Introduction to FreeRTOS

What is an RTOS?

Operating systems you're used to deal with (e.g. Linux, Windows) allow multiple programs to execute at the same time. This is called multi-tasking. In reality, each processor core can only be running a single thread of execution at any given point in time. A part of the operating system called the scheduler is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between each program.

The type of an operating system is defined by how the scheduler decides which program to run when. For example, the scheduler used in a multi user operating system (such as Unix) will ensure each user gets a fair amount of the processing time. As another example, the scheduler in a desktop operating system (such as Windows) will try and ensure the computer remains responsive to its user.

The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable (normally described as *deterministic*) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirement is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the *deadline*). A guarantee to meet real time requirements can only be made if the behavior of the operating system's scheduler can be predicted (and is therefore deterministic).

In summary, RTOS typically does not have the more advanced features that are typically found in other operating systems like Linux and Windows. The emphasis is on compactness and speed of execution.

Benefits of using real-time kernels

- Abstracting away timing information as the kernel is responsible for execution timing and provides a time-related API to the application.
- Maintainability/extensibility since tasks or modules have few interdependencies.
- Modularity since tasks are independent.
- Easier testing

Characteristics of a 'Task'

A real time application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context with no coincidental dependency on other tasks within the system or the RTOS scheduler itself. Only one task within the application can be executing at any point in time and the real time RTOS scheduler is responsible for deciding which task this should be. The RTOS scheduler may therefore repeatedly start and stop each task (swap each task in and out) as the application executes. As a task has no knowledge of the RTOS scheduler activity it is the responsibility of the real

time RTOS scheduler to ensure that the processor context (register values, stack contents, etc) when a task is swapped in is exactly that as when the same task was swapped out. To achieve this each task is provided with its own stack. When the task is swapped out the execution context is saved to the stack of that task so it can also be exactly restored when the same task is later swapped back in.

What is FreeRTOS?

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications. Traditional real time schedulers, such as the scheduler used in FreeRTOS, achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next. In FreeRTOS, a thread of execution is called a *task*.

A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM or Flash) to hold the program to be executed, and the random access memory (RAM) needed by the programs it executes. Typically, the program is executed directly from the read only memory.

Microcontrollers are used in deeply embedded applications (those applications where you never actually see the processors themselves, or the software they are running) that normally have a very specific and dedicated job to do. The size constraints, and dedicated end application nature, rarely warrant the use of a full RTOS implementation - or indeed make the use of a full RTOS implementation possible. FreeRTOS therefore provides the core real time scheduling functionality, inter-task communication, timing and synchronization primitives only. This means it is more accurately described as a real time kernel, or real time executive. Additional functionality, such as a command console interface, or networking stacks, can then be included with add-on components.

As of today, FreeRTOS supports more than 35 different architectures, including ARM, AVR, PIC (PIC18, PIC24, dsPIC and PIC32), x86. The latest release is 10.4.3 (December 14, 2020).

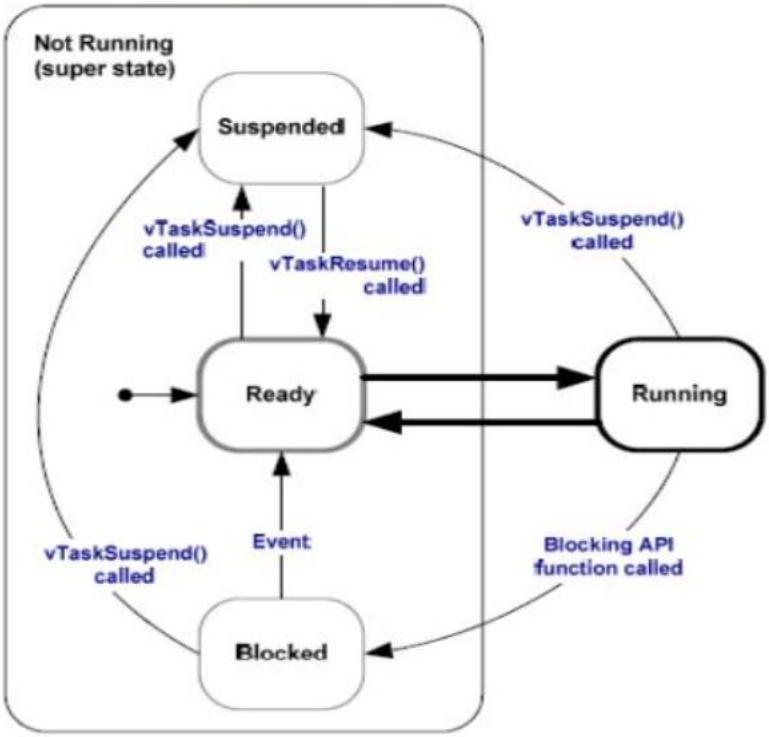
In summary, FreeRTOS is a real-time kernel/scheduler on top of which MZU applications can be built to meet their hard real-time requirements. As such, it allows MZU applications to be organized as a collection of independent threads of execution (e.g. tasks).

