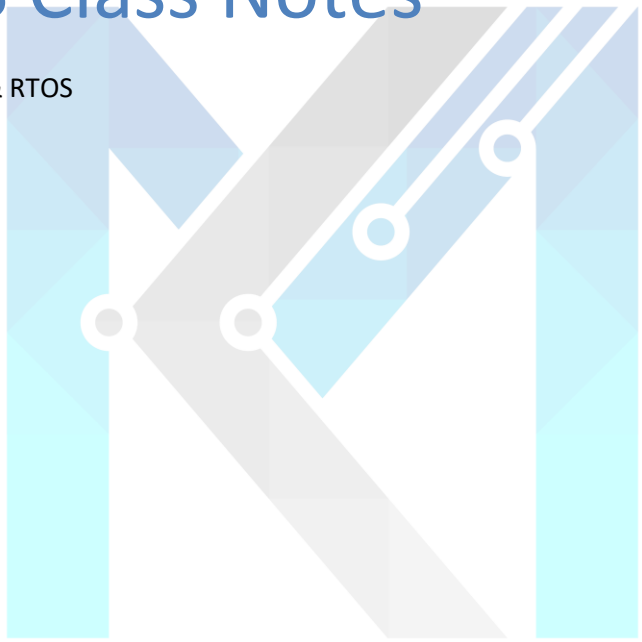
KERNEL MASTERS

Kernel Masters

RTOS Class Notes

Embedded C & RTOS



Kernel Masters
2/27/2022

Contents

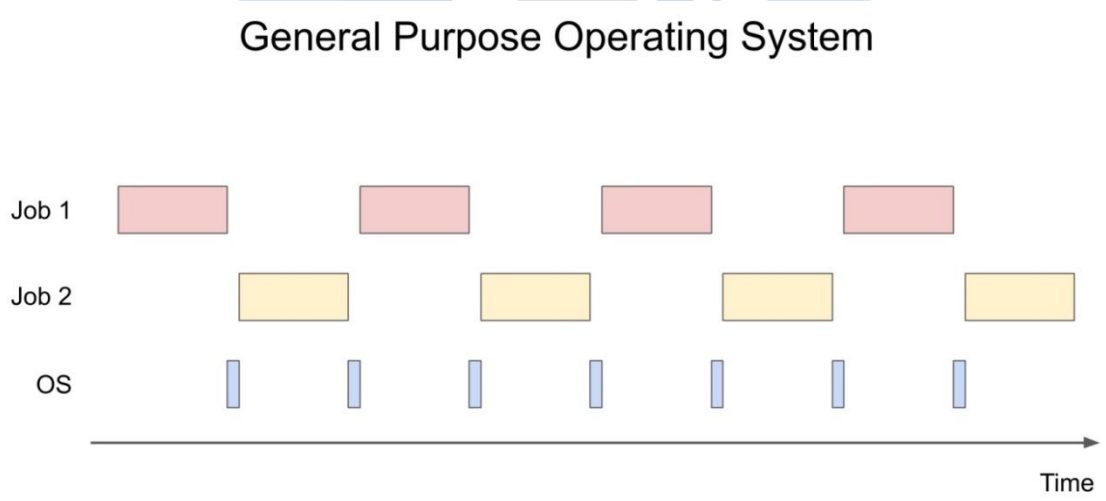
Introduction to Real Time Operating System

1. What is an Operating System?

An operating system (OS) is a piece of software that manages other software and hardware resources in a computer system. You are probably familiar with most of the popular general purpose operating systems, such as Windows, macOS, Linux, iOS, and Android. A general purpose OS is normally designed with a focus on user experience.

For example, let's say we're developing an application on an operating system for a phone, like Android or iOS. A user might want to stream a movie, so we can break that streaming experience into two jobs: downloading chunks of video from the Internet (job 1) and displaying each chunk to the user (job 2). These jobs might be part of the same program, in which case, they might be implemented as concurrently running threads.

If our processor only has 1 core available, our streaming application might need to jump between job 1 and job 2 rapidly to give the user the impression that downloading and viewing is happening at the same time.



Note that I've listed a third job here: the underlying OS software. The OS is in charge of figuring out which job needs to run in order to give the user a seamless experience. This swapping of jobs is known as context switching and incurs some processing overhead as the OS needs to save things like registers, memory, and program counter so that it can return to a previously running thread without losing any information.

