RTOS Overview

Using RTOS on MCU is method to deal with concurrent tasks which need to be handled in real-time without delay. A task is a piece of code that can be scheduled by OS scheduler and dedicated for a specific functionality. Tasks can have different priorities to be run in an order.

Last update: 2021-07-08 22:14:48

Table of Content

1. RTOS

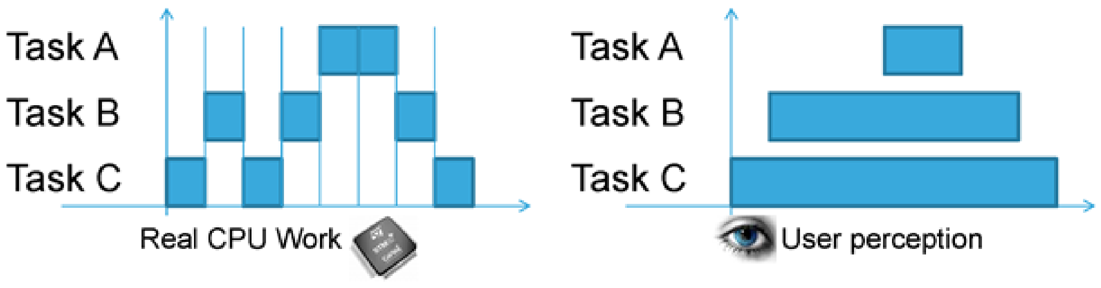
FreeRTOS Quick Start Guide

RTOS stands for Real Time Operating System. And as the name suggests, it is capable of doing tasks, as an operating system does. The main purpose of an OS is to have the functionality, of running multiple tasks at the same time, which obviously isn't possible with bare metal.

The core of an RTOS is an advanced algorithm for scheduling, with the key factors are minimal interrupt latency and minimal thread switching latency. A real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

Refer to the comparison table of RTOSs.

Kernel is the main core of an RTOS which manages tasks, memory, hardware access. The goal of a kernel is to make task runs concurrently in user point of view. In underlying works, kernel run tasks one by one, each task can run in some milliseconds and pause, leave CPU and hardware for other tasks.



Task Execution

The core of any preemptively multitasks system is context switching, in which a task can be halted, its context saved, and then later be restored, allowing it to continue execution. The context is defined primarily as the task's stack and the state of the processor registers.

RTOS still is a normal C program

Even the name RTOS is an Operating System, it is still a part of a single C program which starts from the only one `main` function. The only interesting point is that RTOS has a magic scheduler to switching tasks (loops).

1.1. A Task

A task will do a specific functionality, such as toggling an LED, reading an input. The task function usually is in infinite loop, it means a task will continuously run and never returns.

The Task Function is declared as:

```
void taskFunctionName(void* argument) {
    for(;;) {
        // do things over and over
    }
}
```

In freeRTOS, every task has its own stack that stores TCB (Task Control Block) and other stack-related operations while the task is being executed. It also stores processor context before a context switch (switching to other task). Stack size must be sufficient to accommodate all local variables and processor context.

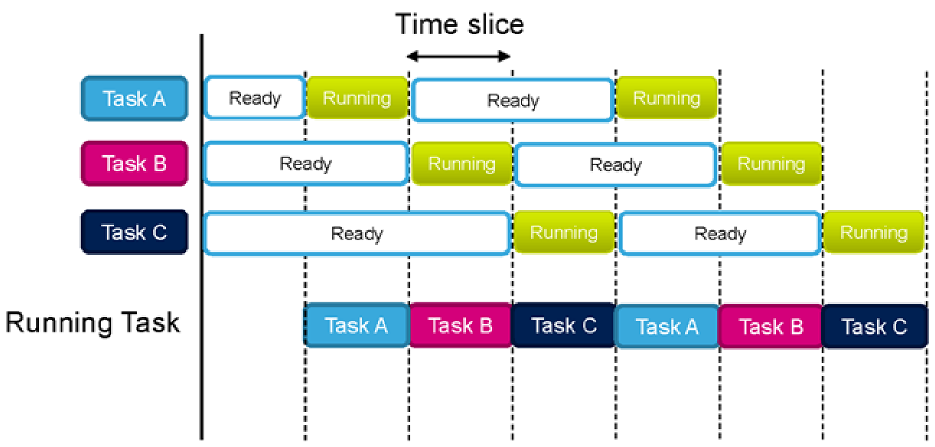
A Task has 4 states:

- **inactive**: not to be run
- **ready**: in queue to be run
- **running**: is being executed
- **waiting/blocked**: is paused, put in run queue, but not to be run in next time slot

