FreeRTOS

A Brief Overview

Christopher Kenna
Avionics

October 1, 2010

Outline

Background Information

- The FreeRTOS Project supports 25 *official* architecture ports, with many more community developed ports.
- The FreeRTOS RT kernel is portable, open source, royalty free, and very small.
- OpenRTOS is a commercialized version by the sister company High Integrity Systems.
- Richard Barry: I know FreeRTOS has been used in some rockets and other aircraft, but nothing too commercial.

  We have customers that use it on ship systems, and WITTENSTEIN sell SafeRTOS which has been certified to various standards, but not DO-178B.

Licensing Information (0/2)

The license is a modified GNU GPL according to the table below.

	FreeRTOS Modified GPL	OpenRTOS Commercial
Free?	✓	✗
Use it in a commercial application?	✓	✓
Royalty free?	✓	✓
Must open source code that makes use of FreeRTOS services?	✗[1]	✗
Must open source kernel changes?	✓	✗

[1] As long as code provides functionality that is distinct from that provided by FreeRTOS.

Licensing Information (1/2)

The license is a modified GNU GPL according to the table below.

	FreeRTOS Modified GPL	OpenRTOS Commercial
Must document that the product uses FreeRTOS?	✓[2]	✗
Required to offer FreeRTOS code to application users?	✓	✗
Technical support for OS?	✗[3]	✓
Warranty?	✗	✓

[2] Web link to FreeRTOS.org is sufficient.
[3] There is an on-line community, though.

FreeRTOS Configuration

- The operation of FreeRTOS is governed by `FreeRTOS.h`, with application specific configuration appearing in `FreeRTOSConfg.h`.
- Obviously, these are static configuration options.
- Some examples:
  - `configUSE_PREEMPTION`
  - `configCPU_CLOCK_HZ` – CPU clock frequency, not necessarily the bus frequency.
  - `configTICK_RATE_HZ` – RTOS tick frequency that dictates interrupt frequency.
  - `configMAX_PRIORITIES` – Total number of priority levels. Each level creates a new list, so memory sensitive machines should keep this to a minimum.
  - And more...

Outline

Tasks in FreeRTOS

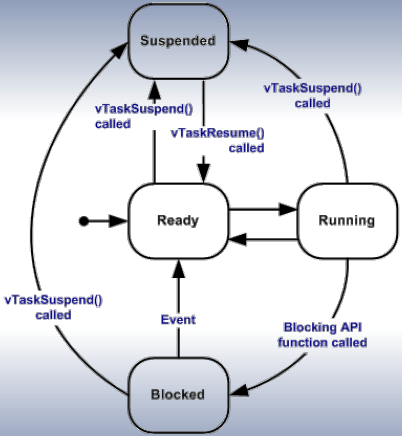
- Tasks have their own context. No dependency on other tasks unless defined.
- One task executes at a time.
- Tasks have no knowledge of scheduler activity. The scheduler handles context switching.
- Thus, tasks each have their own stack upon which execution context can be saved.
- Prioritized and preemptable.

Co-Routines

- Co-routines are intended for use on small processors that have severe RAM constraints. Co-Routines share a single stack.
- Co-routines are implemented through macros.
- Prioritized relative to other co-routines, but preempted by tasks.
- The structure of co-routines is rigid due to the unconventional implementation.
    - Lack of stack requires special consideration.
    - Restrictions on where API calls can be made.
    - Cannot communicate with tasks.
- Will examine tasks in more detail, but not co-routines.

Task States

- **Running** – Actively executing and using the processor.
- **Ready** – Able to execute, but not because a task of equal or higher priority is in the *Running* state.
- **Blocked** – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling `vTaskDelay()` to block until delay period has expired. Always have a "timeout" period, after which the task is unblocked.
- **Suspended** – Only enter via `vTaskSuspend()`, depart via `xTaskResume()` API calls.



