FreeRTOS

Alberto Bosio

Université de Montpellier

bosio@lirmm.fr

February 27, 2017

Outlook

Definitions

- FreeRTOS is suited to deploy embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements;
- **Soft real-time** requirements are those that state a time deadline, but breaching the deadline would not render the system useless;
- **Hard real-time** requirements are those that state a time deadline, and breaching the deadline would result in absolute failure of the system.

Definitions

- FreeRTOS is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements;
- It allows applications to be organized as a collection of independent threads of execution;
- The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer;
- In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements.

Task Functions

- Tasks are implemented as C functions which must return void and take a void pointer parameter:

Example (Task Prototype)

```
void ATaskFunction( void *pvParameters );
```

Task Scheleton

```c
void ATaskFunction( void *pvParameters )
{
 /* Variables declaration. */
 int32_t lVariableExample = 0;
 /* A task will normally be
implemented as an infinite loop. */
 for( ;; ) {
    /* The code to implement
 the task functionality will go here. */
 }

  vTaskDelete( NULL );
}
```

Creating Tasks

- Tasks are created using the FreeRTOS xTaskCreate() API function.

Prototype

```
BaseType_t xTaskCreate (
        TaskFunction_t pvTaskCode,
        const char * const pcName,
        uint16_t usStackDepth,
        void *pvParameters,
        UBaseType_t uxPriority,
        TaskHandle_t *pxCreatedTask )
```

Creating Tasks

- **pvTaskCode**: The pvTaskCode parameter is simply a pointer to the function that implements the task (in effect, just the name of the function);
- **pcName**: A descriptive name for the task. This is not used by FreeRTOS in any way. It is included purely as a debugging aid;
- **usStackDepth**: Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack;
- **pvParameters**: Task functions accept a parameter of type pointer to void (void*);

Creating Tasks

- **uxPriority**: Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority to the highest priority;
- **pxCreatedTask**: pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL;
- There are two possible return values: **pdPASS** or **pdFAIL**. Fail indicates that the task has not been created because there is insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack

Example

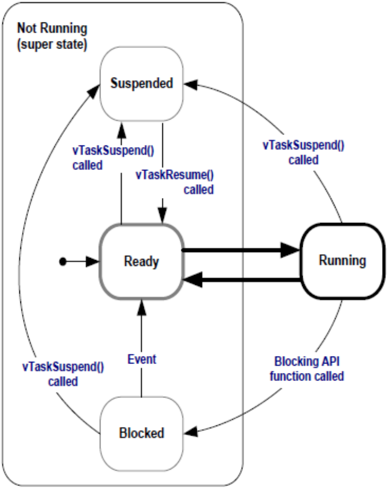
```
#include "FreeRTOS.h"
int main(void)
{
        xTaskCreate(task1, "task1",
                configMINIMAL_STACK_SIZE, NULL, 1, 
        xTaskCreate(task2, "task2",
                configMINIMAL_STACK_SIZE, NULL, 1, 

        /* Start the scheduler so
                the tasks start executing. */
        vTaskStartScheduler();

        for(;;);
}
```

Top Level Task States

- An application can consist of many tasks;



Creating a Delay

Prototype

```c
void vTaskDelay( TickType_t xTicksToDelay );
```

Creating a Delay

- **xTicksToDelay**: The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state, and remain in the Blocked state until the tick count reached 10,100;
- The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. For example, calling vTaskDelay(pdMS_TO_TICKS(100)) will result in the calling task remaining in the Blocked state for 100 milliseconds.

Creating a Period Task

- vTaskDelay() parameter specifies the number of tick interrupts that should occur between a task calling vTaskDelay(), and the same task once again transitioning out of the Blocked state. The length of time the task remains in the blocked state is specified by the vTaskDelay() parameter, but the time at which the task leaves the blocked state is relative to the time at which vTaskDelay() was called.

