FreeRTOS - 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.

#arm #stm32 #rtos

Last update: 2021-08-04 17:31:07

Table of Content

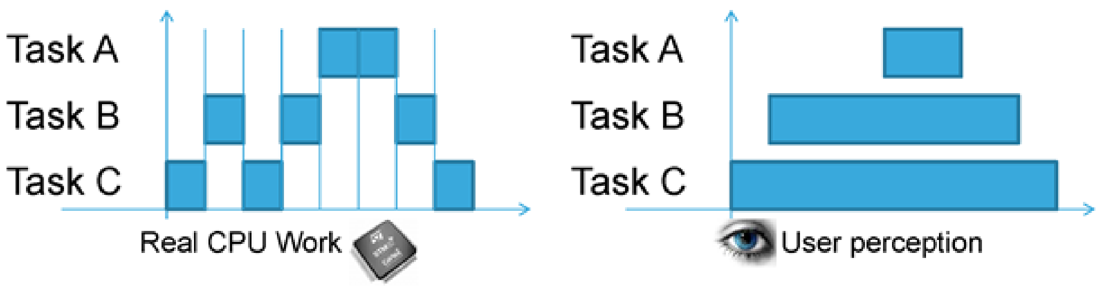
1. RTOS

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.

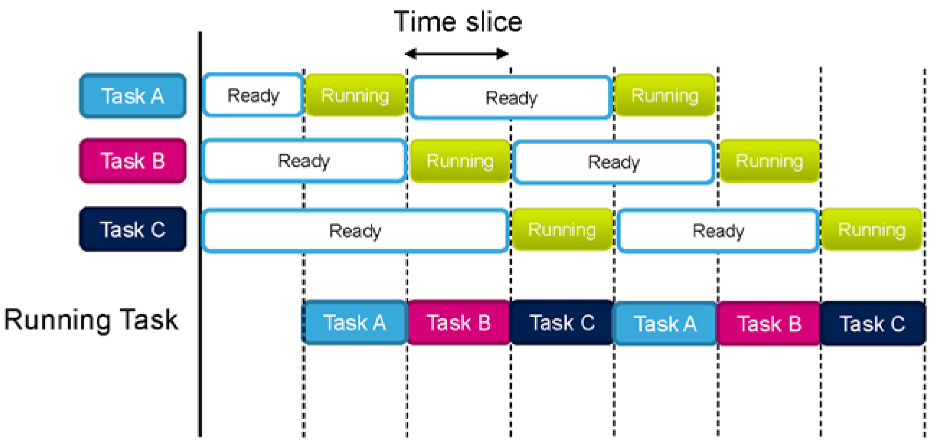
A Task has 4 states:

- **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

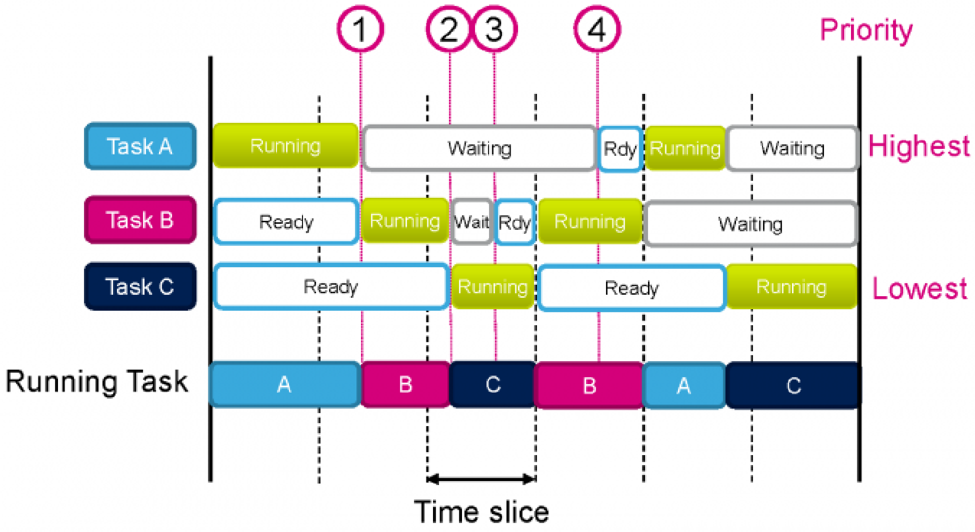
1.2. The Scheduler

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. Memory Allocation

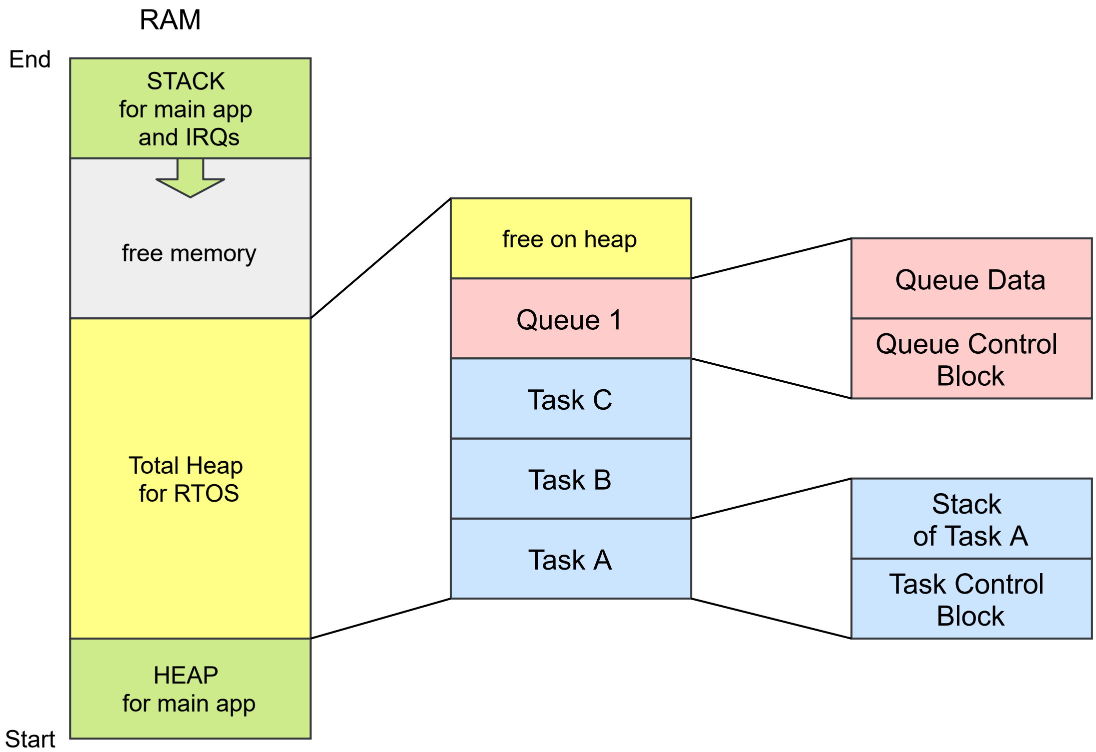
Real time operating system supports **static** and **dynamic** memory allocation, with different strategies and algorithm.

Creating RTOS objects dynamically has the benefit of greater simplicity, and the potential to minimize the application’s maximum RAM usage:

- The memory allocation occurs automatically.
- The RAM used by an RTOS object can be re-used if the object is deleted.
- The memory allocation scheme used can be chosen to best suite the application.

Creating RTOS objects using statically allocated RAM has the benefit of providing the application more control:

- RTOS objects can be placed at specific memory locations.
- It allows the RTOS to be used in applications that simply don't allow any dynamic memory allocation.
- Avoid memory-related issues such as leak memory, dangling pointer, and undefined objects.



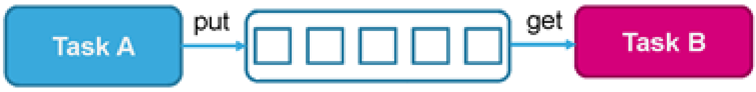
Memory layout in FreeRTOS

1.5. Shared Memory

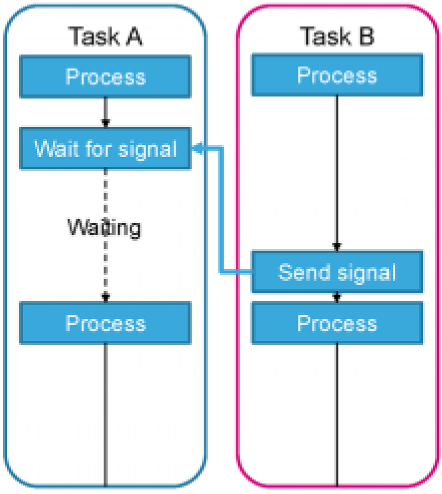
Tasks are usually a work to do in a loop and it thinks it can control all of resource. In a system, there are many tasks run together, and in many cases, they works with condition from others.