Task Priorities

- Each task gets a priority from 0 to `configMAX_PRIORITIES` $- 1$
- Priority can be set on a per application basis.
- Tasks can change their own priority, as well as the priority of other tasks.
- `tskIDLE_PRIORITY` $= 0$

The Idle Task

- The idle task is created automatically when the scheduler is started.

The Idle Task

- The idle task is created automatically when the scheduler is started.
- It frees memory allocated by the RTOS to tasks that have since been deleted.
- Thus, applications that use `vTaskDelete()` to remove tasks should ensure the idle task is not starved.

The Idle Task

- The idle task is created automatically when the scheduler is started.
- It frees memory allocated by the RTOS to tasks that have since been deleted.
- Thus, applications that use `vTaskDelete()` to remove tasks should ensure the idle task is not starved.
- The idle task has no other function, so cases when the idle task need never run exist.

The Idle Task

- The idle task is created automatically when the scheduler is started.
- It frees memory allocated by the RTOS to tasks that have since been deleted.
- Thus, applications that use `vTaskDelete()` to remove tasks should ensure the idle task is not starved.
- The idle task has no other function, so cases when the idle task need never run exist.
- There is an idle task hook, which can do some work at each idle interval without the RAM usage overhead associated with running a task at the idle priority.

Task Control Block (TCB) (0/2)

- A TCB is allocated to each task. It stores the task's context.

Source/tasks.c

```
typedef struct tskTaskControlBlock
{
  :: Top of task's stack. Must be first member b/c of context switch
  volatile portSTACK_TYPE *pxTopOfStack; :: code (later slides).

  #if ( portUSING_MPU_WRAPPERS == 1 )
    :: The MPU settings are defined as part of the port layer.
    xMPU_SETTINGS xMPUSettings; :: Must be 2nd member of struct.
  #endif

  :: List item used to place the TCB in ready and blocked queues.
  xListItem       xGenericListItem;
  :: Used to place the TCB in event lists.
  xListItem       xEventListItem;
  :: The priority of the task where 0 is the lowest priority.
  unsigned portBASE_TYPE  uxPriority;
  :: Points to the start of the stack.
  portSTACK_TYPE      *pxStack;
  :: Descriptive name given to the task when created.
  :: Facilitates debugging only.
  signed char        pcTaskName[ configMAX_TASK_NAME_LEN ];
```

Task Control Block (TCB) (1/2)

Source/tasks.c

```
  #if ( portSTACK_GROWTH > 0 )
    :: Used for stack overflow checking on architectures where the
    :: stack grows up from low memory.
    portSTACK_TYPE *pxEndOfStack;
  #endif
  ...
  #if ( configUSE_TRACE_FACILITY == 1 )
    :: Used only for tracing the scheduler, making debugging easier.
    unsigned portBASE_TYPE  uxTCBNumber;
  #endif

  #if ( configUSE_MUTEXES == 1 )
    :: The priority last assigned to the task, used by the priority
    :: inheritance mechanism.
    unsigned portBASE_TYPE uxBasePriority;
  #endif
  ...
  #if ( configGENERATE_RUN_TIME_STATS == 1 )
    :: Used for calculating how much CPU time each task is utilizing.
    unsigned long ulRunTimeCounter;
  #endif

} tskTCB;
```

Implementing a Task

Source/include/projdefs.h

```
/* Defines the prototype to which task functions must conform. */
typedef void (*pdTASK_CODE)( void * );
```

```
void vATaskFunction( void *pvParameters ){
    for( ;; ){
        :: Task application code here.
    }
}
```

- Tasks are always a function that returns void and takes a void pointer.
- Tasks should never return (loop forever).
- Not much to it, really.

Creating a Task (0/3)

- The kernel creates a task by instantiating and populating a TCB. New tasks are placed in the Ready state and added to the Ready list.
- If the task is the highest priority task, then it is set as the currently running task.
- Created by calling `xTaskCreate()` and deleted by calling `vTaskDelete()`.
- `xTaskCreate()` takes the following parameters.
  - A pointer to the function that implements the task (type `pdTASK_CODE` from earlier).
  - A name for the task.
  - The depth of the task's stack.
  - The task's priority.
  - A pointer to any parameters needed by the task's function.

