FreeRTOS

Everything about FreeRTOS

Queues

This article provides examples of using RTOS Queues.

Why RTOS Queues

There are standard queues, or <vector> in C++, but RTOS queues should almost always be used in your application because they are thread-safe (no race conditions with multiple tasks), and they co-operate with your RTOS to schedule the tasks. For instance, your task could optionally sleep while receiving data if the queue is empty, or it can sleep while sending the data if the queue is full.

Queues vs. Binary Semaphore for "Signal"

Binary Semaphores may be used to "signal" between two contexts (tasks or interrupts), but they do not contain any payload. For example, for an application that captures a keystroke inside of an interrupt, it could "signal" the data processing task to awake upon the semaphore, however, there is no payload associated with it to identify what keystroke was input. With an RTOS queue, the data processing task can wake up on a payload and process a particular keystroke.

The data-gathering tasks can simply send the key-press detected to the queue, and the processing task can receive items from the queue, and perform the corresponding action. Moreover, if there are no items in the queue, the consumer task (the processing one) can sleep until data becomes available. You can see how this scheme lends itself well to having multiple ISRs queue up data for a task (or multiple tasks) to handle.

Examples

Simple

After looking through the sample code below, you should then watch this video.

Let us study an example of two tasks communicating to each other over a queue.

```
QueueHandle_t handle_of_int_queue;
void producer(void *p) {
  int x = 0;

  while (1) {
    vTaskDelay(100);
    ++x;
    if (xQueueSend(handle_of_int_queue, &x, 0)) {
    }
  }
}
void consumer(void *p) {
  while (1) {
    // We do not need vTaskDelay() because this task will sleep for up to 100 ticks until there is an i
    if (xQueueReceive(handle_of_int_queue, &x, 100)) {
      printf("Received %i\n", x);
    } else {
      puts("Timeout --> No data received");
    }
  }
}
void main(void) {
  // Queue handle is not valid until you create it
  handle_of_int_queue = xQueueCreate(10, sizeof(int));
}
```

Queue usage with Interrupts

When an item is sent from within an interrupt (or received), the main difference in the API is that there is no way to "sleep". For example, we cannot sleep while waiting to write an item to the queue if the queue is full. FreeRTOS API has dedicated API to be used from within ISRs, and other RTOSs simply state that you can use the same API, but the sleep time has to be zero if you are inside of an interrupt.

With the FreeRTOS `FromISR` API, in place of the sleep time is a pointer to a variable that informs us if an RTOS scheduling yield is required, and FreeRTOS asks us to yield in our application code.

```
static QueueHandle_t uart_rx_queue;
```

```
// Queue API is special if you are inside an ISR
void uart_rx_isr(void) {
  BaseType_t yield_required = 0;
  if (!xQueueSendFromISR(uart_rx_queue, &x, &yield_required)) {
    // TODO: Queue was full, handle this case
  }

  portYIELD_FROM_ISR(yield_required);
}
void queue_rx_task(void *p) {
  int x = 0;
  // Receive is the usual receive because we are not inside an ISR
  while (1) {
    if(xQueueReceive(uart_rx_queue, &x, portMAX_DELAY)) {
    }
  }
}
void main(void) {
   uart_rx_queue = xQueueCreate(10, sizeof(char));}
```

Advanced Examples

Multiple Producers and Consumers

There are multiple ways to create multiple producers and consumers. The easiest way to do so at the expense of a potentially excessive number of tasks is to have multiple tasks for each producer, and for each consumer.

```
static QueueHandle_t light_sensor_queue;
static QueueHandle_t temperature_sensor_queue;
void light_sensor_task(void *p) {
  while (1) {
    const int sensor_value = rand(); // Some random value
    if(xQueueSend(light_sensor_queue, &sensor_value, portMAX_DELAY)) {
    }
    vTaskDelay(1000);
```

```
  }
}
void temperature_sensor_task(void *p) {
  while (1) {
    const int temperature_value = rand(); // Some random value
    if(xQueueSend(temperature_sensor_queue, &temperature_value, portMAX_DELAY)) {
    }
    vTaskDelay(1000);
  }
}
void consumer_of_light_sensor(void *p) {
  int light_sensor_value = 0;
  while(1) {
    xQueueReceive(light_sensor_queue, &light_sensor_value, portMAX_DELAY);
    printf("Light sensor value: %i\n", light_sensor_value);
  }
}
void consumer_of_temperature_sensor(void *p) {
  int temperature_sensor_value = 0;
  while(1) {
    xQueueReceive(temperature_sensor_queue, &temperature_sensor_value, portMAX_DELAY);
    printf("Temperature sensor value: %i\n", temperature_sensor_value);
  }
}
void main(void) {
   light_sensor_queue = xQueueCreate(3, sizeof(int));
   temperature_sensor_queue = xQueueCreate(3, sizeof(int));

   xTaskCreate(light_sensor_task, ...);
   xTaskCreate(temperature_sensor_task, ...);

   xTaskCreate(consumer_of_light_sensor, ...);
   xTaskCreate(consumer_of_temperature_sensor, ...);}
```