Creating a Period Task

Prototype

```
void vTaskDelayUnti (
        TickType_t * pxPreviousWakeTime,
        TickType_t xTimeIncrement
 );
```

Creating a Period Task

- **xPreviousWakeTime**: This parameter is named on the assumption that vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency. In this case, pxPreviousWakeTime holds the time at which the task last left the Blocked state (was woken up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state.
- **xTimeIncrement**: This parameter is also named on the assumption that vTaskDelayUntil() is being used to implement a task that executes periodically and with a fixed frequency (the frequency being set by the xTimeIncrement value). xTimeIncrement is specified in ticks. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks.

Tasks State Review

- The task that is actually running (using processing time) is in the Running state.
- Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state.
  - Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state
- Tasks can wait in the Blocked state for an event and are automatically moved back to the Ready state when the event occur

Scheduler

- The task that is actually running (using processing time) is in the Running state.
- Tasks that are not actually running, but are not in either the Blocked state or the Suspended state, are in the Ready state.
    - Tasks that are in the Ready state are available to be selected by the scheduler as the task to enter the Running state. The scheduler will always choose the highest priority Ready state task to enter the Running state
- Tasks can wait in the Blocked state for an event and are automatically moved back to the Ready state when the event occur

Configuring the Scheduling Algorithm

- The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state.
- The algorithm can be configured by specifying:
  - configUSE_PREEMPTION
  - configUSE_TIME_SLICING
- The options have to be configured into *FreeRTOSConfig.h*

Configuring the Scheduling Algorithm

- In all possible configurations the FreeRTOS scheduler will use a *Round Robin algorithm*.
- A Round Robin scheduling algorithm does not guarantee time is shared equally between tasks of equal priority, only that Ready state tasks of equal priority will enter the Running state in turn.

Prioritized Pre-emptive Scheduling with Time Slicing

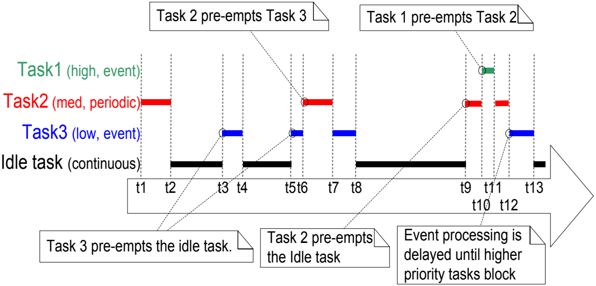
- The configuration:
  - configUSE_PREEMPTION set to 1
  - configUSE_TIME_SLICING set to 1
- sets the FreeRTOS scheduler to use a scheduling algorithm called 'Fixed Priority Pre-emptive Scheduling with Time Slicing', which is the scheduling algorithm used by most small RTOS applications.

Prioritized Pre-emptive Scheduling with Time Slicing

- Where:
  - **Fixed Priority**: Scheduling algorithms described as 'Fixed Priority' do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority, or that of other tasks;
  - **Pre-emptive**: Pre emptive scheduling algorithms will immediately 'pre empt' the Running state task if a task that has a priority higher than the Running state task enters the Ready state. Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state and into the Ready state to allow a different task to enter the Running state;
  - **Time Slicing**: Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using 'Time Slicing' will select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

Prioritized Pre-emptive Scheduling with Time Slicing

- Example:

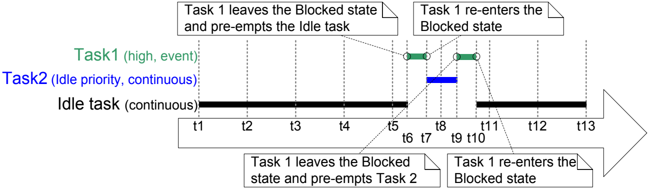


Prioritized Pre-emptive Scheduling (without Time Slicing)

- The configuration:
  - configUSE_PREEMPTION set to 1
  - configUSE_TIME_SLICING set to 0
- If time slicing is not used, then the scheduler will only select a new task to enter the Running state when either:
  - A higher priority task enters the Ready state;
  - The task in the Running state enters the Blocked or Suspended state;