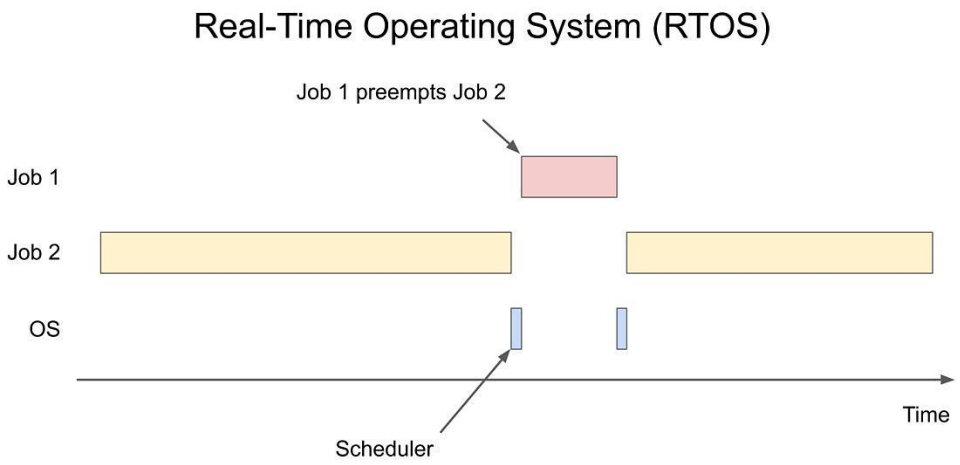
A job might be a little late, a packet might be lost, or a frame might be dropped so that the video can catch up. Since the focus is on user experience in a general purpose OS, these minor variances in job execution will likely go unnoticed by the user, or the user might not care (too much).

The focus changes in most microcontroller applications. Often, meeting strict timing deadlines is critically important. Think about a motor controller or an automatic braking system in an electric vehicle.

In these cases, if timing is off by even a few milliseconds, human lives could be in danger. As a result, an RTOS would be your best bet to manage several jobs running concurrently.

In this second example, a vehicle's electronic control unit (ECU) might be in charge of controlling and assisting with braking. Job 2 monitors the driver's input and helps apply the brakes and turns on the tail lights. However, let's say that our ECU gets notification that the car's sensors have detected an impending crash. As a result, job 1 will preempt job 2 to control the brakes. This all assumes, of course, that this is a vehicle equipped with automatic braking assist.

Keep in mind this is a fairly simple example. Hardware interrupts can also be used to preempt running tasks. An RTOS allows you to create software jobs instead of relying on hardware interrupts and assign them priorities. Additionally, most RTOSes can also act as an abstraction layer, allowing you to write code that can be easily ported to other microcontrollers.



2. What is Real Time Operating System?

A Real-Time Operating System (RTOS) is a lightweight OS used to ease multitasking and task integration in resource and time constrained designs, which is normally the case in embedded systems. Besides, the term "real-time" indicates predictability/determinism in execution time rather than raw speed, hence an RTOS can usually be proven to satisfy hard real-time requirements due to its determinism.

A real-time operating system(RTOS) is an operating system(OS) intended to serve real-time applications that process data as it comes in, typically without buffer delays, which is a time bound system which has well defined fixed time constraints. Process in the RTOS must be done within the defined constraints; otherwise, the system will fail. Most of the operating systems are multi-tasking, which allow more than one program to be running in concurrency. This is achieved by time-sharing, where the available processor time is divided into multiple processes. These processes are each interrupted repeatedly in time slices by a task-scheduling subsystem of the operating system.

3. Tasks

Tasks (could also be called processes/threads) are independent functions running in infinite loops, usually each responsible for one feature. Tasks are running independently in their own time (temporal isolation) and memory stack (spatial isolation). Spatial isolation between tasks can be guaranteed with the use of a hardware memory protection unit (MPU), which restricts accessible memory region and triggers fault exceptions on access violation. Normally, internal peripherals are memory-mapped, so an MPU can be used to restrict access to peripherals as well.

Tasks can be in different states:

- Blocked – task is waiting for an event (e.g. delay timeout, availability of data/resources)
- Ready – task is ready to run on CPU but not running because CPU is in use by another task
- Running – task is assigned to be running on CPU

Most tasks are blocked or ready most of the time because generally only one task can run at a time in one CPU. Since tasks are scheduled in turn in a small time slice, it seems that there are multiple tasks run simultaneously.