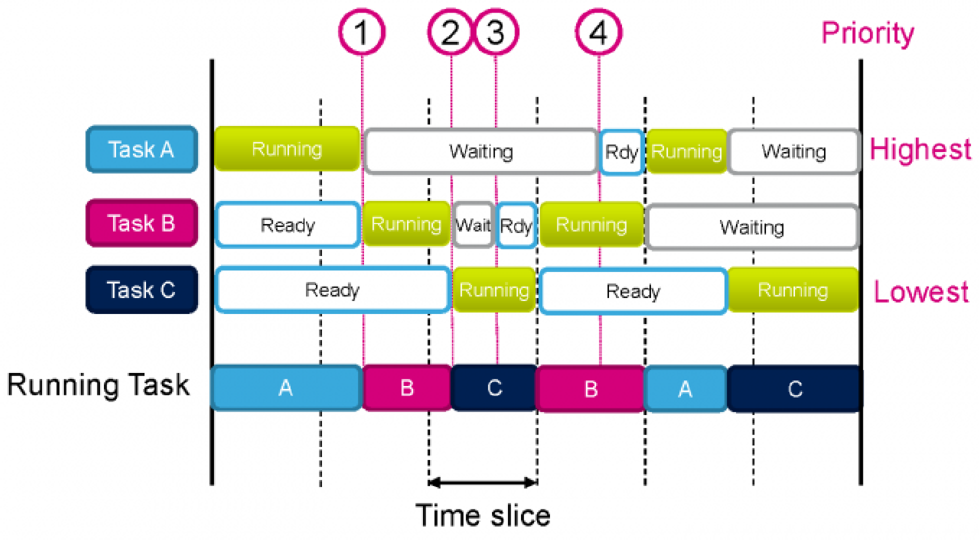
1.2. The Scheduler

This part of kernel decides which task will be run next. There are some rules to pick a task:

- **Cooperative**: task by task, each task does its work until it finishes
- **Round-robin**: each task has a time slice to run, there is no priority for task execution
- **Priority-based**: task has priority which has high number can interrupt the running task and takes place of execution



Round-robin scheduler



Priority-based scheduler

1.3. The SysTick

SysTick is apart of the ARM Core, that counts down from the reload value to zero, and fire an interrupt to make a periodical event. SysTick is mainly used for delay function in non-RTOS firmware, and is used as the interrupt for RTOS scheduler.

SysTick is also used as countable time span of a waiting task. For example, a task need to read an input, and it should wait for 50 ms, if nothing comes, task should move to other work. This task will use SysTick , which is fired every 1 ms, to count up a waiting counter, if the counter reaches 50 ticks, task quits the waiting loop and runs other code.

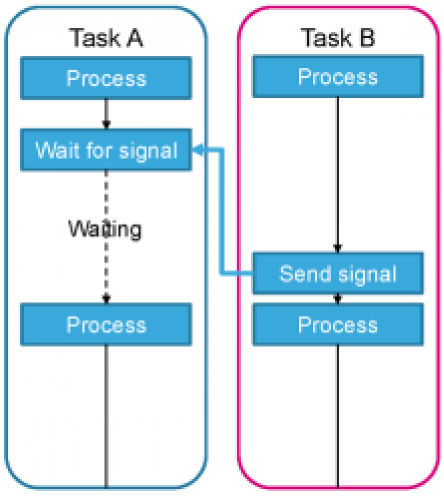
Read more about setting up SysTick and Delay.

1.4. Shared Memory

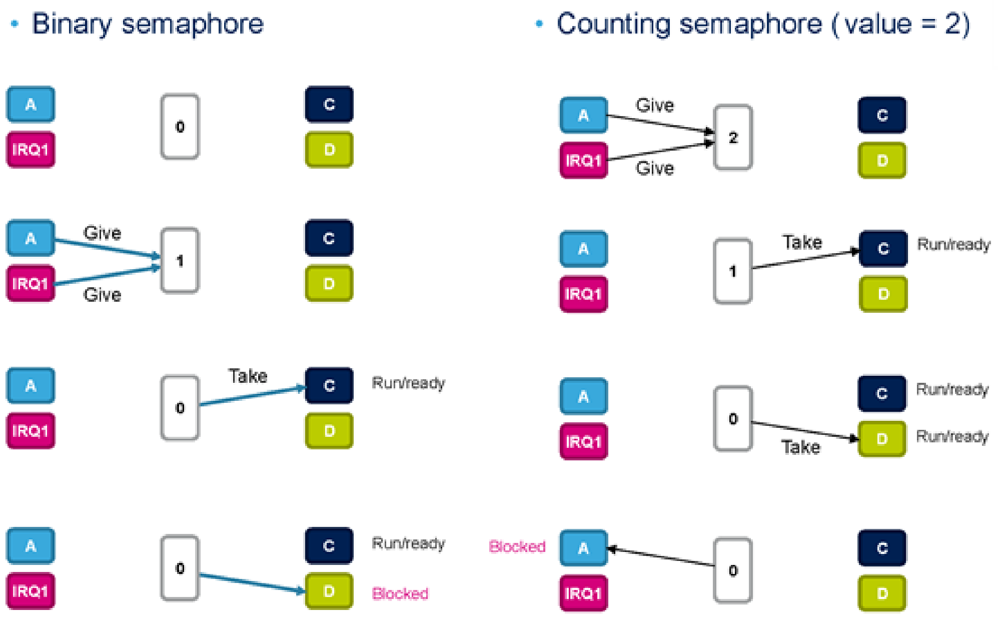
Tasks are usually a work to do in a loop and it thinks it can control all of resource. In a system, there are many tasks run together, and in many cases, they works with condition from others.