Multiple Producers, 1 Consumer

In order to create multiple producers sending different sensor values, we can "multiplex" the data

values. The producer would send a value, and also send an enumeration of what type of data it has sent. The consumer would block on a single queue that all the producers are writing, and then it can use a switch/case statement to handle data from multiple producers sending different kinds of values.

```
static QueueHandle_t sensor_queue;
typedef enum {
  light,
  temperature,
} sensor_type_e;
typedef struct {
  sensor_type_e sensor_type;
  int value;
} sensor_value_s;
void light_sensor_task(void *p) {
  while (1) {
    const sensor_value_s sensor_value = {light, rand()}; // Some random value
    if(xQueueSend(sensor_queue, &sensor_value, portMAX_DELAY)) {
    }
    vTaskDelay(1000);
  }
}
void temperature_sensor_task(void *p) {
  while (1) {
    const sensor_value_s sensor_value = {temperature, rand()}; // Some random value
    if(xQueueSend(sensor_queue, &temperature_value, portMAX_DELAY)) {
    }
    vTaskDelay(1000);
  }
}
void consumer_of_light_sensor(void *p) {
  sensor_value_s sensor;
  while(1) {
    xQueueReceive(sensor_queue, &sensor, portMAX_DELAY);
    switch (sensor.sensor_type) {
      case light:          printf("Light sensor value: %i\n", sensor.value);
        break;
      case temperature:    printf("Temperature sensor value: %i\n", sensor.value);
```

```
        break;
    }
  }
}
void main(void) {
   sensor_queue = xQueueCreate(3, sizeof(sensor_value_s));

   xTaskCreate(light_sensor_task, ...);
   xTaskCreate(temperature_sensor_task, ...);

   xTaskCreate(consumer_of_sensor_values, ...);}
```

Multiple Producers, 1 Consumer using QueueSet API

Before you explore this API, be sure to read FreeRTOS documentation about "Alternatives to Using Queue Sets". In general, this should be the "last resort", and you should avoid the QueueSet API if possible due to its complexity.