4. Scheduler

Schedulers in RTOS control which task to run on the CPU, and different scheduling algorithms are available. Normally they are:

- Pre-emptive – task execution can be interrupted if another task with higher priority is ready
- Co-operative – task switch will only happen if the current running task yields itself

Pre-emptive scheduling allows higher priority tasks to interrupt a lower task in order to fulfill real-time constraints, but it comes in the cost of more overhead in context switching.

5. Inter-task communication (ITC)

Multiple tasks will normally need to share information or events with each other. The simplest way to share is to directly read/write shared global variables in RAM, but this is undesirable due to risk of data corruption caused by a race condition. A better way is to read/write file-scoped static variables accessible by setter and getter functions, and race conditions can be prevented by disabling interrupts or using a mutual exclusion object (mutex) inside the setter/getter function. The cleaner way is using thread-safe RTOS objects like message queue to pass information between tasks.

Besides sharing of information, RTOS objects are also able to synchronize task execution because tasks can be blocked to wait for availability of RTOS objects. Most RTOS have objects such as:

- Message queue
    - First-in-first-out (FIFO) queue to pass data
    - Data can be sent by copy or by reference (pointer)
    - Used to send data between tasks or between interrupt and task
- Semaphore

    - Can be treated as a reference counter to record availability of a particular resource
    - Can be a binary or counting semaphore
    - Used to guard usage of resources or synchronize task execution
- Mutex
    - Similar to binary semaphore, generally used to guard usage of a single resource (MUTual EXclusion)
    - FreeRTOS mutex comes with a priority inheritance mechanism to avoid priority inversion (condition when high priority task ends up waiting for lower priority task) problem.
- Mailbox
    - Simple storage location to share a single variable
    - Can be considered as a single element queue
- Event Group
    - Group of conditions (availability of semaphore, queue, event flag, etc.)
    - Task can be blocked and can wait for a specific combination condition to be fulfilled
    - Available in Zephyr as a Polling API, in FreeRTOS as QueueSets

6. System Tick

RTOS need a time base to measure time, normally in the form of a system tick counter variable incremented in a periodic hardware timer interrupt. With system tick, an application can maintain more than time-based services (task executing interval, wait timeout, time slicing) using just a single hardware timer. However, a higher tick rate will only increase the RTOS time base resolution, it will not make the software run faster.

Why use RTOS

1. Organization

Applications can always be written in a bare metal way, but as the code complexity increases, having some kind of structure will help in managing different parts of the application, keeping them separated. Moreover, with a structured way of development and familiar design language, a new team member can understand the code and start contributing faster. RFCOM Technologies has developed applications using different microcontrollers like Texas Instruments' Hercules, Renesas' RL78 and RX, and STMicroelectronics' STM32 on a different RTOS. Similar design patterns allow us to develop applications on different microcontrollers and even a different RTOS.

2. Modularity

Divide and conquer. By separating features in different tasks, new features can be added easily without breaking other features; provided that the new feature does not overload shared resources like the CPU and peripherals. Development without RTOS will normally be in a big infinite loop where all features are part of the loop. A change to any feature within the loop will have an impact on other features, making the software hard to modify and maintain.

3. Communication stacks and drivers

Many extra drivers or stacks like TCP/IP, USB, BLE stacks, and graphics libraries are developed/ported for/to existing RTOSs. An application developer can focus on an application layer of the software and reduce time to market significantly.

FREE RTOS vs CMSIS-RTOS:

1. FREE RTOS

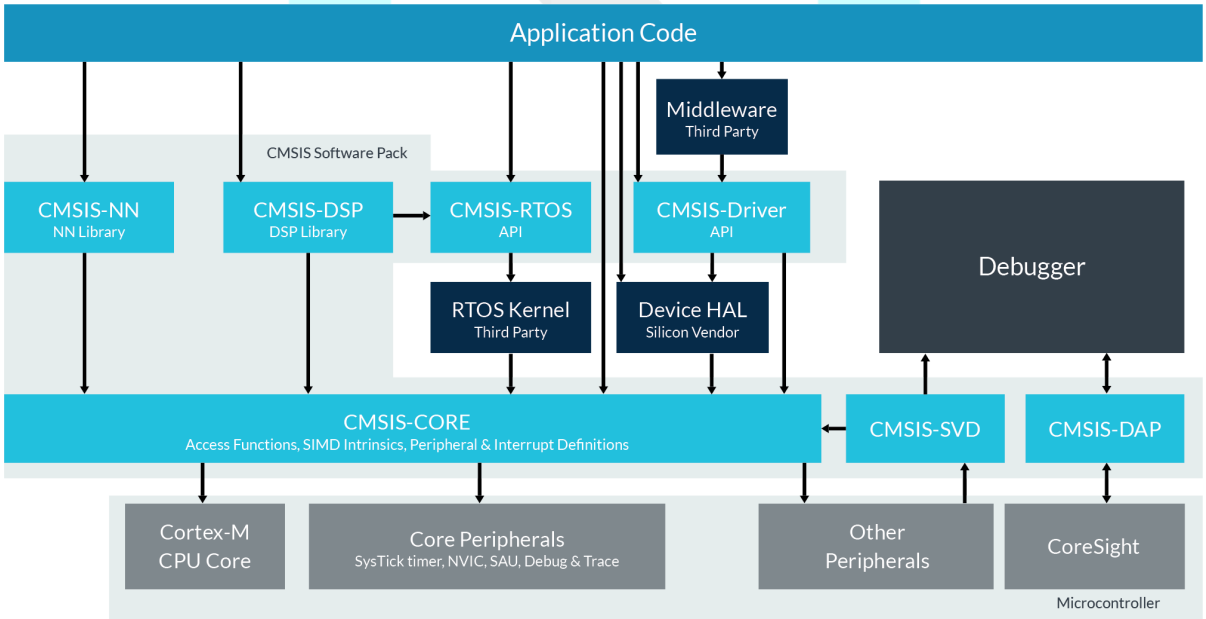
FreeRTOS is a free, open source RTOS for embedded devices, which has been ported to 35 microcontroller platforms. For embedded devices, there are many RTOS, like µC/OS-Ⅱ and WinCE. Alibaba, Tencent and Huawei also have their own RTOS, which is build for their own IoT cloud platforms. We choose FreeRTOS is because it is integrated into the STM32CubeIDE, which is easy to configure.

It's important to understand how STM32CubeIDE has bundled FreeRTOS. While FreeRTOS is an underlying software framework that allows for switching tasks, scheduling, etc., we won't be making calls to FreeRTOS directly.

2. CMSIS-RTOS

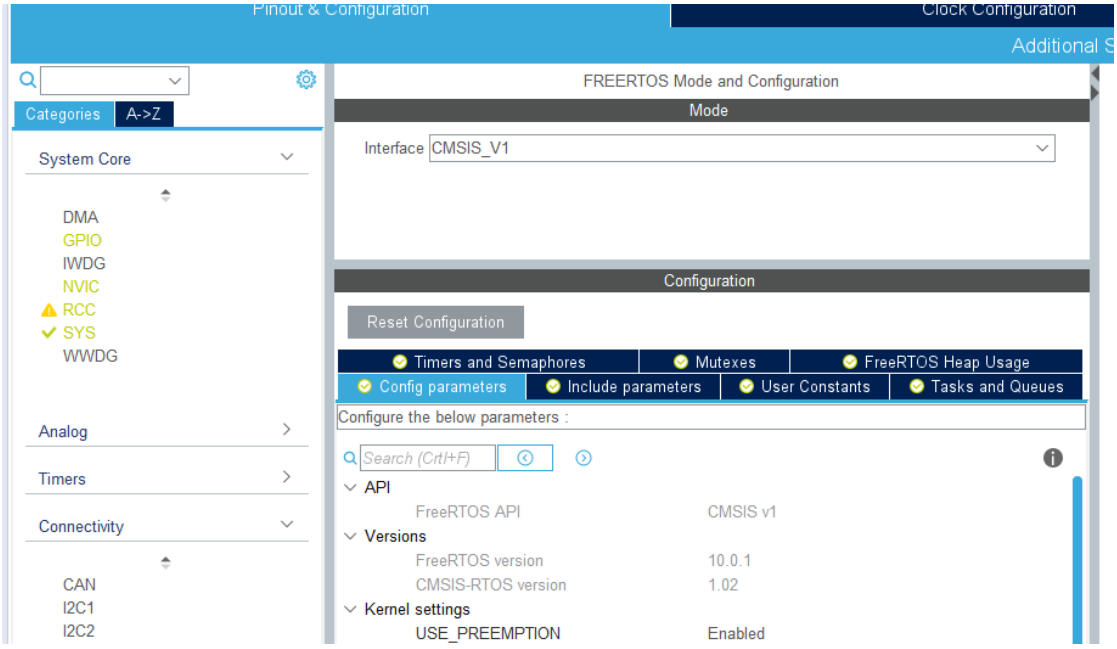
ARM has created the CMSIS-RTOS library, which allows us to make calls to an underlying RTOS, thus improving the portability of code among various ARM processors. This image describes how ARM's CMSIS libraries interact with third party software:

FreeRTOS is our "RTOS Kernel" as depicted in the diagram. We will be making calls to CMSIS-RTOS (version 2, specifically) in order to control the underlying FreeRTOS.



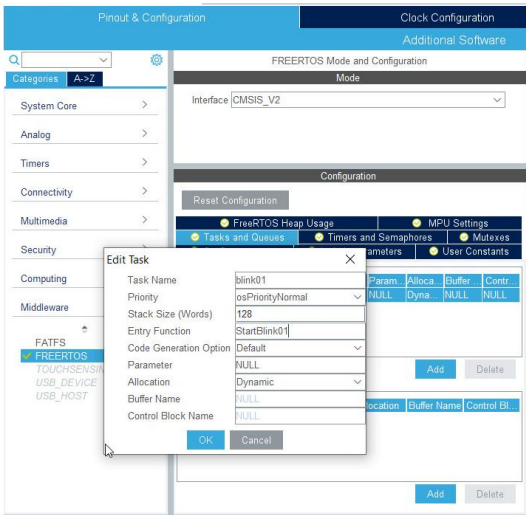
3. Enable FreeRTOS in STM32CubeIDE

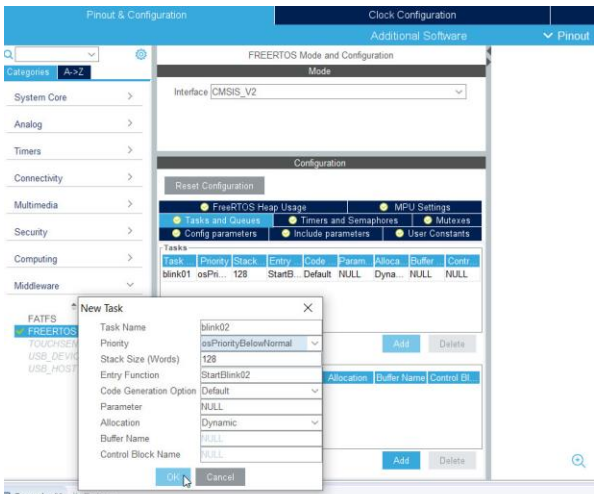
In CubeMX, go to Categories > Middleware > FREERTOS. Under Mode, change Interface to CMSIS_V2.



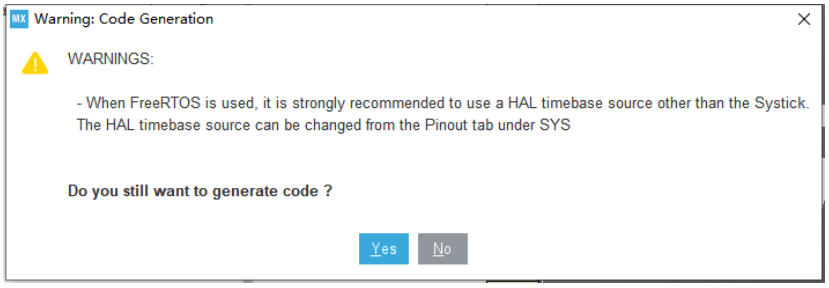
4. Create Task

In the *Configuration* pane, under *Tasks and Queues*, double-click on the default task to make changes. Change the *Task Name* to **blink01** and change the Entry Function to **StartBlink01**.