Creating a Task (1/3)

- An interesting step in task creation is preparing the task for its first context switch.
- The TCB stack is initialized to look as if the task was already running, but had been interrupted by the scheduler. The return address is set to the start of the task function with `pxPortInitialiseStack`.
- See code on next slide.

Creating a Task (2/3)

Source/portable/GCC/ATMega323.c (edited)

```
portSTACK_TYPE *pxPortInitialiseStack( portSTACK_TYPE *pxTopOfStack,
        pdTASK_CODE pxCode, void *pvParameters )
{
  :: Place some known values on bottom of stack, for debugging.
  *pxTopOfStack = 0x11;
  pxTopOfStack--;
  *pxTopOfStack = 0x22;
  pxTopOfStack--;
  ...

  :: Start of task code will be popped off last, so push it on first.
  unsigned short usAddress = ( unsigned short ) pxCode;
  *pxTopOfStack =
      (portSTACK_TYPE) (usAddress & (unsigned short) 0x00ff);
  pxTopOfStack--;
  usAddress >>= 8;
  *pxTopOfStack =
      (portSTACK_TYPE) (usAddress & (unsigned short) 0x00ff);
  pxTopOfStack--;

  :: And then push values for CPU registers, taking into account
  :: what the compiler expects for this architecture.
  ...
```

Outline

Queue Overview

- Queues are the primary form of inter-task communications.
- They can send messages between tasks as well as between interrupts and tasks.
- Supports appending data to the back of a queue, or sending data to the head of a queue.
- Queues can hold arbitrary items of a fixed size.
- The size of each item and the capacity of the queue are defined when the queue is created.
- Items are enqueued by copy, not reference.

Queues and Blocking

- Access to queues is either blocking or non-blocking.
- The scheduler blocks tasks when they attempt to read from or write to a queue that is either empty of full, respectively.
- If the `xTicksToWait` variable is zero and the queue is empty (full), the task does not block. Otherwise, the task will block for `xTicksToWait` scheduler ticks or until an event on the queue frees up the resource.
- This includes attempts to obtain semaphores, since they are special cases of queues.

Binary Semaphores

- Used for both mutual exclusion and synchronization.
- Commonly used by Interrupt Service Routines (ISRs) to wake tasks and avoid polling. Consider the following example.

Example: waking a task from an ISR

Binary Semaphores

- Used for both mutual exclusion and synchronization.
- Commonly used by Interrupt Service Routines (ISRs) to wake tasks and avoid polling. Consider the following example.

Example: waking a task from an ISR

1. A binary semaphore is created along with a task that blocks on this semaphore.

Binary Semaphores

- Used for both mutual exclusion and synchronization.
- Commonly used by Interrupt Service Routines (ISRs) to wake tasks and avoid polling. Consider the following example.

Example: waking a task from an ISR

1. A binary semaphore is created along with a task that blocks on this semaphore.
2. An ISR is written for a peripheral that sets the semaphore when the peripheral requires servicing using `xSemaphoreGiveFromISR()`.

Binary Semaphores

- Used for both mutual exclusion and synchronization.
- Commonly used by Interrupt Service Routines (ISRs) to wake tasks and avoid polling. Consider the following example.

Example: waking a task from an ISR

1. A binary semaphore is created along with a task that blocks on this semaphore.
2. An ISR is written for a peripheral that sets the semaphore when the peripheral requires servicing using `xSemaphoreGiveFromISR()`.
3. This awakens the task waiting on this semaphore. The task resets the semaphore, does some work, and then blocks again.

Counting Semaphores and Mutexes

Counting Semaphores

- FreeRTOS has support for counting semaphores, the standard down and up (wait and signal) operations.
- Semaphores are macros defined over the queue API. Therefore, semaphores use the same API calls as queues do.