The FreeRTOS™ Kernel

Developed in partnership with the world's leading chip companies over a 18 year period, FreeRTOS is a market-leading real--time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. With one download every 170 seconds, FreeRTOS is built with an emphasis on reliability, accessibility, and ease of use.

FreeRTOS implements multiple threads (tasks) by having the host program call a thread tick method at regular short intervals. The thread tick method switches tasks depending on priority and a round robin scheduling scheme. The usual interval is 1 to 10 millisecond via an interrupt from a hardware timer, but this interval is often changed to suit a given application.

Full task state machine



1. Running

When a task is actually executing, it is said to be in the Running state since it is currently using the processor. If there is only a single core then there can only be one task in the Running state at any given time.

2. Ready

Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.

3. Blocked

A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.

Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

4. Suspended

Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.

The difference between the tasks in the blocked state and tasks in the suspended state is that tasks in the blocked state always have a timeout. Tasks in the suspended state are suspended indefinitely and there is no timeout.

The idle task

There is always one task that is created by the kernel and it is known as the **idle task**. The idle task will run at the lowest priority (i.e. zero priority) when no other task is ready for execution. As soon as a task is ready for execution, the idle task is preempted.

Task functions under FreeRTOS

Tasks are implemented as C-functions. The prototype must return void and take a void pointer parameter as the following:

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

Each task is a small program in its own right. It has:

- Entry point
- Normally runs forever within an infinite loop
- Has its own stack
- Does not exit.

If a task is no longer required, it should be explicitly deleted.

Example of a task function under FreeRTOS

```
void ATaskFunction(void *pvParameters)
{

/* Declare your variables here */

int var1, var2 = 5;
float var3;

  for ( ;; )
  {
```

```
    // Write the code that implements the task functionality
  }

/* If the task ever happens to leave the endless for loop before
exiting. The below NULL parameter passed to vTaskDelete()
function indicates that the task to be deleted is the calling
(this) task. */

vTaskDelete(NULL);

}
```

Task creation under FreeRTOS

`xTaskCreate()`: This call will create a task under FreeRTOS. It is called as follows:

```
portBASE_TYPE xTaskCreate (
                    pdTASK_CODE    pvTaskCode,
                    const signed char *const pcName,
                    unsigned short usStackDepth,
                    void           *pvParameters,
                    unsigned portBASE_TYPE uxPriority,
                    xTaskHandle    *pxCreatedTask)
                    )
```

- pvTaskName: Pointer to the function (just the function name) that implements the task
- pcName: A descriptive name of the task. It is not used by FreeRTOS, but 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.
  For the idle task, the size of the stack is defined by configMINIMAL_STACK_SIZE.
- pvParameters: The value assigned to the to pvParameters will be the values passed into the task.
- uxPriority: Defines the priority at which the task will execute. Priorities can be assigned from 0 (lowest priority) to (configMAX_PRIORITIES – 1) which is the highest priority. The parameter configMAX_PRIORITIES is defined in FreeRTOSConfig.h file.

- `pxCreatedTask: A handle to the created task. Will be used to refer the created task in API calls. Can be set to NULL if no one will use the task handle.`

Task return value

- pdTRUE: Task has been created successfully
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY: Task was not created due to insufficient heap memory available to allocate enough RAM to hold the task data structures and stack.