Inter-task communication is defined as some type:

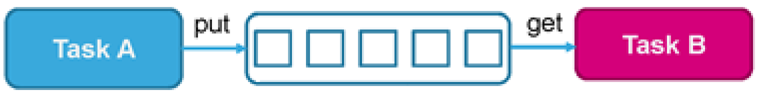
- Signal: tell other task to start doing somethinng, to synchronize tasks
- Message Queue/Mailbox: send data between tasks
- Mutex/Semaphore: synchronize acess to a shared resource, lock resource which is in-use



Signal between tasks



Shared resource between tasks



Queue between tasks

2. Lab 0: Create simple tasks

Assume that an application intend to toggle two LEDs at 1 Second and 2 Second intervals respectively. Below is a bare-metal approach (without timers) of doing it:

```
int main() {
    while(1) {
        LED1_TURN_ON();
            LED2_TURN_ON();
        delay_seconds(1);

        LED1_TURN_OFF();
        delay_seconds(1);

        LED1_TURN_ON();
            LED2_TURN_OFF();
        delay_seconds(1);

        LED1_TURN_OFF();
        delay_seconds(1);
    }
    return 0;
}
```

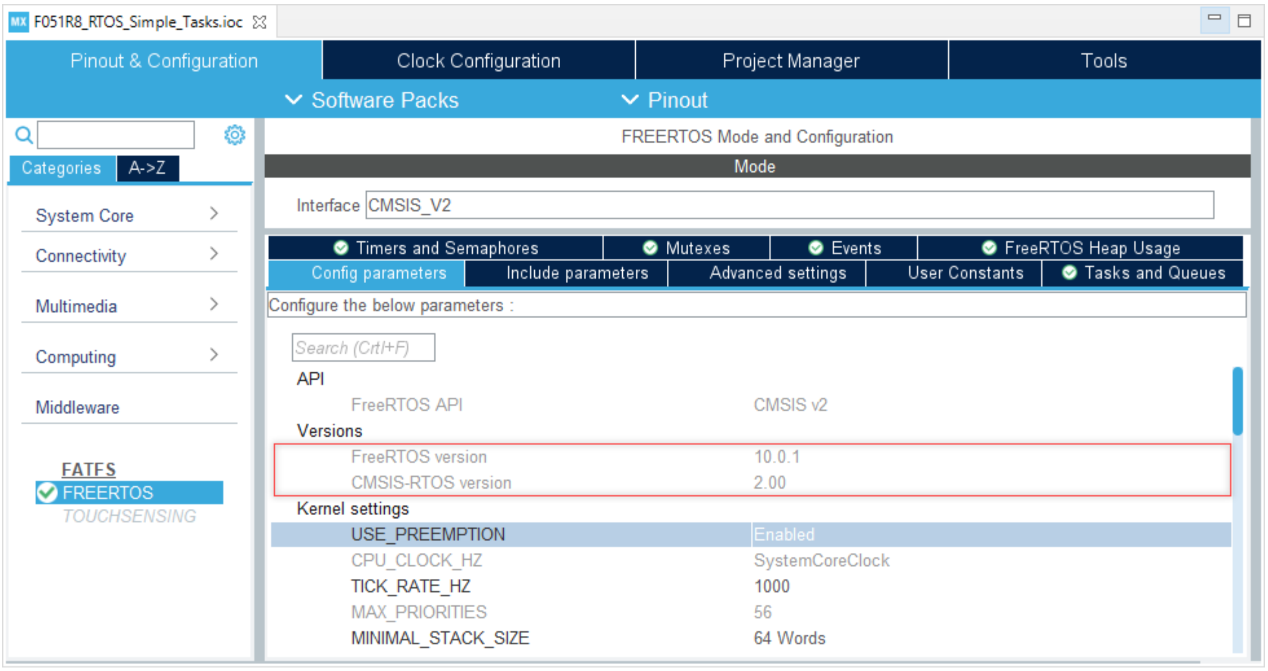
In this approach, a decision about LED states needs to be taken at an interval of the highest common factor of the delays (in the above example it is 1 second). It is cumbersome to design with this approach if the number of LEDs is large. Also, adding a newer LED (with a different blink rate) needs considerable re-work of the older code. Hence, this approach is not scalable.