Inter-task communication is defined as some type:

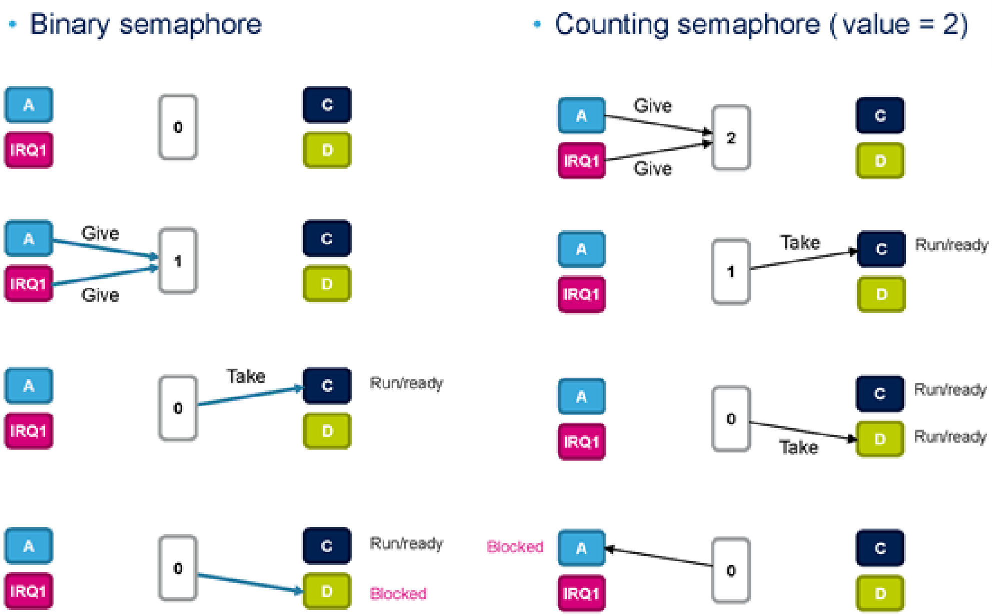
- Signal: tell other task to start doing somethinng, to synchronize tasks
- Message Queue/Mailbox: send data between tasks
- Mutex/Semaphore: synchronize acess to a shared resource, lock resource which is in-use



Queue between tasks



Signal between tasks



Shared resource between tasks

2. FreeRTOS for STM32

In the STM32Cube firmware solution, FreeRTOS is used as a real time operating system through the generic CMSIS-OS wrapping layer provided by ARM. Examples and applications using the FreeRTOS can be directly ported on any other RTOS without modifying the high level APIs, only the CMSIS-OS wrapper has to be changed in this case.

2.1. Main features

- Preemptive or cooperative real-time kernel
- Tiny memory footprint (less than 10kB ROM) and easy scalable
- Includes a tickless mode for low power applications
- Synchronization and inter-task communication using
    - message queues
    - binary and counting semaphores
    - mutexes
    - group events (flags)
    - stream buffer
- Software timers for tasks scheduling
- Execution trace functionality
- CMSIS-RTOS API port

2.2. Used resources

Core resources:

- System timer (`SysTick`) – generate system time (time slice)
- Two stack pointers: MSP, PSP

Interrupt vectors:

- `SVC` – system service call (like SWI in ARM7)
- `PendSV` – pended system call (switching context)
- `SysTick` – System Timer

Memory:

- Flash: 6-10 KB Flash +
- RAM memory: 0.5 KB + task stacks

2.3. File structure

File	Description
`task.c`	ask functions and utilities definition
`list.c`	List implementation used by the scheduler
`queue.c`	Queue implementation used by tasks
`timers.c`	Software timers functions definition
`port.c`	Low level functions supporting SysTick timer, context switch, interrupt management on low hw level – strongly depends on the platform (core and sw toolset). Mostly written in assembly
`FreeRTOS.h`	Configuration file which collect whole FreeRTOS sources
`FreeRTOSConfig.h`	Configuration of FreeRTOS system, system clock and irq parameters configuration
`heap_x.c`	Different implementation of dynamic memory management
`croutine.c`	Co-routines functions definitions. Efficient in 8 and 16bit architecture. In 32bit architecture usage of tasks is suggested
`event_groups.c`	Flags to notify tasks about am event

2.4. Memory Management

FreeRTOS uses a region of memory called Heap (into the RAM) to allocate memory for tasks, queues, timers , semaphores, mutexes and when dynamically creating variables. FreeRTOS heap is different than the system heap defined at the compiler level.

When FreeRTOS requires RAM, instead of calling the standard `malloc()` , it calls `PvPortMalloc()` . When it needs to free memory it calls `PvPortFree()` instead of the standard `free()` .

FreeRTOS offers several heap management schemes that range in complexity and features. The FreeRTOS download includes five sample memory allocation implementations, each of which are described in the following subsections. The subsections also include information on when each of the provided implementations might be the most appropriate to select.

Heap management schemes:

- heap_1 - the very simplest, does not permit memory to be freed.
- heap_2 - permits memory to be freed, but does not coalescence adjacent free blocks.
- heap_3 - simply wraps the standard `malloc()` and free() for thread safety.