Setting up FreeRTOS on Arduino boards

You have 2 ways to setup FreeRTOS on Arduino boards:
1. From Github: https://github.com/ExploreEmbedded/Arduino_FreeRTOS

2. From Arduino library manages:
- Sketch -> include library -> Manage Libraries.
- Search for FreeRTOS library.
- Choose the right version.
- Click on install

Afterwards, when you click on Sketch -> include library, you will notice that FreeRTOS is showing up in the menu. In addition, if you go to File -> Examples -> FreeRTOS, you will see a number of example that tackle FreeRTOS.

FreeRTOS example 1: Creating tasks to blink LEDs on Arduino Uno boards

In the below example, we will create 3 tasks with the following specifications:
1. LED blinks at digital pin 8 with 200ms frequency
2. LED blinks at digital pin 7 with 300ms frequency
3. Print numbers is serial monitor with 500ms frequency.

```
#include <Arduino_FreeRTOS.h>
```

```
void TaskBlink1( void *pvParameters );
void TaskBlink2( void *pvParameters );
void Taskprint( void *pvParameters );
```

```
void setup() {
  // initialize serial communication at 9600 bits per second:

  Serial.begin(9600);

  xTaskCreate(TaskBlink1, "task1", 128, NULL, 1, NULL);

  xTaskCreate(TaskBlink2, "task2", 128, NULL, 1, NULL);

  xTaskCreate(Taskprint, "task3", 128, NULL, 1, NULL);

  vTaskStartScheduler();

}

void loop()
{
}

void TaskBlink1(void *pvParameters)  {
  pinMode(8, OUTPUT);

  while(1)
  {
    Serial.println("Task1");
    digitalWrite(8, HIGH);

    vTaskDelay( 200 / portTICK_PERIOD_MS );
    digitalWrite(8, LOW);
    vTaskDelay( 200 / portTICK_PERIOD_MS );
  }
}

void TaskBlink2(void *pvParameters)

{
  pinMode(7, OUTPUT);

  while(1)

  {
    Serial.println("Task2");
    digitalWrite(7, HIGH);
    vTaskDelay( 300 / portTICK_PERIOD_MS );
    digitalWrite(7, LOW);
    vTaskDelay( 300 / portTICK_PERIOD_MS );
  }
}

void Taskprint(void *pvParameters)  {

  int counter = 0;
```

```
  while(1)
  {
   counter++;
   Serial.println(counter);
   vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}
```

Notes:

1. Note that the loop function remains empty as we don't want manually and infinitely. The task execution is now handled by the scheduler.

2. The function `delay` must not be used as it stops the CPU and the FreeRTOS will stop working as well. Instead, we should use the `vTaskDelay` function:
`vTaskDelay( const  TickType_t xTicksDelay );`

This API delays a task for a given number of ticks. The actual time for which the task remains blocked depends on the tick rate. The constant `portTICK_PERIOD_MS` can be used to calculate real time from the tick rate.

3. For more details on the above example, visit the following link:
https://circuitdigest.com/microcontroller-projects/arduino-freertos-tutorial1-creating-freertos-task-to-blink-led-in-arduino-uno

inter-task communications techniques under FreeRTOS

FreeRTOS supports the following inter-task communication techniques:

- Queues
- Binary Semaphores
- Counting Semaphores
- Mutexes
- Recursive mutexes

FreeRTOS example 2: Communication between tasks using FreeRTOS queues

The objective of this post is to explain how to use FreeRTOS queues to achieve inter task communication, using the Arduino core. Besides communication amongst tasks, queues also allow communication between tasks and interrupt service routines.

Generically, queues can be used for a task to produce items and another to consume them, working as a FIFO (first in first out). This is what we are going to do in this example, where a task will put some integers in the queue and another will consume them. Although we are going to use integers, FreeRTOS queues allow for using more complex data structures, such as structs.