Counting Semaphores and Mutexes

Counting Semaphores

- FreeRTOS has support for counting semaphores, the standard down and up (wait and signal) operations.
- Semaphores are macros defined over the queue API. Therefore, semaphores use the same API calls as queues do.

Mutexes

- FreeRTOS supports mutexes (binary semaphores with priority inheritance).
- As mutexes use the semaphore API, they also support the blocking timeout mechanism. Moreover, they are implemented using the queue API calls.

Counting Semaphores and Mutexes

Counting Semaphores

- FreeRTOS has support for counting semaphores, the standard down and up (wait and signal) operations.
- Semaphores are macros defined over the queue API. Therefore, semaphores use the same API calls as queues do.

Mutexes

- FreeRTOS supports mutexes (binary semaphores with priority inheritance).
- As mutexes use the semaphore API, they also support the blocking timeout mechanism. Moreover, they are implemented using the queue API calls.
- The queue data strictures are protected from corruption by disabling interrupts in some places and disabling the scheduler in others where it would be okay for an interrupt to occur, but not for another task to preempt the queuing task.

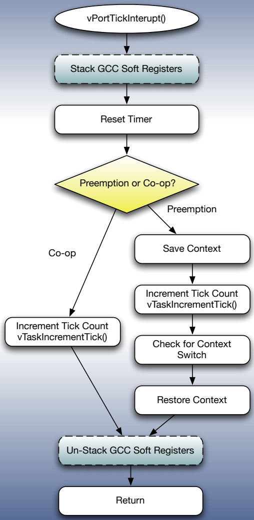
Outline

General Scheduler Operation

- As mentioned, the scheduler can be co-operative or preemptive.
- The scheduler is an interrupt handler activated on each tick of hardware clock. Therefore, it contains hardware specific code.
- The ready list is arranged in order of priority with tasks of equal priority being served on a round-robin basis.
- The scheduler starts with the highest priority list and works its way downward.
- There is no explicit Running list or state. The kernel maintains `pxCurrentTCB` to identify the process in the Ready list that is currently running.

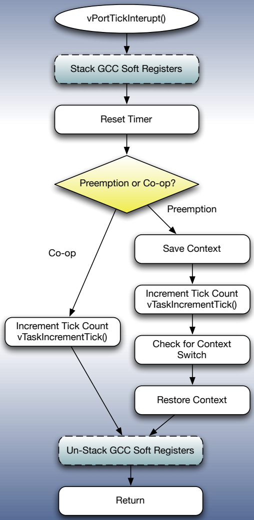
Performing a Context Switch (0/2)

- The scheduler first resets the counter timer.
- Without preemption enabled, the timer interrupt simply increments the tick count and returns.



Performing a Context Switch (1/2)

- If the scheduler is preemptive, the scheduler pushes the current task context on the stack.
- Then it increments the tick count and checks to see if this action has caused a blocked task to unblock.
- If a task of higher priority unblocked, then a context switch is executed.
- Context is restored.
- The scheduler returns, potentially starting the execution of a different task than the one that was interrupted.



ISR for Ticks (0/2)

Source/portable/GCC/ATMega323/port.c (edited)

```
#if configUSE_PREEMPTION == 1
  void SIG_OUTPUT_COMPARE1A( void ) __attribute__(( signal, naked ));
  void SIG_OUTPUT_COMPARE1A( void ){
    vPortYieldFromTick();
    asm volatile ( "reti" );
  }
#else
  void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
  void SIG_OUTPUT_COMPARE1A( void ) {
    vTaskIncrementTick();
  }
```

- Explanation on next slide.

