

CoRTOS - The World's Simplest RTOS

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

Introduction & Overview

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

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

- Task delays
- Periodic tasks
- Timers & timeouts
- Mutexes
- Signals
- Messages

System responsiveness is maximized with an interrupts always enabled design. Nested interrupts can make signaling calls to the OS.

CoRTOS is intended for smaller microprocessors, such as the MSP430, PIC24 and AVR, and for 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

CoRTOS and its documentation are released under the
GNU General Public License, Version 3, available at
<https://www.gnu.org/licenses/gpl.txt>

There are no guarantees, warranties or claims
concerning proper operation of this software.

No liability of any kind is assumed by Nicholas O. Lindan
and Cleveland Engineering Design, LLC.

"Worth price charged"

Nicholas O. Lindan
Cleveland Engineering Design, LLC
1412 Dorsh Road, Cleveland Ohio 44121 USA
nolindan@ix.netcom.com - 216-691-3980
Copyright 2017, GPL 3.0
CoRTOS Manual v1a.pdf - 12 March, 2017

Contents

Introduction and Overview.	1
The Basics of RTOS.	4
The CoRTOS Kernel and Basic Task Switching.	6
Blocking.	9
Anatomy of a Task.	11
Simple Examples: Blinky & Co..	12
The CoRTOS API.	16
The Kernel.	17
Timers.	18
Signals.	20
Messages.	22
Resources (mutexes).	23
Interrupts and CoRTOS.. . . .	24
Distribution Files.	25
Appendix A - Adding Features to the Kernel.	26
Prioritized tasks.	26
Ultra low power applications.	26
Prelude functions.	27
Appendix B - Stacks & Stack Pointers.	28
Appendix C - Processor Specific Code.	30
Appendix D - Common problems.	30

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:

Kernel code.	6
API Summary.	16
Appendix A - Adding features to the kernel.	26

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

The Basics of RTOS.	4
The CoRTOS Kernel & Basic Task Switching.	6
Anatomy of a Task.	11
Blinky & Co..	12

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 it's 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 each remember where they are and when blocked return to the big loop. As pop singers tell us, life is but a succession of circles within circles.
- Scheduler programs call functions/tasks. Tasks, like the functions they are, 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 quasi-tasks.

- Multi-tasking 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:

<u>Call to RTOS</u>	<u>Task waits for</u>
delay()	A specified time;
wait_for_message()	A message from another task, sent through the RTOS via send_message();
acquire_resource()	Sole ownership of a shared resource (printer, etc.), when finished with the resource the task calls release_resource() to allow another task ownership;
wait_for_signal()	A signal from an ISR or task via send_signal().

Multi-tasking 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 then it continues execution - setting up the next action for the real world to take. Tasks can have equal priority and execute round-robin style or they can be prioritized. Cooperative systems are very fast. Although they impose some burden on each blocked task this per task overhead can be as little as 1 μ Second. 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. Small preemptive systems start at 4K 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 and are at the inner heart 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. A commented version of the kernel source code is provided in the distribution files.

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

5 uint8_t current_task;

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

10 void relinquish (void) {
11     pushall();
12     sp_save[current_task] = _SP;
13     while (1) {
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         }
24         else {
25             _SP = sp_save[current_task];
26             popall();
27             return;
28         }
29     }
30 }

31 void suspend (void) {
32     suspended[current_task] = true;
33 }

34 void resume_task (uint8_t tn) {
35     suspended[tn] = false;
36 }

37 void start_CoRTOS (void)
38 {
39     uint8_t tn;
40     uint16_t spv;