- heap_4 - coalescence adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
- heap_5 - as per **heap_4**, with the ability to span the heap across multiple non-adjacent memory areas.

Notes:

- **heap_1** is less useful since FreeRTOS added support for static allocation.
- **heap_2** is now considered legacy as the newer **heap_4** implementation is preferred.

For more detail, refer to RTOS Memory Management.

2.5. Interrupts

PendSV interrupt

- Used for task switching before tick rate
- Lowest NVIC interrupt priority
- Not triggered by any peripheral

SVC interrupt

- Interrupt risen by SVC instruction
- SVC 0 call used only once, to start the scheduler (within `vPortStartFirstTask()` which is used to start the kernel)

SysTick timer

- Lowest NVIC interrupt priority
- Used for task switching on `configTICK_RATE_HZ` regular time base
- Set PendSV if context switch is necessary

2.6. API conventions

1. Prefixes at variable names:
   - `c` – char / `s` – short / `l` – long / `u` – unsigned
   - `x` – portBASE_TYPE defined in `portmacro.h` for each platform (in STM32 it is long)
   - `p` - pointer
2. Functions name structure: `prefix` + `file name` + `function name`.
   For example: `vTaskPrioritySet()`.
3. Prefixes at macros defines their definition location and names.
   For example: `portMAX_DELAY`

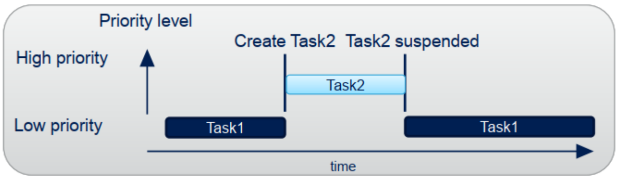
2.7. General Configs

Configuration options are declared in file `FreeRTOSConfig.h`.

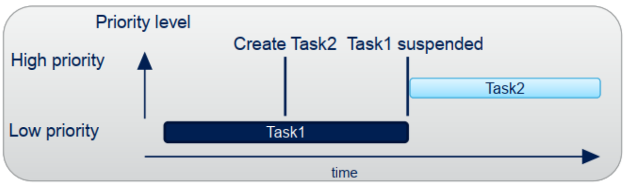
Important configuration options are:

Config option	Description
`configUSE_PREEMPTION`	Enables Preemption
`configCPU_CLOCK_HZ`	CPU clock frequency in Hz
`configTICK_RATE_HZ`	Tick rate in Hz
`configMAX_PRIORITIES`	Maximum task priority
`configTOTAL_HEAP_SIZE`	Total heap size for dynamic allocation
`configLIBRARY_LOWEST_INTERRUPT_PRIORITY`	Lowest interrupt priority (0xF when using 4 cortex preemption bits)
`configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`	Highest thread safe interrupt priority (higher priorities are lower numeric value)

If **preemption is enabled**, RTOS will uses **pre-emptive** scheduling, otherwise, RTOS will uses **co-operative** scheduling:



Pre-emptive scheduling



Co-operative scheduling

The `xPortGetFreeHeapSize()` API function returns the total amount of heap space that remains unallocated (allowing the `configTOTAL_HEAP_SIZE` setting to be optimized). The total amount of heap space that remains unallocated is also available with `xFreeBytesRemaining` variable for heap management schemes 2 to 5.

Each created task (including the idle task) requires a Task Control Block (TCB) and a stack that are allocated in the heap. The TCB size in bytes depends of the options enabled in the `FreeRTOSConfig.h`:

- With minimum configuration the TCB size is 24 words i.e 96 bytes.
- if `configUSE_TASK_NOTIFICATIONS` enabled add 8 bytes (2 words)
- if `configUSE_TRACE_FACILITY` enabled add 8 bytes (2 words)
- if `configUSE_MUTEXES` enabled add 8 bytes (2 words).

The task stack size is passed as argument when creating at task. The task stack size is defined in words of 32 bits not in bytes. `Task Memory = TCB size + (4 x Task Stack size)`.

The `configMINIMAL_STACK_SIZE` defines the minimum stack size that can be used in words. the idle task stack size takes automatically this value.

When Soft Timers are enabled (`configUSE_TIMERS` enabled) , the scheduler creates automatically the timers service task (daemon) when started. The timers service task is used to control and monitor (internally) all timers that the user will create. The scheduler also creates automatically a message queue used to send commands to the timers task (timer start, timer stop, etc.).

The number of elements of a queue (number of messages that can be hold) are configurable through the define `configTIMER_QUEUE_LENGTH`.