Prioritized Pre-emptive Scheduling (without Time Slicing)

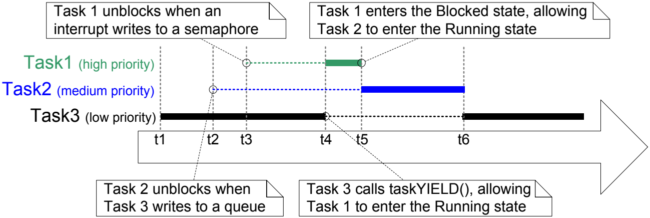
- Example:



Co-operative Scheduling

- The configuration:
  - configUSE_PREEMPTION set to 0
  - configUSE_TIME_SLICING set to any value
- When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling *taskYIELD()*. Tasks are never pre-empted, so time slicing cannot be used.

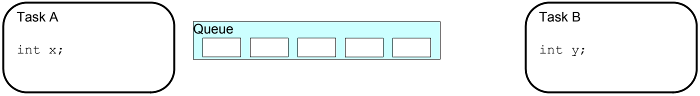
- Example:


Task 1 unblocks when an interrupt writes to a semaphore
Task 1 enters the Blocked state, allowing Task 2 to enter the Running state
Task1 (high priority)
Task2 (medium priority)
Task3 (low priority)
t1
t2
t3
t4
t5
t6
Task 2 unblocks when Task 3 writes to a queue
Task 3 calls taskYIELD(), allowing Task 1 to enter the Running state

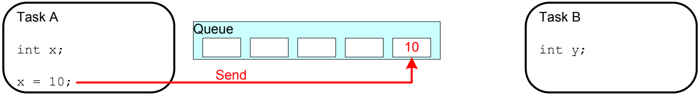
Characteristics of a Queue

- A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created;
- Queues are normally used as First In First Out (FIFO) buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue.

Example

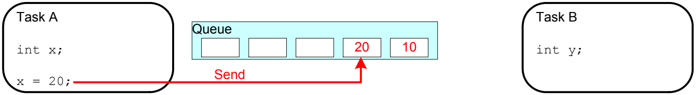


A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

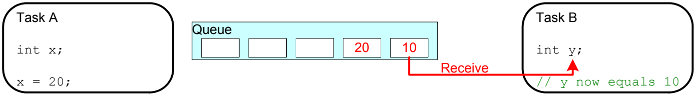


Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

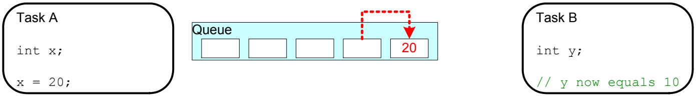
Example (cont)



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Access by Multiple Tasks

- Queues are objects in their own right that can be accessed by any task that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue. In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.;

Blocking on Queue Reads

- When a task attempts to read from a queue, it can **optionally** specify a 'block' time. This is the time the task will be kept in the Blocked state to wait for data to be available from the queue, should the queue already be empty.
- A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue.
- The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.

Blocking on Queue Writes

- Just as when reading from a queue, a task can **optionally** specify a block time when writing to a queue.
- In this case, the block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.

Using a Queue

Create

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength,
 UBaseType_t uxItemSize );
```

Using a Queue

- **uxQueueLength**: The maximum number of items that the queue being created can hold at any one time;
- **uxItemSize**: The size in bytes of each data item that can be stored in the queue
- **Return Value**: A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

Write

Prototypes

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,
                              const void * pvItemToQueue,
                              TickType_t xTicksToWait );

BaseType_t xQueueSendToBack( QueueHandle_t xQueue,
                             const void * pvItemToQueue,
                             TickType_t xTicksToWait );
```

Write

- **xQueue**: The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue;
- **pvItemToQueue**: A pointer to the data to be copied into the queue;
- **xTicksToWait**: The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full;
- **Return Value**:
  - pdPASS will be returned only if data was successfully sent to the queue.
  - errQUEUE_FULL will be returned if data could not be written to the queue because the queue was already full.