This lab guides to setup RTOS with 3 simple tasks to blink LEDs and read one input button.

2.1. Enable RTOS

Under the **Pinout and Configuration** tab, select the **Middleware** section and choose **FreeRTOS**.

There are 2 version of CMSIS wrapper: Vesion 1 and Version 2. The differences are listed in ARM document site. Note to enable the option *USE_PREEMPTION*.



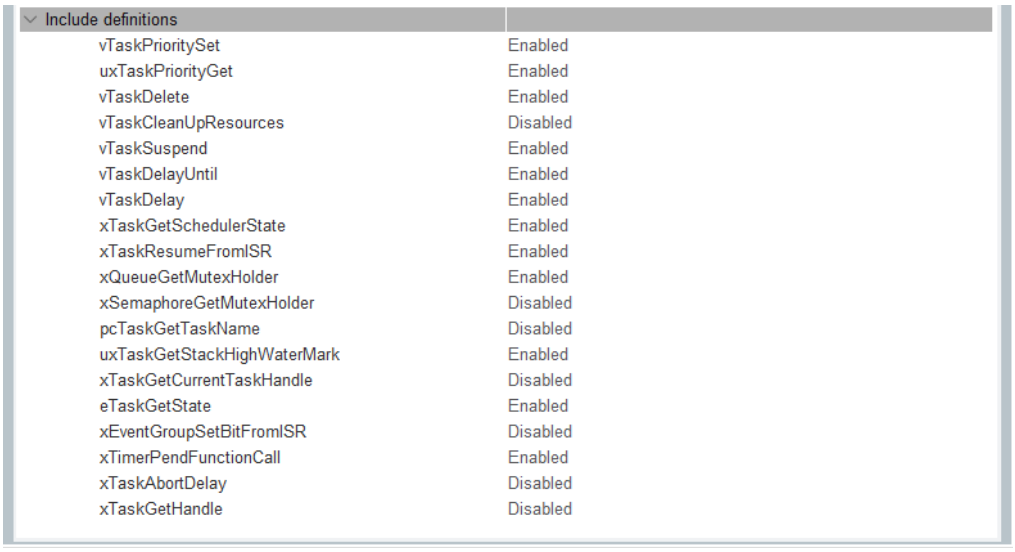
Interface CMSIS_V2

Configure the below parameters :

```
API
    FreeRTOS API              CMSIS v2
Versions
    FreeRTOS version          10.0.1
    CMSIS-RTOS version        2.00
Kernel settings
    USE_PREEMPTION            Enabled
    CPU_CLOCK_HZ              SystemCoreClock
    TICK_RATE_HZ              1000
    MAX_PRIORITIES            56
    MINIMAL_STACK_SIZE        64 Words
```

Enable RTOS version 10 with CMSIS V2

User can config some features of RTOS through a list of enabled definition.