2.8. CMSIS_OS API

- CMSIS-OS API is a generic RTOS interface for Cortex-M processor based devices. Implementation in file `cmsis-os.c` in `\Middlewares\Third_Party\FreeRTOS\Source\CMSIS_RTOS`.
- Middleware components using the CMSIS-OS API are RTOS independent, this allows an easy linking to any third-party RTOS.
- The CMSIS-OS API defines a minimum feature set including
  - Thread Management
  - Kernel control
  - Semaphore management
  - Message queue and mail queue
  - Memory management

For detailed documents, refer to CMSIS-RTOS.

2.8.1. CMSOS_RTOS Wrapper

API category	CMSIS_RTOS API	FreeRTOS API
Kernel control	`osKernelStart`	`vTaskStartScheduler`
Thread management	`osThreadCreate`	`xTaskCreate`
Semaphore	`osSemaphoreCreate`	`vSemaphoreCreateBinary`, `xSemaphoreCreateCounting`
Mutex	`osMutexWait`	`xSemaphoreTake`
Message queue	`osMessagePut`	`xQueueSend`, `xQueueSendFromISR`
Timer	`osTimerCreate`	`xTimerCreate`

Most of the functions returns `osStatus` value, which allows to check whether the function is completed or there was some issue (defined in the `cmsis_os.h` file).

Each OS component has its own ID:

- Tasks: `osThreadId` (mapped to `TaskHandle_t` within FreeRTOS API)
- Queues: `osMessageQId` (mapped to `QueueHandle_t` within FreeRTOS API)
- Semaphores: `osSemaphoreId` (mapped to `SemaphoreHandle_t` within FreeRTOS API)
- Mutexes: `osMutexId` (mapped to `SemaphoreHandle_t` within FreeRTOS API)
- SW timers: `osTimerId` (mapped to `TimerHandle_t` within FreeRTOS API)

Delays and timeouts are given in ms:

- `0` – no delay
- `>0` – delay in ms
- `0xFFFFFFFF` – wait forever (defined in osWaitForever within cmsis_os.h file)

3. 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);
```

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

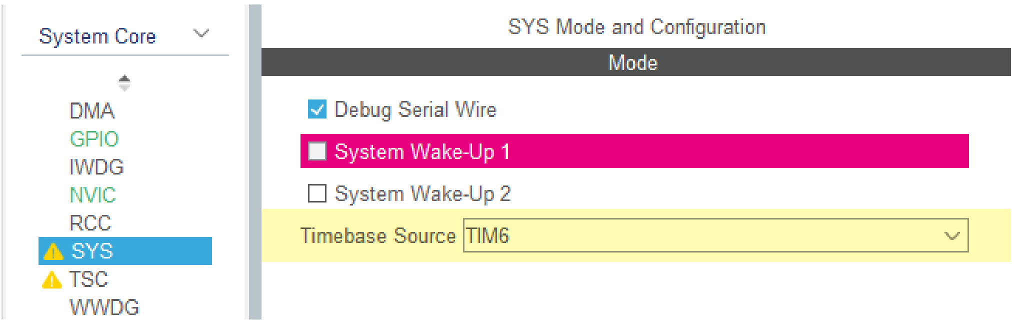
```
        LED1_TURN_OFF();
        delay_seconds(1);

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

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

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.

3.1. Create a new project

Start a new project and select a target MCU. After setting up the clock and basic pinouts, it is the time to select a timer for HAL timebase.