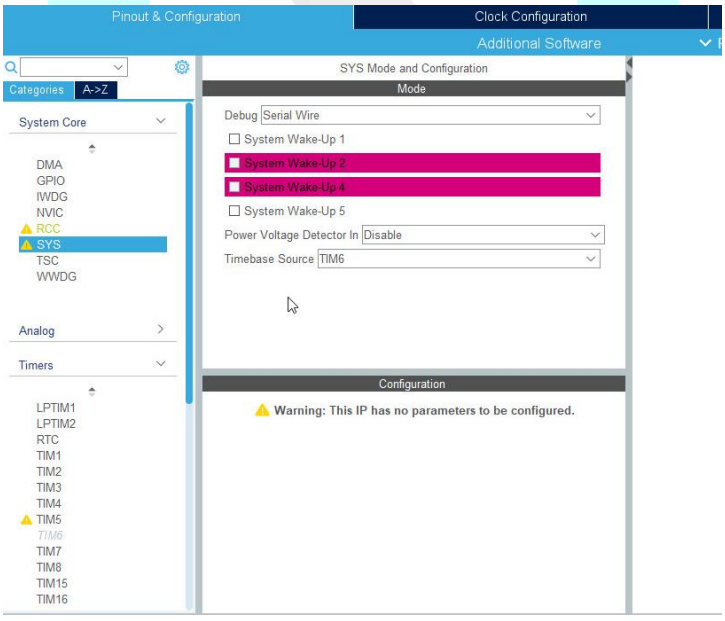
Click **OK** and click **Add** to create a new task. Change the task name to **blink02**, set the priority to **osPriorityBelowNormal**, and change the entry function name to **StartBlink02**. Note that the priority of task 2 is below that of task 1, which means that in the event of blink01 and blink02 trying to run concurrently, blink01 will take priority.
It seems everything is done. Let's try to generate the code:



5. Choose System Tick Timer

SysTick is a special timer in most ARM processors that's generally reserved for operating system purposes. By default, SysTick in an STM32 will trigger an interrupt every 1 ms. If we're using the STM32 HAL, by default, SysTick will be used for things like HAL_Delay() and HAL_GetTick(). As a result, the STM32 HAL framework gives SysTick a very high priority. However, FreeRTOS needs SysTick for its scheduler, and it requires SysTick to be a much lower priority.

We can fix this conflict in a few ways, but the easiest is to assign another, unused timer as the timebase source for HAL. Timers 6 and 7 in most STM32 microcontrollers are usually very basic, which makes them perfect as a timebase for HAL. Go to System Core > SYS and under Mode, change the Timebase Source to TIM6.



Save and generate the code.

6. Write Concurrent Threads

What we referred to as “jobs” earlier could theoretically be anything that accomplishes our desired task: separate programs, threads, etc. You should keep a few terminology differences in mind when working with FreeRTOS.
A “task” in FreeRTOS is a piece of a program that can run concurrently with other pieces in the same program. It’s analogous to “threads,” if you’ve done other concurrent programming. However, note that CMSIS-RTOS, the abstraction layer for our RTOS, refers to these concurrent pieces as “threads.” As a result, you might see them used interchangeably throughout the program, even if they don’t quite mean the same thing.

Hello World Embedded C Program without RTOS

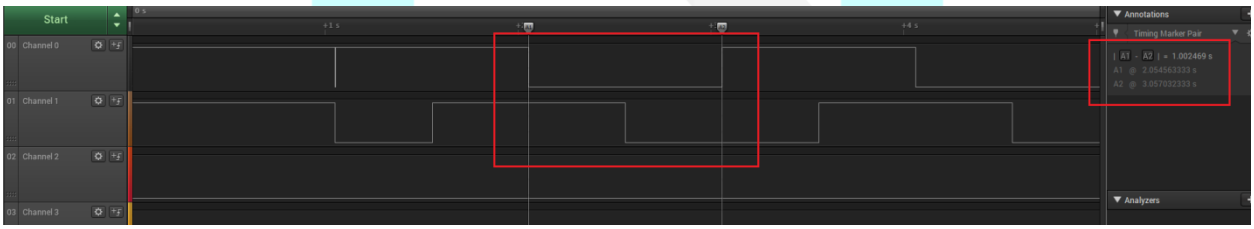
Task1 is toggle RED LED every 500msec and Task2 is toggle GREEN LED every 500msec. Here both tasks can’t execute at a time.
In Super loop cannot execute more than one task at a time.