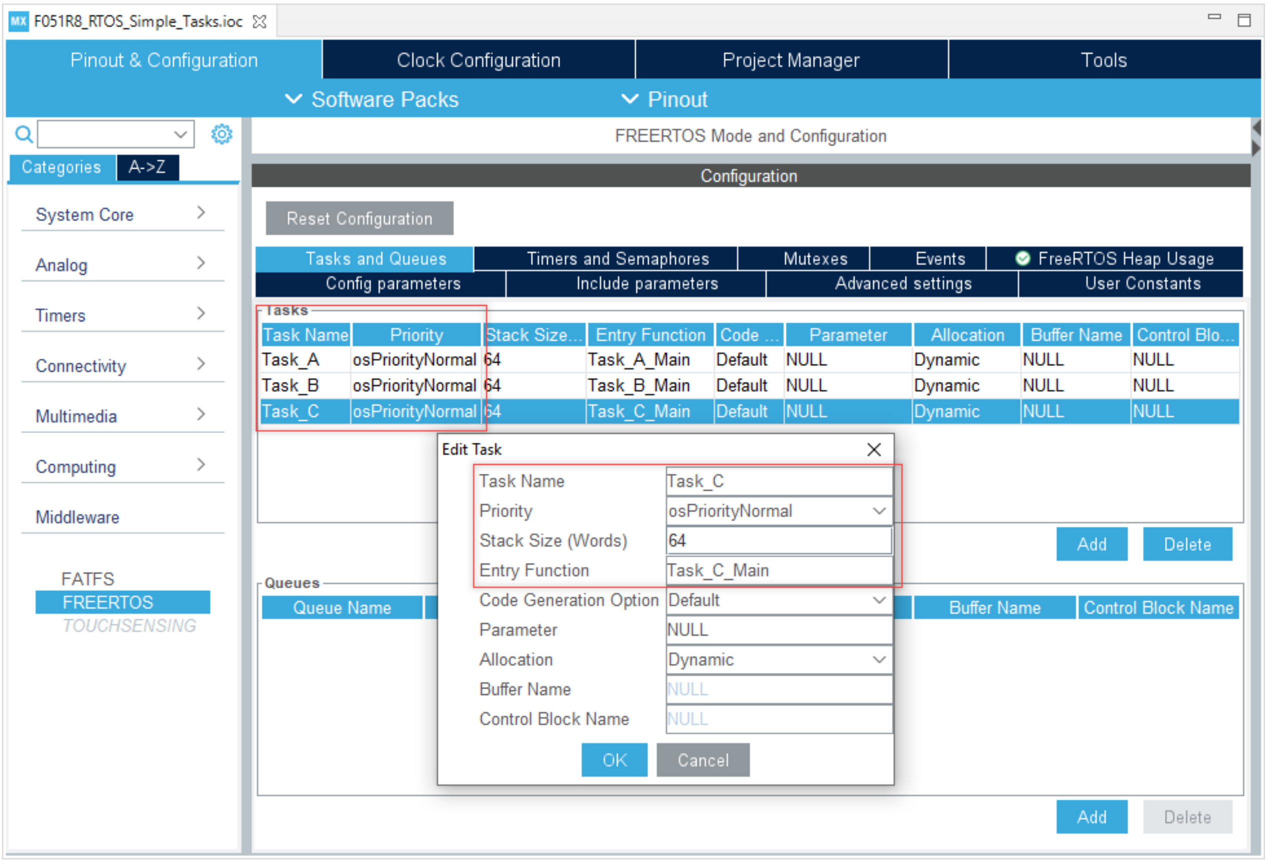


Include definitions	
vTaskPrioritySet	Enabled
uxTaskPriorityGet	Enabled
vTaskDelete	Enabled
vTaskCleanUpResources	Disabled
vTaskSuspend	Enabled
vTaskDelayUntil	Enabled
vTaskDelay	Enabled
xTaskGetSchedulerState	Enabled
xTaskResumeFromISR	Enabled
xQueueGetMutexHolder	Enabled
xSemaphoreGetMutexHolder	Disabled
pcTaskGetTaskName	Disabled
uxTaskGetStackHighWaterMark	Enabled
xTaskGetCurrentTaskHandle	Disabled
eTaskGetState	Enabled
xEventGroupSetBitFromISR	Disabled
xTimerPendFunctionCall	Enabled
xTaskAbortDelay	Disabled
xTaskGetHandle	Disabled

RTOS optional definitions

2.2. Add Tasks

Adding a task using IDE is very simple. In the tab **Tasks and Queues**, add 3 new tasks by filling the Task Name, Task Stack, and Task Function.



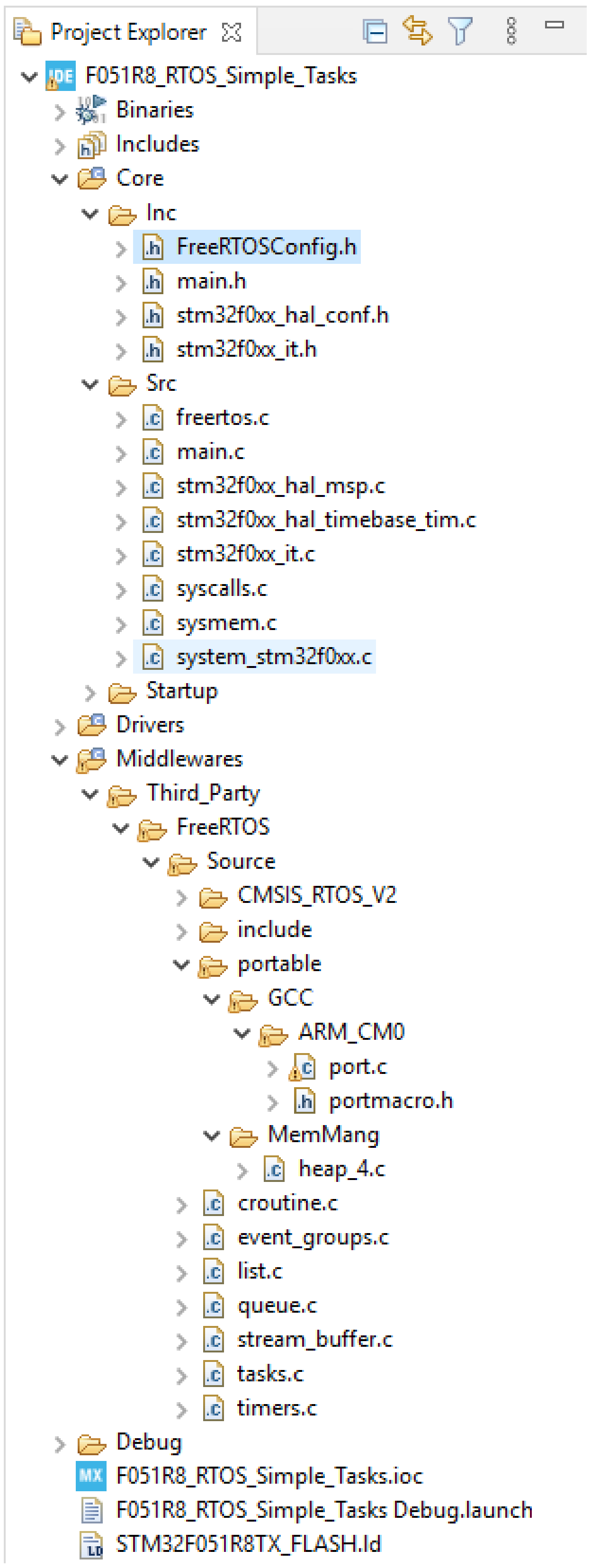
Task Name	Priority	Stack Size...	Entry Function	Code ...	Parameter	Allocation	Buffer Name	Control Blo...
Task_A	osPriorityNormal	64	Task_A_Main	Default	NULL	Dynamic	NULL	NULL
Task_B	osPriorityNormal	64	Task_B_Main	Default	NULL	Dynamic	NULL	NULL
Task_C	osPriorityNormal	64	Task_C_Main	Default	NULL	Dynamic	NULL	NULL