ISR for Ticks (1/2)

`Source/portable/GCC/ATMega323/port.c` (edited)

```
#if configUSE_PREEMPTION == 1
  void SIG_OUTPUT_COMPARE1A( void ) __attribute__(( signal, naked ));
  ...
```

- The `signal` attribute informs GCC the function is an ISR, which means GCC saves and restores every register the ISR modifies and the return uses `reti` instead of `ret`. The AVR microcontroller disables interrupts upon entering an ISR, and `reti` is required to re-enable them.
- But FreeRTOS saves registers, so `naked` is used so that GCC won't generate any function entry or exit code. Macros `portSAVE_CONTEXT()` and `portRESTORE_CONTEXT()` save and restore the execution context.

The Context Saving Macro (0/2)

`Source/portable/GCC/ATMega323/port.c` (edited)

```
#define portSAVE_CONTEXT()                               \
  asm volatile (  "push r0                         \n\t" \  :: Save register r0.
          "in   r0, __SREG__            \n\t" \ :: See (1) below.
          "cli                    \n\t" \       :: Disable interrupts.
          "push r0                  \n\t" \     :: Save processor status.
          "push r1                  \n\t" \     :: Save original r1 value.
          "clr  r1                  \n\t" \     :: See (2) below.
          "push r2                  \n\t" \     :: Save r3 through r30.
          ...
          "push r31                 \n\t" \
          ...
```

1. Before the microcontroller jumps to an ISR, it pushes the PC onto the stack, which is why this is not done here.
2. Move processor status register to r0.
3. The instruction `clr r1` sets `r1` to zero, because the GCC generated ISR expects it to be zero.
4. Continued on next slide.

The Context Saving Macro (1/2)

`Source/portable/GCC/ATMega323/port.c` (edited)

```
        ...
        "push r31              \n\t" \
        "lds  r26, pxCurrentTCB    \n\t" \    :: See (1).
        "lds  r27, pxCurrentTCB + 1 \n\t" \
        "in   r0, 0x3d          \n\t" \       :: (2)
        "st   x+, r0            \n\t" \
        "in   r0, 0x3e          \n\t" \       :: (3)
        "st   x+, r0            \n\t" \
    );
```

1. The `X` register allows data indirect addressing with pre-decrement. `r26` is `X` low byte, 27 the high byte. The `X` processor register is loaded with the address to which the stack pointer is to be saved.
2. Save stack pointer low byte.
3. Save stack pointer high nibble.
4. This is the reason `struct tskTCB`'s first member must be `portSTACK_TYPE *pxTopOfStack`.

Incrementing the Tick (0/2)

Source/task.c (edited)

```
void vTaskIncrementTick( void ){
  if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE ){
    :: In this case, the scheduler not suspended.
    ++xTickCount; :: Increment ticks.

    if( xTickCount == ( portTickType ) 0 ) {
      xList *pxTemp;

      /*
      Tick count has overflowed so we need to swap the delay lists.
      If there are any items in pxDelayedTaskList here then there is
      an error!
      */
      pxTemp = pxDelayedTaskList;
      pxDelayedTaskList = pxOverflowDelayedTaskList;
      pxOverflowDelayedTaskList = pxTemp;
      xNumOfOverflows++;
    }
    prvCheckDelayedTasks(); :: Has tick made a timeout expire?
  } else {
    ...
```

Incrementing the Tick (1/2)

Source/task.c (edited)

```
   ...
   } else {
    ++uxMissedTicks; :: Scheduler is suspended, so we missed a tick.

    :: The tick hook gets called at regular intervals, even if the
    :: scheduler is locked.
    #if ( configUSE_TICK_HOOK == 1 ){
      extern void vApplicationTickHook( void );
      vApplicationTickHook();
    }
    #endif
  }

  #if ( configUSE_TICK_HOOK == 1 ){
    extern void vApplicationTickHook( void );

    /* Guard against the tick hook being called when the missed tick
    count is being unwound (scheduler is being unlocked). */
    if( uxMissedTicks == 0 )
      vApplicationTickHook();
  }
  #endif
  ...
```

References

- FreeRTOS website: www.FreeRTOS.org
- Atmel AVR Instruction Set Guide
  www.Atmel.com/ateml/acrobat/doc0856.pdf