Select timebase source for HAL functions

3.2. Enable RTOS

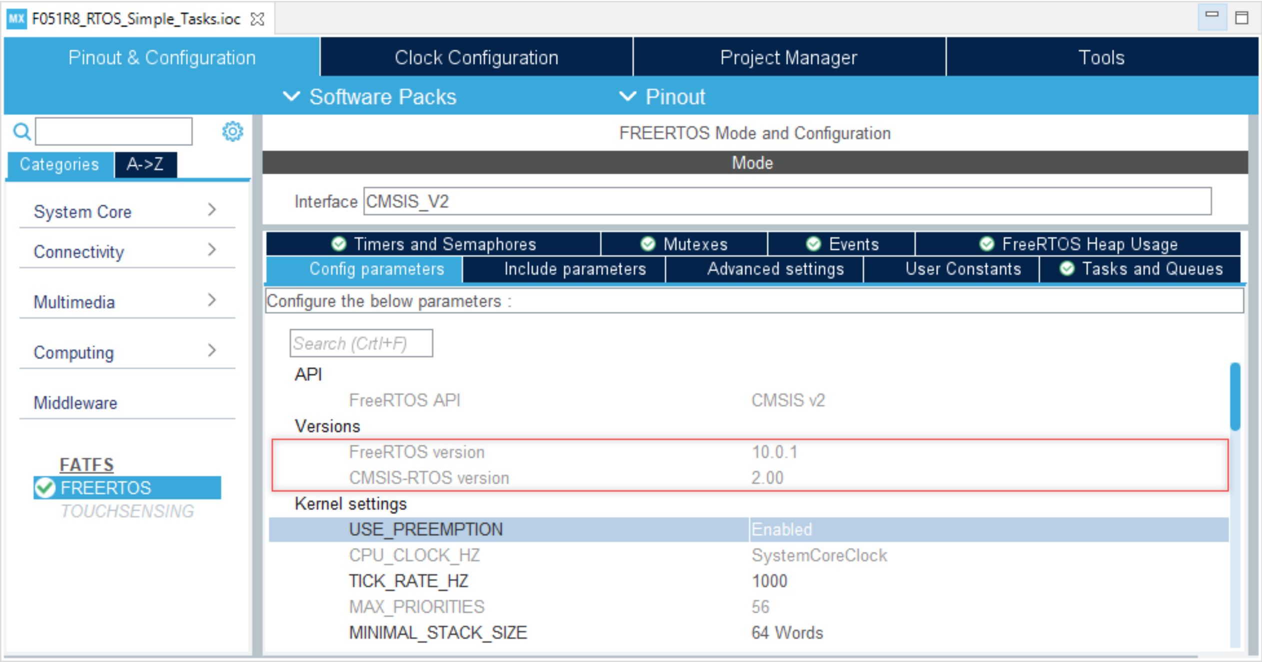
Under the **Pinout and Configuration** tab, select the **Middleware** section and choose **FreeRTOS**. There are 2 version of CMSIS wrapper: Vesion 1 and Version 2. The differences are listed in ARM document site.

Note to enable the option *USE_PREEMPTION*. User can config some features of RTOS through a list of enabled definition.