Edit Task

Task Name	Task_C
Priority	osPriorityNormal
Stack Size (Words)	64
Entry Function	Task_C_Main
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

Add a new Task

2.3. RTOS components



```
Project Explorer
F051R8_RTOS_Simple_Tasks
  Binaries
  Includes
  Core
    Inc
      FreeRTOSConfig.h
      main.h
      stm32f0xx_hal_conf.h
      stm32f0xx_it.h
    Src
      freertos.c
      main.c
      stm32f0xx_hal_msp.c
      stm32f0xx_hal_timebase_tim.c
      stm32f0xx_it.c
      syscalls.c
      sysmem.c
      system_stm32f0xx.c
    Startup
  Drivers
  Middlewares
    Third_Party
      FreeRTOS
        Source
          CMSIS_RTOS_V2
          include
          portable
            GCC
              ARM_CM0
                port.c
                portmacro.h
            MemMang
              heap_4.c
          croutine.c
          event_groups.c
          list.c
          queue.c
          stream_buffer.c
          tasks.c
          timers.c
  Debug
  F051R8_RTOS_Simple_Tasks.ioc
  F051R8_RTOS_Simple_Tasks Debug.launch
  STM32F051R8TX_FLASH.ld
```

RTOS components

After running code generation, there are some new folders and files added to the project. The RTOS Source code is located in the `Middlewares` folder which includes *FreeRTOS* core and `CMSIS_RTOS` wrapper.

The core files of FreeRTOS are: `task.c`, `timer.c`, `queue.c`, `list.c`, etc. Note that, on a target hardware, FreeRTOS will include some specific files for that hardware only. In the demo project which uses F051R8 MCU, FreeRTOS will include ARM_CM0 porting files.

All of the default configs for FreeRTOS are defined in the `FreeRTOS.h`. The Kernel settings in the IDE will be set in the `FreeRTOSConfig.h` file, and user can override default settings in this config file.

In the `main.c` file, there are tasks created by IDE, such as the task *Task_A* below. Note that those functions are actually CMSIS wrappers which have the `os` prefix.

```
/* Definitions for Task_A */
osThreadId_t Task_AHandle;
const osThreadAttr_t Task_A_attributes
= {
    .name = "Task_A",
    .stack_size = 64 * 4,
    .priority = (osPriority_t)
osPriorityNormal,
};

/* Definition of the Task_A_Function
*/
void Task_A_Main(void *argument) {
    for(;;) { // loop forever
        osDelay(1);
    }
}
```

Finally, in the `main()` function, FreeRTOS kernel is initialized by calling `osKernelInitialize()` and each task will be create with function `osThreadNew()` such as below call for *Task_A*:

```
Task_AHandle = osThreadNew(Task_A_Main, NULL, &Task_A_attributes);
```

To start the OS, call `osKernelStart()` and it will start a kernel loop to schedule the tasks.