Other important behavior to consider is performing blocking API calls. Both trying to write on a full queue or read from an empty one will cause the task to block until the operation can be executed or a pre-defined amount of time expires [1]. If more than one task blocks on a queue, the highest priority task will be the one to unblock first when the operation can be completed.

```
#include <Arduino_FreeRTOS.h>

QueueHandle_t queue;
int queueSize = 10;

void setup() {

 Serial.begin(112500);
 queue = xQueueCreate( queueSize, sizeof( int ) );

 if(queue == NULL){
   Serial.println("Error creating the queue");
 }

 xTaskCreate(producerTask,     /* Task function. */
             "Producer",       /* String with name of task. */
             10000,            /* Stack size in words. */
             NULL,             /* Parameter passed as input of the task */
             1,                /* Priority of the task. */
             NULL);            /* Task handle. */

  xTaskCreate(consumerTask,     /* Task function. */
              "Consumer",       /* String with name of task. */
              10000,            /* Stack size in words. */
              NULL,             /* Parameter passed as input of the task */
              1,                /* Priority of the task. */
              NULL);            /* Task handle. */

}

void loop() {
  delay(100000);
}

void producerTask( void * parameter )
{
     for( int i = 0; I < queueSize; i++ ){
      xQueueSend(queue, &i, portMAX_DELAY);
    }

    vTaskDelete( NULL );
}

void consumerTask( void * parameter)
{
    int element;
```

```
    for( int i = 0; i < queueSize; i++ ){

        xQueueReceive(queue, &element, portMAX_DELAY);
        Serial.print(element);
        Serial.print("|");
    }

    vTaskDelete( NULL );

}
```

For more details on the above example, visit the following link: https://techtutorialsx.com/2017/09/13/esp32-arduino-communication-between-tasks-using-freertos-queues/

FreeRTOS example 3: Communication between tasks using FreeRTOS binary semaphores

In the below example, we'll use FreeRTOS semaphores to protect the serial port so that 2 tasks can share the same hardware resource.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>  // add the FreeRTOS functions for Semaphores (or Flags).

// Declare a mutex Semaphore Handle which we will use to manage
// the Serial Port.
// It will be used to ensure only one Task is accessing this
// resource at any time.
SemaphoreHandle_t xSerialSemaphore;

// define two Tasks for DigitalRead & AnalogRead
void TaskDigitalRead( void *pvParameters );
void TaskAnalogRead( void *pvParameters );

// the setup function runs once when you press reset or power the board
void setup()
{
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

  // Semaphores are useful to stop a Task proceeding, where it
  // should be paused to wait,
  // because it is sharing a resource, such as the Serial port.
```

```
  // Semaphores should only be used whilst the scheduler is
  // running, but we can set it up here.

  // Check to confirm that the Serial Semaphore has not already
  // been created.

  if ( xSerialSemaphore == NULL )
  {
    // Create a mutex semaphore we will use to manage the Serial Port
    xSerialSemaphore = xSemaphoreCreateMutex();

    if ( ( xSerialSemaphore ) != NULL )

      // Make the Serial Port available for use, by "Giving"
      // the Semaphore.
      xSemaphoreGive( ( xSerialSemaphore ) );
  }

  // Now set up two Tasks to run independently.
  xTaskCreate(
      TaskDigitalRead,
      (const portCHAR *)"DigitalRead",  // A name just for humans
      128,  // This is the stack size
      NULL,
      2,  // Priority, with 1 being the highest, and 4 the lowest.
      NULL );

  xTaskCreate(
    TaskAnalogRead,
    (const portCHAR *) "AnalogRead",
    128,  // Stack size
    NULL,
    1,
    NULL );

  // Now the Task scheduler, which takes over control of
  // scheduling individual Tasks, is automatically started.
}

void loop()
{
  // Empty. Things are done in Tasks.
}
```

```
/*--------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*--------------------------------------------------*/

void TaskDigitalRead( void *pvParameters __attribute__((unused)) )  //
This is a Task.
{
  // DigitalReadSerial
  // Reads a digital input on pin 2, prints the result to the
  // serial monitor

  // This example code is in the public domain.