Toggle RED and GREEN LED’s one after another one with 500 msec delay. Here both LED’s can’t toggle at a time.

```
 while(1)
 {
   HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_13); // RED LED
   HAL_Delay(500);
   HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14); // GREEN LED
   HAL_Delay(500);
}
```

Waveform:

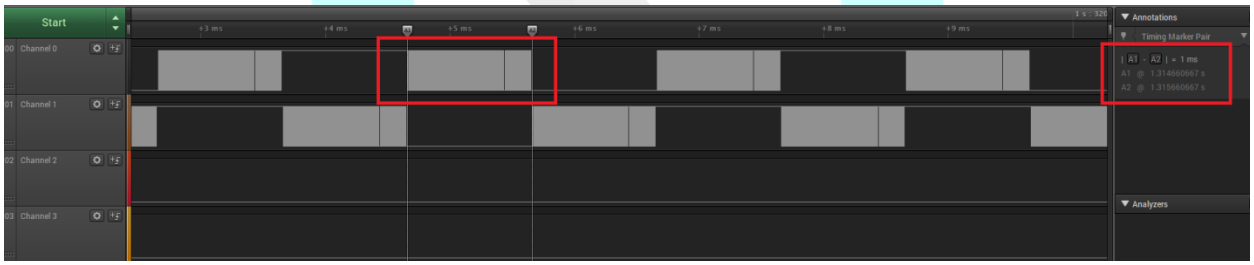
Within one sec time period toggle RED & GREEN LED 500msec delay one after another.



How to measure Timelslice (or) Tick Rate?

- In RTOS Tick Rate decides the time slice between one to another task.
- In FREE RTOS Kernel configuration option is TICK_RATE_HZ=1000Hz means t=1/f = 1/1000 = 1msec Delay.
- *Task1* & *Task2* run continuously (Blink LED's Contionously) without any delay. Whenever *Task1* excecue 1msec duration (reach timeslice) than scheduler algorithm preempt running *Task1* and scheldle to *Task2*.

```
void StartTask1(void const * argument)
{
 for(;;)
 {
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_13);
 }
}

void StartTask02(void const * argument)
{
 for(;;)
 {
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
 }
}
```



Hello World RTOS Program

Write an Embedded C Program to create two threads and add toggle LED functions in both thread functions and toggle both LED's at the same time.

In the beginning of main(), you should see the threads being defined for us, as set up by CubeMX. Notice that we pass our entry function names into the osThreadNew() function, which will call these functions as soon as we call osKernelStart(). Once osKernelStart() has been called, we do not want to have any code after it in main(), as the program should, ideally, never return from osKernelStart().

At this point, our threads should be running simultaneously, with their own setup code and forever while loops. There is also a background scheduler task that runs, which is in charge of switching context between our threads.

StartBlink01() and StartBlink02() are our threads. Each has its own forever loops, and they should run concurrently. While they can't take up the same space and time in our single-core processor, the scheduler will switch them in and out to give the appearance that we're running 2 threads at the same time.
Note that instead of HAL_Delay(), we need to use osDelay(). HAL_Delay() in a high priority task might hog the processor, preventing a context switch. It also prevents the scheduler from idling, which could save on power. So, we use osDelay() to tell the scheduler that it's OK to switch to a different task while we wait.

We are using different delay times for blink01 and blink02 in order to have the two tasks fight over toggling the LED.

Test It!
Build the project and start a debugging session. You should see the LD2 LED blinking on and off, but at a varying duty cycle that increases or decreases in distinct steps. This is a result of the two threads with different wait times.

OS_Delay() vs HAL_Delay()

osDelay is used to block the task, and execute the task in 500 milliseconds. It acts like HAL_Delay but they are different. When HAL_Delay is called, CPU cannot do anything except waiting, while osDelay just suspend the current task, and CPU can execute other task when the pervious task is blocked

Case 1: osDelay() Example:

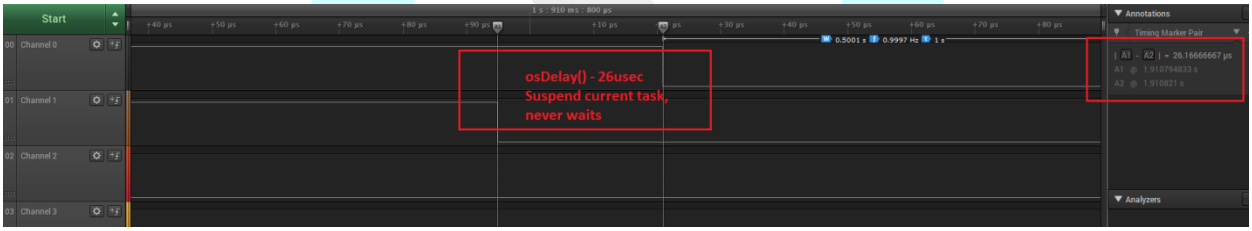
osDelay just suspend the current task, and CPU can execute other task when the pervious task is blocked.