Read

Prototype

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void * const pvBuffer,
            TickType_t xTicksToWait );
```

Read

- **xQueue**: The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue;
- **pvBuffer**: A pointer to the memory into which the received data will be copied;
- **xTicksToWait**: The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty;
  - If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty.
- **Return Value**:
  - pdPASS will be returned only if data was successfully read from the queue.
  - errQUEUE_EMPTY will be returned if data cannot be read from the queue because the queue is already empty.

Example

```
static void vSenderTask( void *pvParameters ) {
        int32_t lValueToSend;
        BaseType_t xStatus;

        lValueToSend = ( int32_t ) pvParameters;

        for( ;; ) {

        xStatus = xQueueSendToBack(
                xQueue, &lValueToSend, 0 );
        if( xStatus != pdPASS ) {

            vPrintString( "Error" );
                }
        }
}
```

Example

```
static void vReceiverTask( void *pvParameters ) {
        int32_t lReceivedValue;
        BaseType_t xStatus;
        const TickType_t xTicksToWait =
                pdMS_TO_TICKS( 100 );
        for( ;; ) {
                xStatus = xQueueReceive(
        xQueue, &lReceivedValue, xTicksToWait );
        if( xStatus == pdPASS )
                vPrintStringAndNumber( "Received␣=␣
           else
                vPrintString( "erro" );
        }
}
```

Example

```
QueueHandle_t xQueue;
int main( void )
{
    xQueue = xQueueCreate( 5, sizeof( int32_t ) );
    if( xQueue != NULL )
    {

        xTaskCreate( vSenderTask, "Sender1",
        1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2",
                1000, ( void * ) 200, 1, NULL );
        xTaskCreate( vReceiverTask, "Receiver",
                1000, NULL, 1, NULL );

        vTaskStartScheduler();
    }
```

Binary Semaphore (Mutex)

- The binary semaphore can be considered conceptually as a queue with a length of one.
- The queue can contain a maximum of one item at any time, so is always either empty or full (hence, binary).

Init

Prototype

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Init

- **Return Value**:
  - If NULL is returned, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
  - A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

Sem_P

Prototype

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore,
TickType_t xTicksToWait );
```

Sem_P

- **xSemaphore**: The semaphore being 'taken'.
- **xTicksToWait**: The maximum amount of time the task should remain in the Blocked state to wait for the semaphore if it is not already available. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without a timeout) if INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
- **Return Value**:
  - pdPASS is returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore.
  - The semaphore is not available. It will return pdFAIL.

Sem_V

Prototype

```
BaseType_t xSemaphoreGive (
SemaphoreHandle_t xSemaphore );
```

Init

- **xSemaphore**: The semaphore being 'given'.
- **Return Value**:
  - pdPASS will be returned only if the call to xSemaphoreGiveFromISR() is successful.
  - If a semaphore is already available, it cannot be given, and xSemaphoreGiveFromISR() will return pdFAIL.

Counting Semaphores

- Counting semaphores can be thought of as queues that have a length of more than one.
- Tasks are not interested in the data that is stored in the queue, just the number of items in the queue.
- configUSE_COUNTING_SEMAPHORES must be set to 1 in FreeRTOSConfig.h for counting semaphores to be available.

Init

Prototype

```
SemaphoreHandle_t xSemaphoreCreateCounting(
UBaseType_t uxMaxCount,
                  UBaseType_t uxInitialCount );
```

Init

- **uxMaxCount**: The maximum value to which the semaphore will count. To continue the queue analogy, the uxMaxCount value is effectively the length of the queue.
- **uxInitialCount**: The initial count value of the semaphore after it has been created.
- **Return Value**:
  - If NULL is returned, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
  - A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.