QueueSets is the most efficient way of blocking on multiple queues/semaphores. The `semaphore_task` is one of the producer tasks that give the semaphore at 1Hz, Queue1, and Queue2 handler in the light sensor and temperature task which randomly generates the data of light and temperature sensor. The idea of using queue sets in the consumer task is to wait on either the semaphore task or queues data.

```
static QueueHandle_t q_light_sensor, q_temperature_sensor;
static QueueSetHandle_t xQueueSet;
static SemaphoreHandle_t xSemaphore;
int generate_random_sensor_values(int lower, int upper) {
  int sensor_value = (rand() % (upper - lower + 1)) + lower;
  return sensor_value;
}
// runing @1Hz
void semaphore_task(void *p) {
  while (1) {
    vTaskDelay(1000);
    xSemaphoreGive(xSemaphore);
  }
}
```

```
// When u don't want to wait for queue/semaphore forever block on multiple queues
// Add data of light sensor to Queue1
void light_sensor_task(void *p) {
  while (1) {
    const int sensor_value = generate_random_sensor_values(10, 99);
    xQueueSend(q_light_sensor, &sensor_value, 0);
    vTaskDelay(500);
  }
}
//Add data of Temperature Sensor to Queue2
void temperature_sensor_task(void *p) {
  while (1) {
    const int sensor_value = generate_random_sensor_values(25, 85);
    xQueueSend(q_temperature_sensor, &sensor_value, 0);
    vTaskDelay(500);
  }
}
// Unblocks the task when any of the queues receives some data
void consumer(void *p) {
  int count = 0;
  int sensor_value = 0;
  while (1) {
    //xQueueSelectFromSet returns the handle of the Queue, or Semaphore that unblocked us
    QueueSetMemberHandle_t xUnBlockedMember = xQueueSelectFromSet(xQueueSet, 2000);

    if (xBlockedMember == q_light_sensor) {
      // Use zero timeout during xQueueReceive() because xQueueSelectFromSet() has
      // already informed us that there is an event on this q_light_sensor
      if (xQueueReceive(xUnBlockedMember, &sensor_value, 0)) {
        printf("Light sensor value: %i\n", sensor_value);
      }
    } else if (xBlockedMember == q_temperature_sensor) {
      if (xQueueReceive(xUnBlockedMember, &sensor_value, 0)) {
        printf("Temperature sensor value: %i\n", sensor_value);
      }
    } else if (xUnBlockedMember == xSemaphore) {
```

```
      // TODO: Do something at 1Hz, such as averaging sensor values
      if (xSemaphoreTake(xUnBlockedMember, 0)) {
        puts("----------------------------------");
        puts("1Hz signal");
      }
    } else {
      puts("Invalid Case");
    }
  }
}
int main(void) {
  srand(time(0));
  // Create an empty semaphores and the queues to add the data of queues and semaphores to the queueSe
  xSemaphore = xSemaphoreCreateBinary();
  q_light_sensor = xQueueCreate(10, sizeof(sensor_value_s));
  q_temperature_sensor = xQueueCreate(10, sizeof(sensor_value_s));
  // Make sure before creating a queue set Queues and the semaphore we want to add to the queue sets a
  // Length of the queue set is important -> Q1:10 | Q2:10 | Semaphore:1
  xQueueSet = xQueueCreateSet((10 + 10 + 1) * sizeof(int));
  // Associate the queues and the semaphore to the queue set
  xQueueAddToSet(xSemaphore, xQueueSet);
  xQueueAddToSet(q_light_sensor, xQueueSet);
  xQueueAddToSet(q_temperature_sensor, xQueueSet);
  xTaskCreate(semaphore_task, "semaphore task", 2048/(sizeof(void*)), NULL, 1, NULL);
  xTaskCreate(light_sensor_task, "light", 2048, NULL, 1, NULL);
  xTaskCreate(temperature_sensor_task, "temperature", 2048, NULL, 1, NULL);
  xTaskCreate(consumer, "single consumer", 4096, NULL, 2, NULL);
  vTaskStartScheduler();}
```

Additional Information

- Queue Management (Amazon Docs)
- Queue API (FreeRTOS Docs)

Semaphores

This article provides examples of various different Semaphores.

Binary Semaphore

A binary semaphore is a signal. In FreeRTOS, it is nothing but a queue of 1 item.

Counting Semaphore

Mutex

A mutex stands for "Mutual Exclusion".

Recursive Mutex

Tasks

Applications

Handle multiple Queues

APIs to avoid

This article lists FreeRTOS APIs that are discouraged from being used.

[warning: this article is under construction]

Just because an API exists does not mean it should be used. Different programmers have different guidelines sometimes, but ultimately it is up to you to ensure that your RTOS application is performing deterministic operations. Operations such as dynamic memory usage, dynamic task or queue usage encourages the creation of non deterministic RTOS behavior, and becomes a maintenance issue.

Fundamental Guidance

Create or allocate all resources before the RTOS starts, and after the RTOS has started, avoid any API usage that can cause the system to fail.

Avoid Deletion API

`vTaskDelete`

Theoretically, you can create a task after the RTOS is running, and while the RTOS is running, you could delete a task. In practice, however, it is discouraged to create tasks, and then to delete tasks. A task should not be created if it is going to be deleted because there are better options to handle this scenario.

Let us use a practical example. Let us assume that there is a task responsible to play an MP3 song. In place of creating a task to do this work, and then to delete the task when the playback is completed, it is better to consider alternate options:

1. Let the task sleep on a semaphore (or a Queue). Once the tasks' work is done, one can block on the semaphore or queue again
2. Let the creator of the task also process the code that is meant to run by another dynamically created task

`vQueueDelete`

Similar to why we should not create or delete a task dynamically, a queue should not be created or deleted dynamically. Fundamentally, dynamic operations such as these encourage or create non-deterministic behavior. Furthermore, depending on the RTOS configuration, the queue may utilize dynamic memory, which may exist at one time, but may not exist at another time during the RTOS runtime.

Fundamental Guidance applies here which is to avoid allocating a resource during runtime. We do not want to deal with queue handles being valid, or invalid. We do not want to run into scenarios when a task is blocked on a queue, and then the queue is deleted to abruptly "pull the plug" and to compromise our running program.