41     spv = _SP;
42     for (tn = 0; tn < number_of_tasks; tn++)
43     {
44         starting_stack[tn] = spv;
45         spv -= task_stack_size[tn];
46         start_from_beginning[tn] = true;
47         suspended[tn] = false;
48     }
49     start_from_beginning[0] = false;
50     current_task = 0;
51     start_addresses[0] ();
52 }
```

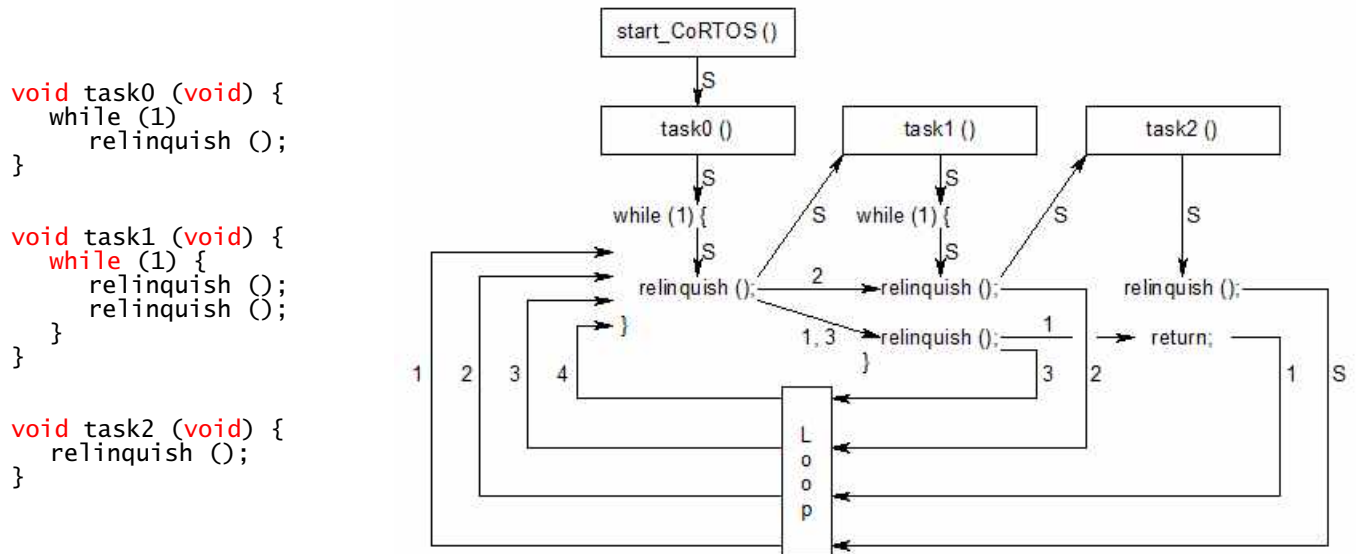
That's it. The entirety of the CoRTOS task management kernel is the function `relinquish()`, lines 10-30, implemented in 16 lines of executable C.

CoRTOS knows about the tasks from the following lines in CoRTOStask.h:

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

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.



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()`.

42-49 Stack space for the tasks is carved from the stack reserved by the compiler's startup code. `start_CoRTOS()` sets the tasks to active (more about that later) and flags `relinquish` to start the tasks from their entry point. In general, tasks only start from their 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.

50-52 `task0()` is started from the beginning by `start_CoRTOS()` to get things rolling and beginning 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 5, `task0` calls `relinquish()`, and it is here that the CoRTOS kernel first starts.

11-13 `relinquish()` saves the registers that the C compiler expects to be conserved across function calls and remembers `task0()`'s stack pointer. `pushall()` and `popall()` are assembler macros for saving and restoring registers defined in `CoRTOSasmmacros`.

14-17 `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.

18-21 Path 5 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 5 ends and Path 1 begins on line 16 when `current_task` is set back to 0 / `task0()`.

Path 1

18 & 26-28 `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 return address 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.

22-23 `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.

14 The reason for the `while(1)` 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(1)` loop and repeats its first call to `relinquish()`.

16-17 `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 53, and the subsequent return from `start_CoRTOS` will take the code back to `main()`.

Blocking

Blocking, and all of the ways a task can block, is a major facet of all real time systems.

A task is blocked when it is waiting for something to happen and can no longer continue executing. The blocked task then calls CoRTOS to relinquish execution and CoRTOS gives control to the next available task.

There are two techniques for blocking in CoRTOS:

- Directly sensed blocking The task sits in a `while()` loop, relinquishing control 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()` to set itself to inactive 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()`. Obviously the blocked task first needs to tell its unblocker that it is waiting.

Some common reason for a task block are that the task is waiting for:

- An input signal on one of the processor's input pins;
- The passage of time;
- A message from another task;

- A/D or PWM cycle completion;
- Reception of a communications message;
- Availability of a shared resource, such as an HMI (Human Machine Interface) terminal.

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 response time to the removal of a block - all tasks are normally blocked and so the added worst case response from this idle state is the sum of the time all the tasks take testing 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:

<pre>void task0 (void) { ... suspend (); task_waiting_for_PortB_2 = current_task; relinquish (); /* Block has just been removed by ISR, continue */ ... }</pre>	<pre>ISR (PortB_2_vect) { /* 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; } ... }</pre>
--	---

Note that the task calls `suspend()` before indicating it is waiting. If done in the reverse order there is the possibility the ISR sees the waiting task and resumes it (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 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 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 and 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 and 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:

```
task greetings (void)
{
    acquire_resource (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_resource (human_machine_interface);
}
```

if `greetings()` did not acquire exclusive control of the HMI then several tasks might pile their status messages on top of each other.

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:

Processor Family	File	Tested on Platform
AVR	CoRTOSblinkyAVR.c	ATmega328PB-XMINI, STK500
MPS430	CoRTOSBlinkyMPS.c	MSP-EXP430FR6989
PIC24	CoRTOSblinkyPIC.c	DM240004 - PIC24FJ128GA204

The two self-explanatory tasks in blinky are:

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

8 task LED_off_task ( void ) {
9     while (1) {
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 (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 timer service routine called by the ISR and the `delay()` function called by the tasks.

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_tasks; 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         }
15     }
16 }

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

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:

9-10 The ISR scans all the timers looking for any active ones.

11-13 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 the delay feature for a cooperative RTOS is implemented in a dozen lines of executable code.

Super Blinky

Super Blinky uses most of the features of CoRTOS to blink a set of up to 8 LEDs. There are versions in the distribution files for AVR, PIC24 and MSP processors. The PIC24 and MSP versions have to make do with only a few LEDs. The AVR versions uses Port D to flash all 8, using either an STK500 or an Explained Mini with an LED indicator sheild.

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

The distribution files CoRTOSsuperblinky???.c hold main() and the task code. The files CoRTOStask.h is customized to define the SuperBlinky tasks to CoRTOS; you would re-customize this file to define your tasks when making an application using the OS.

The necessary files for the SuperBlinky demonstration are:

CoRTOSsuperblinky???.c CoRTOStask.h

CoRTOS and extensions

Processor specific:

CoRTOSkernel???.c	CoRTOSkernel.h
CoRTOSioint???.c	CoRTOSioint.h

Generic:

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  static uint8_t led [3] = {0, 1, 2};
2
3  task task0 (void) {
4      start_periodic_signal (0, 0, 47);
5      while (true) {
6          led[0] = (led[0] + 3) & 0x07;
7          send_message (0, (void *)&led[0]);
8          wait_for_signal (0);
9      }
10 }

11 task task1 (void) {
12     while (true) {
13         led[1] = (led[1] + 5) & 0x07;
14         send_message (0, (void *)&led[1]);
15         delay (1, 61);
16     }
17 }

18 static task task2 (void) {
19     while (true) {
20         led[2] = (led[2] + 7) & 0x07;
21         send_message (0, (void *)&led[2]);
22         delay (2, 73);
23     }
24 }

25 static task 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 queue #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
mqn message queue number
sin signal number

Summary:

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, uint16_t ticks);
```

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);
```

Resources - CoRTOSresource.c

