number_of_active_tasks` is reset at the beginning of each scan for tasks.

14 If a task is found that is ready to run then the code will pass this point and the `number_of_active_tasks` will be greater than 0.

Adding Task Prelude Functions

It can be useful to have the RTOS call a function every time a task gets control.

An example of this might be a system with several identical operator panels that each works independently - in this case it would save code, and make maintenance easier, if there were one copy of the task with multiple instances of the task in `task_addresses[]`. The prelude function would then make sure that I/O is redirected appropriately and also set up any variables for that particular virtual instantiation of the task.

The added code is shown in blue. Task names and such are just placeholders for the example.

```
1 /* Task addresses with two instances of the op_task for the operator panels. */
2 task_address_type start_addresses [number_of_tasks] = {
3    task0, op_task, op_task, task3};

4 typedef void (* prelude_function_type) (void);
5 prelude_function_type prelude_addresses [number_of_tasks] = {
6    prelude_null, prelude_op_task1, prelude_op_task2, prelude_null};

7 void relinquish (void) {
8    asm volatile ("NOP":::"r2","r3","r4","r5","r6","r7","r8","r9","r10",\
```

```
 9       "r11","r12","r13","r14","r15","r16","r17", "r28", "r29");
10    sp_save[current_task] = _SP;
11    while (true) {
12       do {
13          if (++current_task == number_of_tasks) {
14             current_task = 0;
15       } while (suspended[current_task] != false);
16       if (start_from_beginning[current_task] == true) {
17          start_from_beginning[current_task] = false;
18          _SP = starting_stack[current_task];
19          prelude_addresses[current_task] ();
20          start_addresses[current_task] ();
21          suspended[current_task] = true;
22          start_from_beginning[current_task] = true;
23       }
24       else {
25          _SP = sp_save[current_task];
26          prelude_addresses[current_task] ();
27          return;
28       }
29    }
```

Appendix B - Stacks and Stack Pointers

Extremely Small Systems - Saving stack space

Some RISC μPs have a large number of registers and the language specification expects them all to be preserved across a context switch or on the occurrence of an interrupt.

If, in a gross theoretical calculation, a compiler expects to have 16 registers pushed in a context switch and the same 16 pushed again in the case of an interrupt then it is possible that (16 + 16) * 2 = 64 bytes of stack needs to be set aside in each task in addition to the stack space needed for function calls and parameter passing.

Excessive stack requirements can be mitigated by having a separate system stack, used to absorb stack growth due to interrupts that may happen inside `relinquish()`. The added code is shown in blue.

The downside is that interrupts are off while context is saved by pushing all the necessary registers.

```
 1  void relinquish (void) {
 2     disable ();
 3     asm volatile ("NOP":::"r2","r3","r4","r5","r6","r7","r8","r9","r10",\
 4        "r11","r12","r13","r14","r15","r16","r17", "r28", "r29");
 5     sp_save[current_task] = _SP;
 6     _SP = system_stack;
 7     enable ();
 8     while (true) {
 9        do {
10           if (++current_task == number_of_tasks)
11              current_task = 0;
12        } while (suspended[current_task] != false);
13        if (start_from_beginning[current_task] == true) {
14           start_from_beginning[current_task] = false;
15           _SP = starting_stack[current_task];
16           start_addresses[current_task] ();
17           suspended[current_task] = true;
18           start_from_beginning[current_task] = true;
19        }
20        else {
21           disable ();
```

```
22          _SP = sp_save[current_task];
23          enable ();
24          return;
25       }
26    }
27 }
```

```
 1 void start_CoRTOS (void)
 2 {
 3    uint8_t tn;
 4    uint16_t spv;

 5    system_stack = _SP;
 6    spv = system_stack - system_stack_size;
 7    for (tn = 0; tn < number_of_tasks; tn++) {
 8       starting_stack[tn] = spv;
 9       spv -= task_stack_size[tn];
10       start_from_beginning[tn] = true;
11       suspended[tn] = false;
12    }
13    start_from_beginning[0] = false;
14    current_task = 0;
15    start_addresses[0] ();
   }
```

Atomic Stack Pointer Operations

Processors with byte wide registers and two byte wide stack pointers, such as the AVR, need to have interrupts disabled when the stack pointer register is changed.

Reading the stack pointer is safe - the stack pointer value before and after an interrupt is always the same.