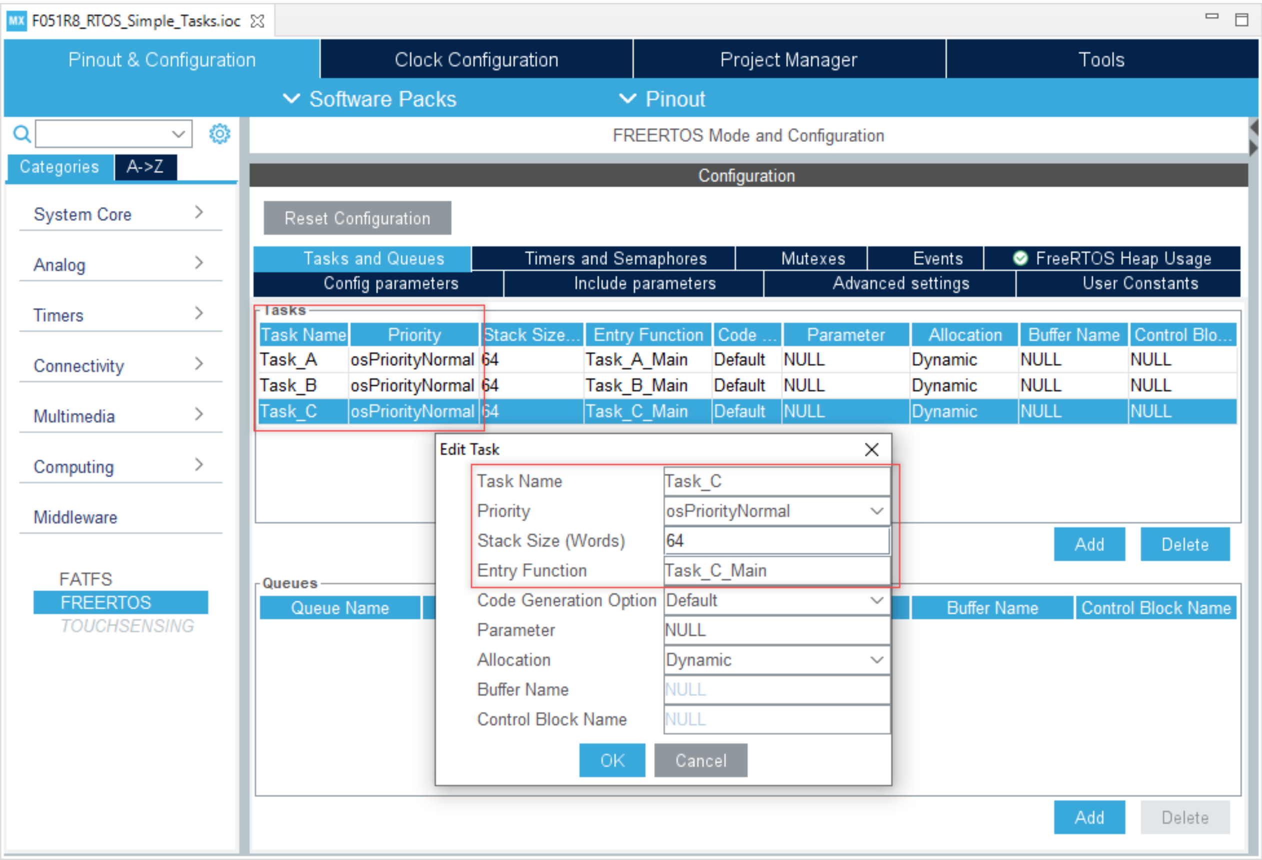
3.3. Add Tasks

Adding a task using IDE is very simple. In the tab **Tasks and Queues**, add 3 new tasks by filling some importance settings for a task: Task Name, Task Priority, Task Stack size, and Task Function.

It is optional to set the Task Allocation mode, which is set to Dymamic as default.

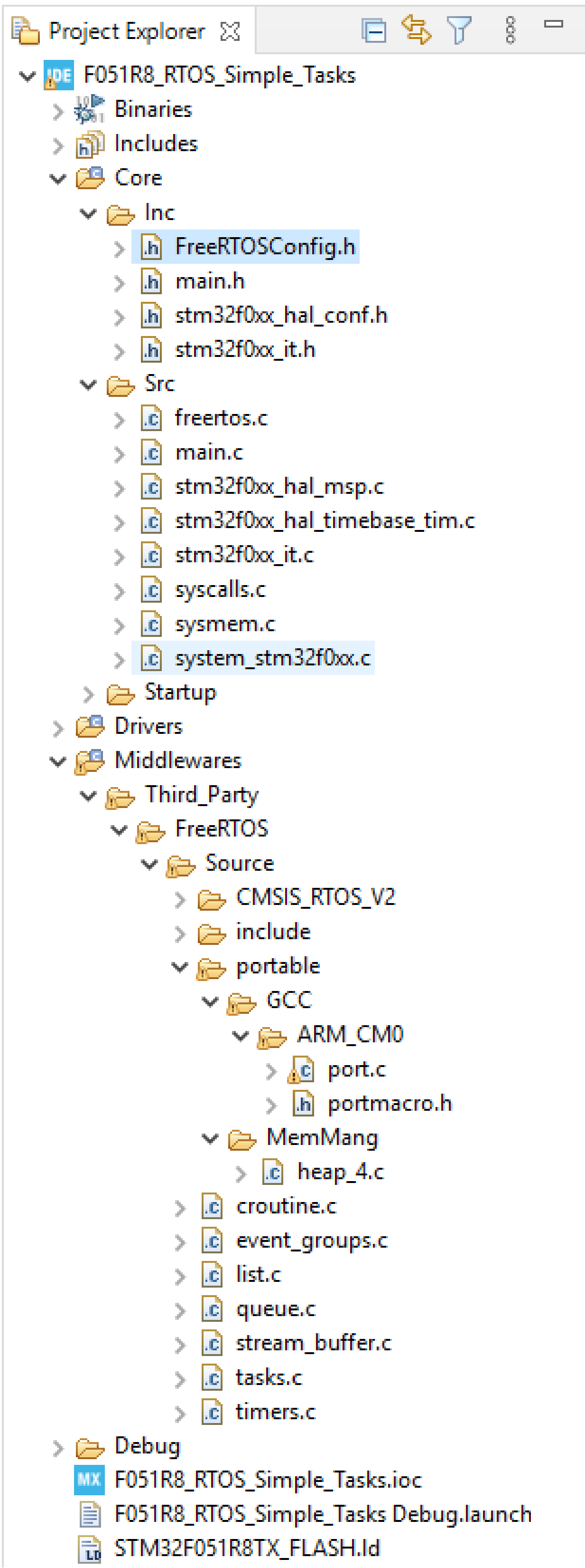


Enable RTOS version 10 with CMSIS V2



Add a new Task

3.4. RTOS components



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 pressed

  ```
  void Task_C_Main(void *argument) {
    for(;;) {
      if (isButtonPressed) {
          HAL_GPIO_TogglePin(LED_A_GPIO_PortLED_A_Pin);
      }
      osDelay(100);
    }
  }
  ```