TICK_RATE_HZ=1000Hz means t=1/f = 1/1000 = 1msec Delay; So Time slice is 1msec.

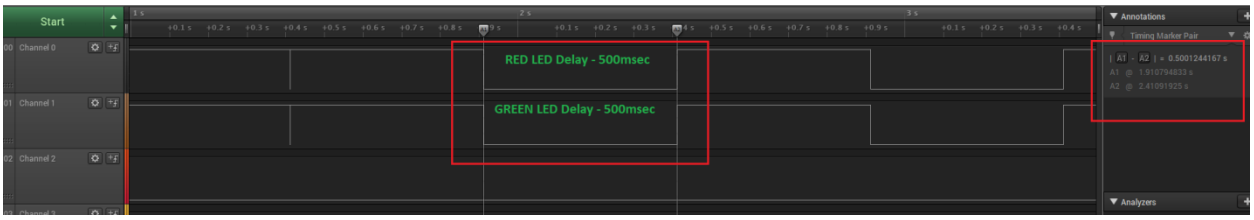
```
void StartRED_LED(void const * argument)
{
  for(;;)
  {
   HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_13);
    osDelay(500);
    }
}

void StartGREEN_LED(void const * argument)
{
 for(;;)
 {
   HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
   osDelay(500);
 }
}
```

Waveforms:

osDelay in between Task1 & Task2



RED & GREEN LED Delays:



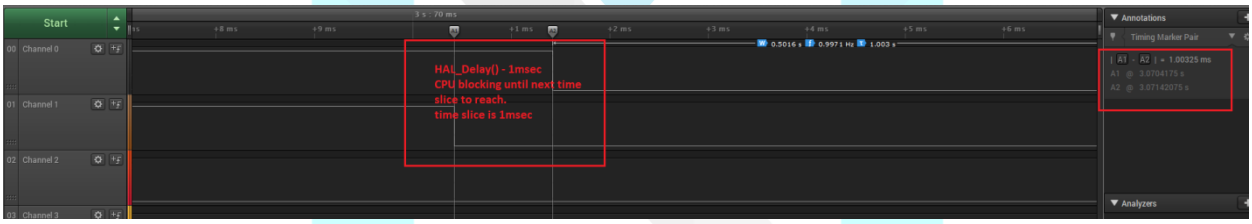
Case 2: HAL_Delay() Example:

When HAL_Delay is called, CPU cannot do anything except waiting.

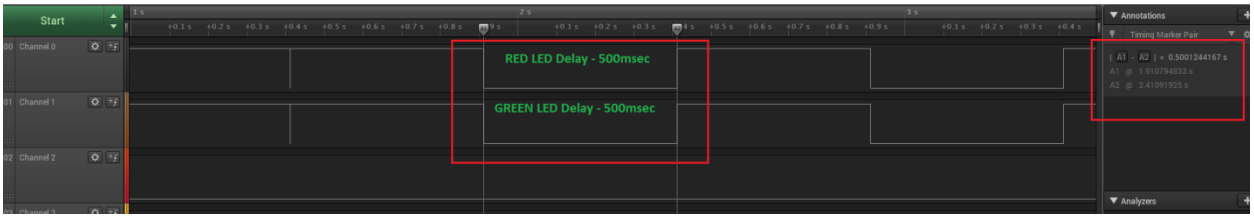
TICK_RATE_HZ=1000Hz means t=1/f = 1/1000 = 1msec Delay; So Time slice is 1msec.

```
void StartRED_LED(void const * argument)
{

 for(;;)
 {
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_13);
   HAL_Delay(500);
   }
}

void StartGREEN_LED(void const * argument)
{
 for(;;)
 {
   HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
   HAL_Delay(500);
 }
}
```

osDelay in between Task1 & Task2



RED & GREEN LED Delays:



Reference Links:

- FreeRTOS API Reference: https://www.freertos.org/a00106.html
- CMSIS-RTOS API Reference: http://www.keil.com/pack/doc/CMSIS/RTOS/html/group__CMSIS__RTOS.html