```
void initialize_resource_module (void);  
status_t acquire_resource (uint8_t rn, boolean wait);  
void release_resource (uint8_t rn);  
uint8_t query_resource_owner (uint8_t rn);
```

Interrupts - CoRTOSioint???.c

Interrupts are an important and needed part of most CoRTOS applications, but they not really part of the API.

Kernel - CoRTOSkernel???.c

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

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

The `start_address[]` array in `CoRTOStask.c` (or other application file) identifies the tasks and must be initialized by the user before `start_CoRTOS()` is called. It is best to simply initialize the entries with the task names in the definition statement. Task numbers correspond to the order of the tasks in the array.

`start_CoRTOS` first initializes the CoRTOS's variables.

It then passes control to the task associated with task number 0x00 - 'task0' for want of a better name. The 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 the extensions to CoRTOS to implement suspended blocking. They 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.

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

If a task suspends itself then it will be flagged as inactive and it will be skipped over by `relinquish()`. The software clearing the block would resume the task. Calling `resume_task()` has no effect if the task is not suspended.

The following are not used at present. If used, they need to be used with care: If task `tn` is reset/restarted when it was queued to a resource then the resource 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 possible malfunction; If the task was waiting for a signal then extra signals may accumulate to the task, ditto 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 `current_task` 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. A `ticks` value of `0x0000` will also result in a maximum delay. The actual delay can vary by -1/+0 ticks - `delay(1)` may be problematic.

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 resources;
- 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 time triggered ISR must call the function

`service_timers()`.

The rate of the call from the ISR 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 wait 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.

General purpose count-down timers

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

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

Timeouts used with wait_for_xxx()

Timeouts can be used in conjunction with waiting for signals, messages and resources. A timeout is first started and then the task waits 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 resource was successful.

It is very important to call `stop_timeout()` after successfully acquiring a message, signal or resource, 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 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_source);
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

These functions are meant for `isr->task` signaling, though there is nothing that precludes them from being used for `task->task` signaling. A signal carries with it no extra information. Signals accumulate in 8-bit signal counters. As distributed, the signal module declares 4 signal counters - this value can be changed as needed in `CoRTOStask.h`.

There is nothing that precludes this function from being used for `task->task` signaling.

Signaling is one-on-one. Broadcast signals must use multiple signal counters, one for each receiver.

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

Needs to be called from `main()` before 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, and 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` will be made if the signal counter is at its maximum of `0xff`, otherwise the return value will be `status_ok`. `status_xxx` 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 than they are being serviced.

If a timeout has been started in conjunction with the wait 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. As distributed there are 4 message queues of 8 messages each. These 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 at a time.

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

Only needs to be called if the message module is used. 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 queue is full then `status_full` - 0x02 is returned, otherwise the return value is `status_ok` - 0x00. `status_xxx` 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.

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.

Resources - CoRTOSresource.c

Resources are abstract entities that can be used for controlling access to shared physical resources. There are many other names for the same, or very similar, constructs. So many that CoRTOS adds another one to "lessen the confusion."

Only one task can have ownership of a resource at any one time. Any other task trying to acquire the resource will wait in a FIFO queue, or, optionally can choose to continue execution without acquisition.

When a task releases a resource then ownership transfers to the next queuing task. If there is no queuing task then the resource becomes freely available.

Resources are static entities created at compile time. The number of resources and their queue length is specified in the file `CoRTOStask.h`. In the distribution code there are 4 resources, each with a 4 task long waiting queue.

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.

WARNING: As it is possible for a task to acquire more than one resource 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 r1 to 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.

```
void initialize_resource_module (void);
```

Only needs to be called if the resource module is used. Called once from the user code before starting the system.

```
status_t acquire_resource (uint8_t rn, boolean wait);
```

The calling task attempts to acquire resource rn.

If the resource is free then task will be given ownership of the resource and the function will return immediately with a value of **status_ok**.

If **wait** is **true** and the resource is in use by another task then the task will enter the tail of a queue of tasks waiting to acquire the resource. When resource ownership is acquired then a return is made with a value of **status_ok**.

If **wait** is **false** and the resource is in use then an immediate return will be made with a value of **status_busy**.

In summary, the return values of **acquire_resource()**, defined in **CoRTOScomdefs.h**, are:

status_ok	(0x00)	The task has acquired the resource;
status_busy	(0x01)	The resource is in use and 'wait' was false;
status_full	(0x02)	The waiting queue is full;
status_timeout	(0x04)	A timeout was started and has expired, the task does not own the resource.

```
void release_resource (uint8_t rn);
```

The current task gives up ownership of the resource. If a task was queuing for the resource then it is given ownership. If no task was queuing then the resource is marked as free.

```
uint8_t query_resource_owner (uint8_t rn);
```

Returns the task number of the owner of the resource. If the resource is free a value of 0xff is returned.

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 should be turned on in `main()` 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, or PIC for PIC24. All files use 3 space tab characters.

System files:

Required

CoRTOScomdefs.h	common definitions, included in all CoRTOS files
CoRTOStask.c CoRTOStask.h	Define the tasks to the OS, need to be modified as tasks are added and removed from the system. These can be substituted with your own main.c module as long as <code>start_addresses[]</code> is defined somewhere.
CoRTOSkernel???.c CoRTOSkernel.h	CoRTOS itself
CoRTOSasmmacros???.h	Contains assembler macros for pushing/popping registers when tasks relinquish and gain control. For other processors, see the C-compiler documentation for your processor under "Assembly Language Calling Conventions," or some similar topic.
CoRTOSioint???.c CoRTOSioint.h	Required if the system uses interrupts for timers and signals. These files include I/O for the superblinky demo program. Expect to have to make minor changes for processor variations, especially with different members of the MSP430 family.

CoRTOS extensions

CoRTOStimer.c CoRTOStimer.h	Only need for the delay & timer functions
CoRTOSmessage.c CoRTOSmessage.h	Only needed for passing messages
CoRTOSresource.c CoRTOSresource.h	Only needed for resource management (mutex)
CoRTOSsignal.c CoRTOSsignal.h	Only needed for ISR -> task signaling

These demonstration files are also included:

Blinky example

CoRTOSblinky???.c
CoRTOSblinky???.h
CoRTOSsuperBlinky???.c
CoRTOSsuperBlinky???.h

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) {
    pushall();
    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];
            popall();
            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     pushall();
4     sp_save[current_task] = _SP;
5     while (1) {
6         do {
7             if (++current_task == number_of_tasks) {
8                 if (number_of_active_tasks == 0)
9                     enter_low_power_state ();
10                current_task = 0;
11                number_of_active_tasks = 0;
12            }
13        } while (suspended[current_task] != false);
14    }
```

```

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

29 void resume_task ( uint8_t tin) {
30     suspended[tin] = false;
31     exit_low_power_state ();
32 }