Implement tasks

In this lab, there are 3 tasks:

- **Task_C** reads the button state every 100 ms

  ```
  void Task_C_Main(void *argument) {
      for(;;) {
          isButtonPressed = (HAL_GPIO_ReadPin(BUTTON_GPIO_Port,
  BUTTON_Pin)==GPIO_PIN_SET);
          osDelay(100);
      }
  }
  ```

- **Task_A** toggles the LED_A every 100 ms if button is not pressed

  ```
  void Task_C_Main(void *argument) {
      for(;;) {
          if (!isButtonPressed) HAL_GPIO_TogglePin(LED_A_GPIO_Port, LED_A_Pin);
          osDelay(100);
      }
  }
  ```

- **Task_B** toggles the LED_B every 100 ms if button is not pressed

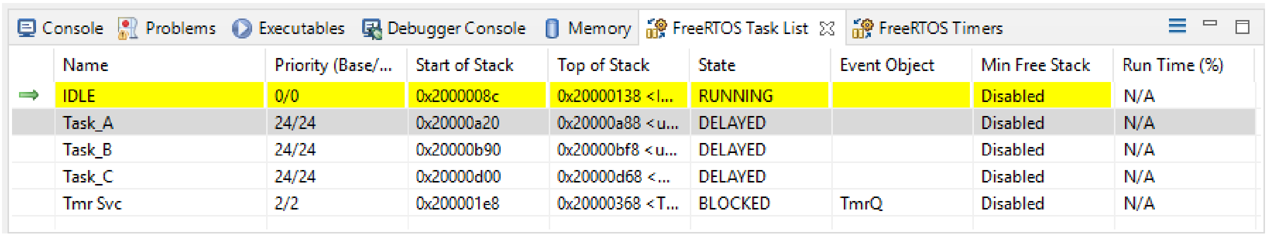
  ```
  void Task_C_Main(void *argument) {
      for(;;) {
          if (!isButtonPressed) HAL_GPIO_TogglePin(LED_B_GPIO_Port, LED_B_Pin);
          osDelay(100);
      }
  }
  ```

That is enough to create 3 concurrent tasks. Let’s run it and see how the LEDs and the button work.

2.4. The Idle Task

When running in a debug session, CubeIDE supports to see the state of all tasks under FreeRTOS environment. To open it, click on **Windows** » **Show View** » **FreeRTOS**.

There is 2 new tasks appearing in the list: *IDLE* and *TmrSrv*.



	Name	Priority (Base/...	Start of Stack	Top of Stack	State	Event Object	Min Free Stack	Run Time (%)
⇒	IDLE	0/0	0x2000008c	0x20000138 <I...	RUNNING		Disabled	N/A
	Task_A	24/24	0x20000a20	0x20000a88 <u...	DELAYED		Disabled	N/A
	Task_B	24/24	0x20000b90	0x20000bf8 <u...	DELAYED		Disabled	N/A
	Task_C	24/24	0x20000d00	0x20000d68 <...	DELAYED		Disabled	N/A
	Tmr Svc	2/2	0x200001e8	0x20000368 <T...	BLOCKED	TmrQ	Disabled	N/A

Task List

The idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run. It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.

The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the `vTaskDelete()` function to ensure the idle task is not starved of processing time. The idle task has no other active functions so can legitimately be starved of microcontroller time under all other conditions.

The Idle Task Hook: An idle task hook is a function that is called during each cycle of the idle task. It is common to use the idle hook function to place the microcontroller CPU into a power saving mode.

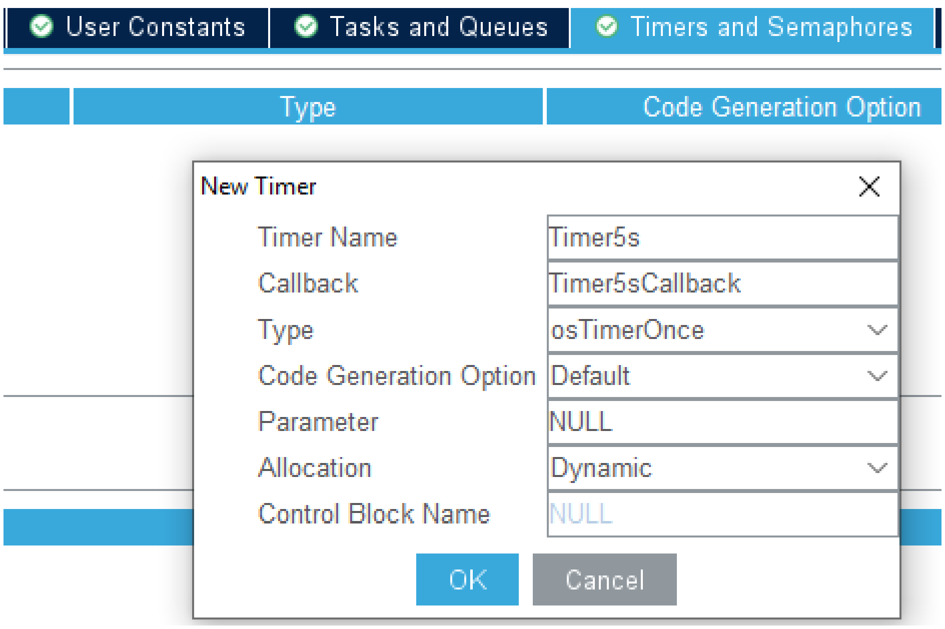
2.5. The Timer Service

There is a dedicated `Tmr Svc` (Timer Service or Daemon) task that maintains an ordered list of *Software Timers*, with the timer to expire next in front of the list). The Timer Service task is not continuously running: from the Timer List, the task knows the time when it has to wake up each time a timer in the timer list has expired. When a timer has expired, the Timer Service task calls its callback (the Timer callback).