The kernel code for the AVR in CoRTOSkernelAVR.c already has the required changes, shown here in blue.

```
 1 /* Code for the Atmel AVR ATMega processor family. */

 2 void relinquish (void) {
 3    asm volatile ("NOP":::"r2","r3","r4","r5","r6","r7","r8","r9","r10",\
 4       "r11","r12","r13","r14","r15","r16","r17", "r28", "r29");
 5    sp_save[current_task] = SP;
 6    while (true) {
 7       do {
 8          if (++current_task == number_of_tasks)
 9             current_task = 0;
10       } while (suspended[current_task] != false);
11       if (start_from_beginning[current_task] == true) {
12          start_from_beginning[current_task] = false ;
13          disable ();
14          SP = starting_stack[current_task];
15          enable ();
16          start_addresses[current_task] ();
17          suspended[current_task] = true;
18          start_from_beginning[current_task] = true;
19       }
20       else {
21          disable ();
22          SP = sp_save[current_task];
23          enable ();
24          return;
25       }
26    }
27 }
```

Appendix C - Processor Customization

CoRTOS as supplied is setup for the AVR ATMega168/328/1284, Texas Instruments MSP430/FR5994/FR5969/FR6989, Microchip PIC24FJ128GA204 and Atmel SAMD10/Cortex M0+. Changes may be needed to the supplied code when using other versions of these µPs with differing numbers of timers and different I/O assignments.

CoRTOS is much easier to port to other µPs than would be the case with a preemptive OS. In a preemptive system the context is switched inside an ISR and is complicated and tricky, especially, it seems, in µPs 'designed' for use with an RTOS like the ARM family.

When ported to different µPs the changes will include:

When using CoRTOS with another processor:

Files to be changed	Changed element
CoRTOSuP.h	#define the microprocessor
CoRTOSkernel.c	Stack pointer access, stack growth direction and context save with either a clobber list or inline asm push/pop instructions
CoRTOStask.c	Stack size, especially with 32bit processors
CoRTOSint.c/h CoRTOSio.c/h	Only used with the SuperBlinky demo. These files can be used as a template for your own timer interrupt, enable and disable interrupts, I/O and timer initialization. CoRTOSio.c will need to be changed to properly map logical to physical LEDs
CoRTOScomdefs.h	Typedef collision resolution -- the typedefs that come with the µP's system include files might collide with this file. CoRTOS only uses unsigned chars and ints and changes should be easy.

Appendix D: Development Environment

CoRTOS was created in a Windows 7 environment using the following tools:

AVR
- AVR Studio 7
- GCC compiler
- ATmega328PB-XMINI, MEGA-1284P & STK500 development boards
- UM-UNO design LED IO status indicator shield (ebay)

MSP430
- TI Code Composer Studio V7
- TI Compiler
- TI MSP-EXP430FR5994/5969/6989 Launch Pad boards

PIC24
- MPLAB X
- Microchip X16 compiler
- Microchip DM24004 Curiosity board w/ PIC24FJ128GA204

Cortex M0+ / Atmel ATSAMD10D14AM
- AVR Studio 7
- GCC compiler
- ATSAMD10-XMINI development board
- UM-UNO design LED IO status indicator shield (ebay)

Appendix E - Common Problems

If the code goes awry and gets very lost for no discernable reason the cause is almost certain to be stack collision. Try doubling the size of each task's stack and see if the problem goes away. If stacks are colliding with variables then the stack size in the linker needs to be increased. Needless to say CoRTOS, like any other OS, uses far more stack space than any compiler would calculate as sufficient. Stack overflow checking may need to be disabled or modified

If relinquish() is forever stuck trying to find an active task then there may be a problem with interrupts. Whatever it is that is supposed to service the timers or signal a task isn't working. Set a breakpoint in the ISR to confirm the interrupt is getting triggered.

Be sure the number of tasks/timer/sems/etc. in `CoRTOStask.h` is correct and that the stack size and task address arrays have the right number of entries.

REVISION LOG

1.10	12Dec2018	Fix bug that didn't save R17 when used with AVR processors. Added semaphores, removed resources. Expanded test suite. Better support for boards with differing #s of LEDs. Miscellaneous other changes.
1d01	18May2018	Changes for MSP processors. Clarification on files.
1d	15May2018	Consolidated the processor specific files using conditional compilation. Add support for Cortex M0+. Use clobber lists where possible. Revise manual as needed.
1c	19Jan2018	Delay tolerance changed to -0/+1; change "`task task0 ...`" to "`void task0`"; general clean up.
1b	23May2017	Add comment on the need for interrupt enable/disable in CoRTOSsignal.c for RISC processors under some conditions
1a		Remove need for interrupt disable in CORTOSsignal.c