Avoid APIs that facilitate indeterministic behavior

`vTaskPrioritySet`

A task priority should be set once while creating the task, and it should never be changed again . When tasks priorities change, it is hard to diagnose issues and figure out which tasks are using the CPU at any given time because the priorities are not constant and could be changing.

Using a mutex could alter task priorities, but that is something we can let the RTOS manage by itself. Using a mutex may cause tasks to inherit and disinherit priority of another task, but the RTOS would do that under certain situations to avoid priority inversion issue. This should be considered outside of a developer's control as it is the behavior of the RTOS that is well tested.

If you run into a situation when a task needs to perform some work at a higher priority level, then you could interface to this piece of code using an RTOS construct (Queue or Semaphore), and dedicate the task for this effort and let that task have constant priority.

Avoid Task Notifications

Avoid Queue Sets

vTaskDelay

`vTaskDelay()` is a naive function, but it is important to understand how it really works.

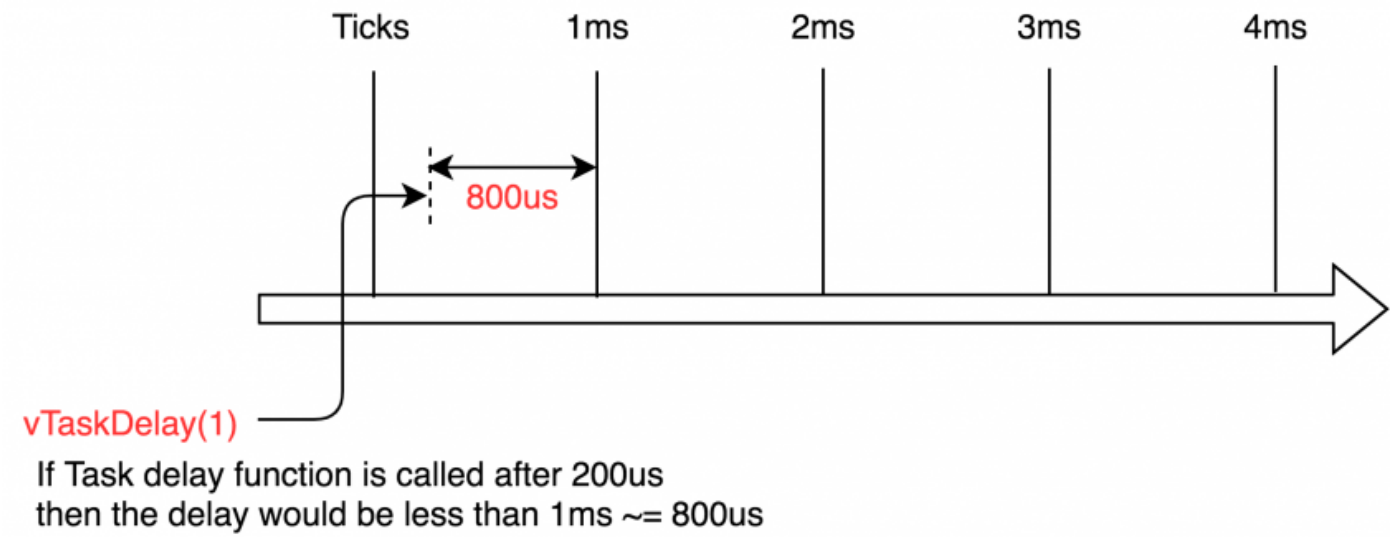
- The function expects a parameter in "tick time" (not necessarily in milliseconds).
- The function will sleep the current task for approximately the tick time.
- The function will trigger a "software interrupt" to perform a cooperative scheduling event and allow another task to run.

`vTaskDelay(1)`

For FreeRTOS configuration where 1 tick is 1 millisecond, the following is true:

- The function may sleep for less than 1 millisecond

If you use a `vTaskDelay(1)` Ideally, you will get a delay somewhere between `0ms` and `1ms`. If you ask for a 2 ticks delay, you will get a delay between 1ms and 2ms. The low end of the range happens if the `vTaskDelay()` is called right after the end of a tick as shown in the figure below.



Assuming FreeRTOS configuration where 1 tick is 1 millisecond again, the following is also true:

- The function may sleep for more than 1 millisecond

The delay of course can also be extended if the currently running task is preempted by the higher priority task during the delay and takes over the CPU for some time or it never gives up the CPU. As the delay just affect when this task is eligible to get CPU time.