```

9 If relinquish() finds no ready to run tasks in the course of a scan then the can go into a low power state until there is an unblocking event. An ISR calling resume would also 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 the 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 work independently. 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  pushall();
9  sp_save[current_task] = _SP;
10 while (1) {
11     do {
12         if (++current_task == number_of_tasks) {
13             current_task = 0;
14         } while (suspended[current_task] != false);
15         if (start_from_beginning[current_task] == true) {
16             start_from_beginning[current_task] = false;
17             _SP = starting_stack[current_task];
18             prelude_addresses[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         popall();
26         prelude_addresses[current_task] ();
27         return;
28     }
29 }

```

Appendix B - Stacks and Stack Pointers

Extremely Small Systems - Saving stack space

It is an anomaly of modern RISC μ P's that they have an enormous number of registers, far more than the compiler can use, but 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 processor 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      pushall();
4      sp_save[current_task] = _SP;
5      _SP = system_stack;
6      enable ();
7      while (1) {
8          do {
9              if (++current_task == number_of_tasks)
10                 current_task = 0;
11             } while (suspended[current_task] != false );
12             if (start_from_beginning[current_task] == true ) {
13                 start_from_beginning[current_task] = false ;
14                 _SP = starting_stack[current_task];
15                 start_addresses[current_task] ();
16                 suspended[current_task] = true ;
17                 start_from_beginning[current_task] = true ;
18             }

```

```

19     else {
20         disable ();
21         _SP = sp_save[current_task];
22         popall();
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] ();
16 }

```

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     pushall();
4     sp_save[current_task] = SP;
5     while (1) {
6         do {
7             if (++current_task == number_of_tasks)
8                 current_task = 0;
9         } while (suspended[current_task] != false );
10        if (start_from_beginning[current_task] == true ) {
11            start_from_beginning[current_task] = false ;
12            disable ();
13            SP = starting_stack[current_task];
14            enable ();
15            start_addresses[current_task] ();
16            suspended[current_task] = true ;
17            start_from_beginning[current_task] = true ;
18        }
19        else {
20            disable ();
21            SP = sp_save[current_task];
22            enable ();
23            popall();
24            return;
25        }
26    }
27 }

```

Appendix C - Processor Customization

Three versions of the kernel and I/O & interrupt module are provided: one set each for Atmel's AVR ATmega168/328 processor, Texas Instruments' MSP430(FR6989) processor and Microchip's PIC24(FJ128GA204) processor. Changes may be needed to the MSP code for versions of the μ P with differing numbers of timers (a confusing subject all on its own).

When using CoRTOS with another processor:

<u>Files to be changed</u>	<u>Changed element</u>
CoRTOSkernel???.c	Stack pointer access, stack growth direction
CoRTOStask.c	Stack size
CoRTOSasmmacros???.h	Register pushes & pops on context switch
CoRTOSioint???.c	Timer interrupt, enable and disable interrupts, I/O and timer initialization

Appendix D - 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. Stack sizes would be set in your main module, or anywhere you might find convenient. In the distribution code they are in CoRTOSsuperblinky???.c and CoRTOStask.c.

If the processor vectors off into a violation/exception/fault interrupt it may be that stack overflow checking needs to be disabled or modified or the size of RAM set aside for the stack needs to be increased. Needless to say CoRTOS, like an other OS, uses far more stack space than any compiler would calculate as sufficient.

If relinquish() is forever stuck trying to find an active task then there is 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. Also check that the timer generating the interrupt is running at the right speed - it isn't hard to get registers and dividers turned around and have the timer ISR run at 1/100Hz rather than 100Hz.