  // digital pin 2 has a pushbutton attached to it. Give it a name:
  uint8_t pushButton = 2;

  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);

  for (;;) // A Task shall never return or exit.
  {
    // read the input pin:
    int buttonState = digitalRead(pushButton);

    // See if we can obtain or "Take" the Serial Semaphore.
    // If the semaphore is not available, wait 5 ticks of the
    // Scheduler to see if it becomes free.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE
)
    {
      // We were able to obtain or "Take" the semaphore and can
      // now access the shared resource.
      // We want to have the Serial Port for us alone, as it takes
      // some time to print,
      // so we don't want it getting stolen during the middle of
      // a conversion.
      // print out the state of the button:
      Serial.println(buttonState);

      // Now free or "Give" the Serial Port for others.
      xSemaphoreGive( xSerialSemaphore );
    }
```

```
    // one tick delay (15ms) in between reads for stability
    vTaskDelay(1);
  }
}

void TaskAnalogRead( void *pvParameters __attribute__((unused)) )
{
  for (;;)
  {
    // read the input on analog pin 0:
    int sensorValue = analogRead(A0);

    // See if we can obtain or "Take" the Serial Semaphore.
    // If the semaphore is not available, wait 5 ticks of the
    // Scheduler to see if it becomes free.
    if ( xSemaphoreTake( xSerialSemaphore, ( TickType_t ) 5 ) == pdTRUE
)
    {
      // We were able to obtain or "Take" the semaphore and can
      // now access the shared resource.
      // We want to have the Serial Port for us alone, as it takes
      // some time to print,
      // so we don't want it getting stolen during the middle of
      // a conversion.
      // print out the value you read:
      Serial.println(sensorValue);

      // Now free or "Give" the Serial Port for others.
      xSemaphoreGive( xSerialSemaphore );
    }

    // one tick delay (15ms) in between reads for stability
    vTaskDelay(1);
  }
}
```

For more details on the above example, visit the following link:
https://create.arduino.cc/projecthub/feilipu/using-freertos-semaphores-in-arduino-ide-b3cd6c

You can also have a look at the following link:
https://microcontrollerslab.com/freertos-binary-semaphore-tasks-interrupt-synchronization-u-arduino/

FreeRTOS example 4: Communication between tasks using FreeRTOS counting semaphores

The counting semaphores can be used to either count events or manage resources. In the below example, we'll create 2 tasks named Task 1 and Task 2 successively. Both tasks will use the same UART module of Arduino to write data on the serial monitor. If we consider the Arduino serial monitor and UART communication module as a resource that will be utilized by both tasks, then, we can use counting semaphore for this resource management for Task1 and Task2. Because only one task can hold these Arduino resources at a time.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

SemaphoreHandle_t xCountingSemaphore;

void setup()
{
  Serial.begin(9600); // Enable serial communication library.
  pinMode(LED_BUILTIN, OUTPUT);

  // Create task for Arduino led

  xTaskCreate(Task1, "Ledon", 128, NULL, 0 , NULL );

  xTaskCreate(Task2, "Ledoff", 128, NULL, 0, NULL );

  xCountingSemaphore = xSemaphoreCreateCounting(1,1);

  xSemaphoreGive(xCountingSemaphore);
}

void loop() {}

void Task1(void *pvParameters)
{
  (void) pvParameters;

  for (;;)
    {
      xSemaphoreTake(xCountingSemaphore, portMAX_DELAY);
      Serial.println("Inside Task1 and Serial monitor Resource
Taken");
      digitalWrite(LED_BUILTIN, HIGH);
```

```
xSemaphoreGive(xCountingSemaphore);
      vTaskDelay(1);
   }
}

void Task2(void *pvParameters)
{
  (void) pvParameters;

  for (;;)
    {
      xSemaphoreTake(xCountingSemaphore, portMAX_DELAY);
      Serial.println("Inside Task2 and Serial monitor Resource
Taken");
      digitalWrite(LED_BUILTIN, LOW);
      xSemaphoreGive(xCountingSemaphore); vTaskDelay(1);

    }
}
```

For more details on the above example, visit the following link:
https://microcontrollerslab.com/freertos-counting-semaphore-examples-arduino/