A Software Timer

Let’s modified the Lab 0 a bit:

- **Task_A** and **Task_B** toggle their LEDs be default
- If user presses on the button, **Task_C** will block LED toggling
- After 5 seconds, system will unlock LED toggling
- During 5 seconds, if user presses on the button again, the 5 second period is restarted



Timer Name	Timer5s
Callback	Timer5sCallback
Type	osTimerOnce
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Control Block Name	NULL

Create a Software Timer

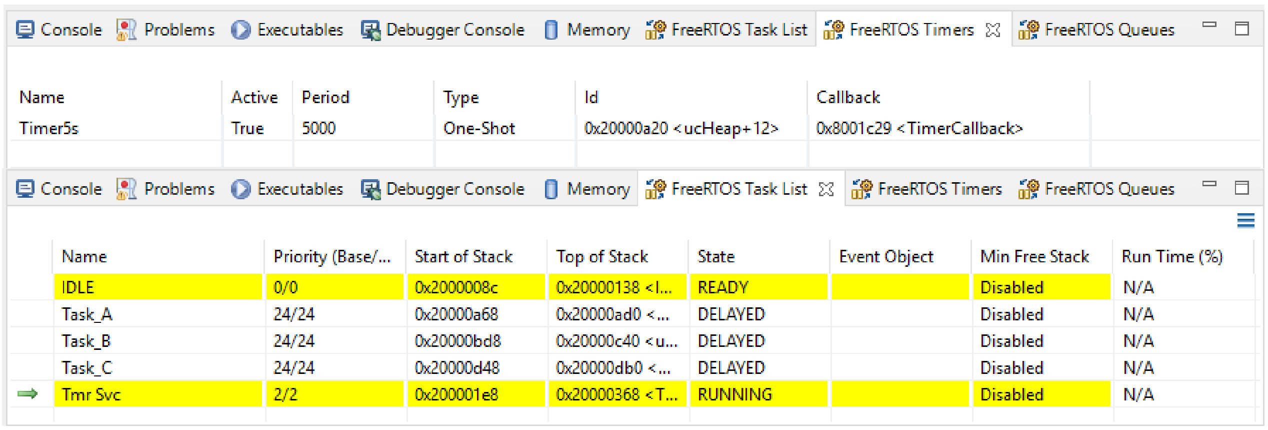
Here are generated code for this Soft Timer:

```
osTimerId_t Timer5sHandle;
const osTimerAttr_t Timer5s_attributes = {
  .name = "Timer5s"
};

void Timer5sCallback(void *argument) {
    /* add code here */
}

int main() {
    Timer5sHandle = osTimerNew(Timer5sCallback, osTimerOnce, NULL,
&Timer5s_attributes);
}
```

And here is the modified work of the **Task_C**:

```
void Task_C_Main(void *argument){
  for(;;) {
    if(HAL_GPIO_ReadPin(BUTTON_GPIO_Port, BUTTON_Pin) == GPIO_PIN_SET) {
      isButtonPressed = 1;
      osTimerStart(Timer5sHandle, 5000);
    }
    osDelay(100);
  }
}

void Timer5sCallback(void *argument) {
  isButtonPressed = 0;
}
```

When debugging, Soft timers are listed in the FreeRTOS Timers list, and `Tmr Srv` will be executed when one of soft timers reaches to its configured period counter.



Name	Active	Period	Type	Id	Callback
Timer5s	True	5000	One-Shot	0x20000a20 <ucHeap+12>	0x8001c29 <TimerCallback>

	Name	Priority (Base/...	Start of Stack	Top of Stack	State	Event Object	Min Free Stack	Run Time (%)
	IDLE	0/0	0x2000008c	0x20000138 <I...	READY		Disabled	N/A
	Task_A	24/24	0x20000a68	0x20000ad0 <...	DELAYED		Disabled	N/A
	Task_B	24/24	0x20000bd8	0x20000c40 <u...	DELAYED		Disabled	N/A
	Task_C	24/24	0x20000d48	0x20000db0 <...	DELAYED		Disabled	N/A
⇨	Tmr Svc	2/2	0x200001e8	0x20000368 <T...	RUNNING		Disabled	N/A

Software Timer and the Timer Service status