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

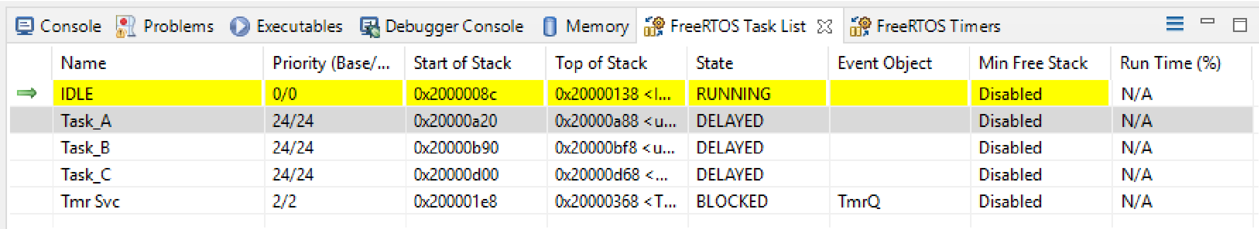
  ```
  void Task_C_Main(void *argument) {
    for(;;) {
      if (!isButtonPressed) {
          HAL_GPIO_TogglePin(LED_B_GPIO_Port, LED_B_Pin);
      }
      osDelay(100);
    }
  }
  ```

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

3.5. 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.

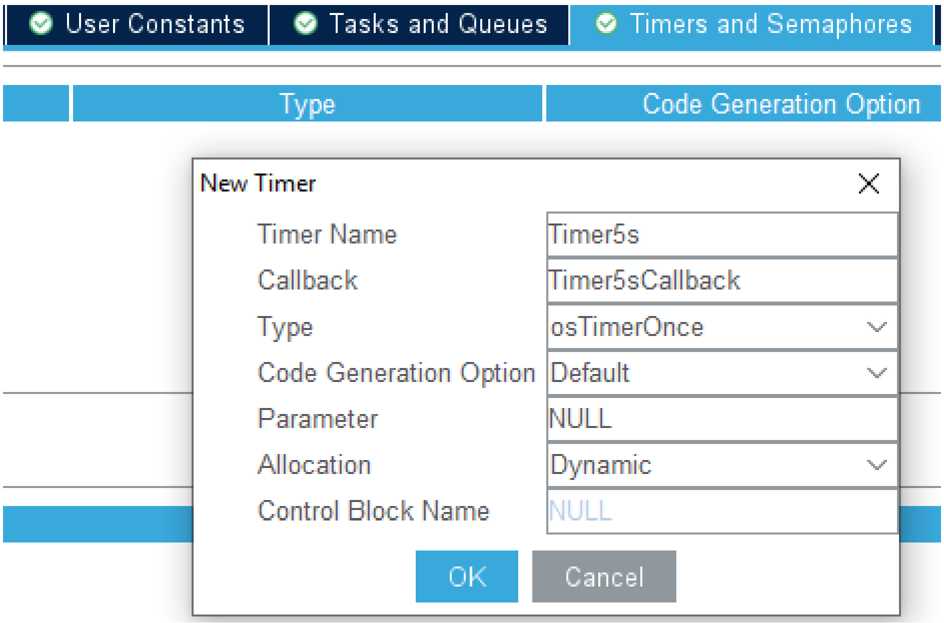
3.6. The Timer Service

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



Create a Software Timer

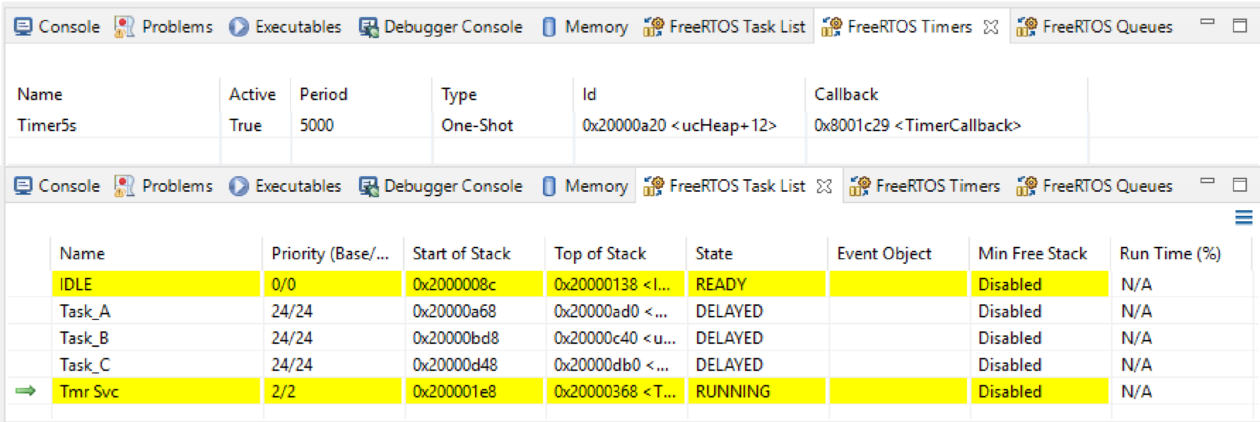
Here are generated code for this Soft Timer:

```
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