Lab 6 - Real-time Operating System & DMA

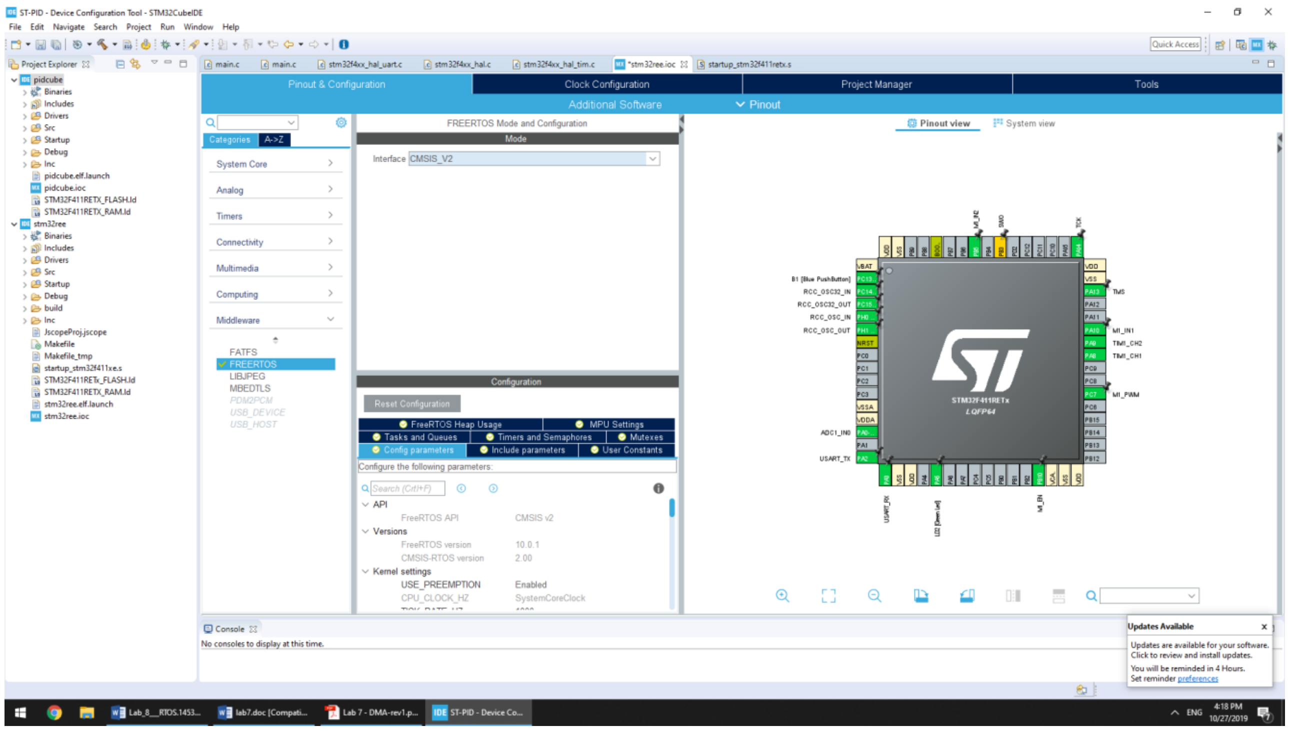
Typical microcontroller for an appliance nowaday may require to operate many functionality at the same time. For example, a miicrocontroller for an air conditioning would need to control the fan speed, the compressor operations, the IR interface to receive remote control, and the actual temperature control. While this is applicable using a single thread program, it is cumbersome and would require a careful planning so that one operation does not completely block another.

A better approach to handle multiple tasks on a microcontroller is to use real-time operating system. Real-time operating system is similar to a normal operating system in that it is implementing multi-task management and scheduling. In some case, it may also implement device drivers, file system and access control. What differentiate real-time operating system to a normal operating system the scheduling of RTOS is real-time, i.e. it has some guarantee deadline.

There are many RTOSs: mBed, Nuttx, FreeRTOS, ChibiOS, VxWorks, etc. Wikipedia curratted the list of RTOSs
https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems .

The most well-known in industrial one is VxWorks, which powered many things ranging from industrial equipment, satellite or even Mars's landover. VxWorks is orbitally expensive (each development seat costs $10,000 yearly, not including other licensing), so instead we will be one of the most widely used free RTOS, which is FreeRTOS.

Setting up a RTOS to use with microcontroller manually can be challenging for beginner. To help with that problem, CubeMX can be used to setup the project.



Interface CMSIS_V2

Configure the following 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
```

During code generation, it may complaint about SysTick. This is because FreeRTOS use SysTick for their scheduling, and if a user has their own code in SysTick, it may not behave the same way. You do not need to fix this, but if you want to, you can go to Pinout->SYS->Timebase Source and change it to any other timer.

After you generate the code with FreeRTOS enable, you will find that there is a new piece of code

```
/* Create the thread(s) */
 /* definition and creation of defaultTask */
 osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
 defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
```

The `osThreadDef` macro is used to create a thread definition, which has the following parameter

```
#define osThreadDef(name, thread, priority, instances, stacksz)
```

```
/// Create a Thread Definition with function, priority, and stack requirements.
/// \param        name        name of the thread function.
/// \param        thread      call function
/// \param        priority    initial priority of the thread function.
/// \param        instances   number of possible thread instances.
/// \param        stacksz     stack size (in bytes) requirements for the thread function.
```

`The thread is then created using osThreadCreate,` Note that this is not a FreeRTOS thread creation API, but rather CMSIS-RTOS, which is a standard RTOS for Arm Cortex not specific to FreeRTOS. Unfortunately, ST implementation of CMSIS-RTOS using FreeRTOS is slightly different from CMSIS-RTOS specification.

Suppose that you want to create a new thread, you could do so by the following code:

```
/* Add this code in   StartDefaultTask  */
 osThreadDef(led, led_thread, osPriorityNormal, 0, 128);
 osThreadId ledTaskHandle = osThreadCreate(osThread(led), NULL);
```

This will create a thread calling function ***led_thread*** of the following type.

```
void led_thread(void const *args)
```

In this case, we assume that this thread stack size is 128. You may also pass a parameter when you create a thread using osThreadCreate to the function. Read through https://os.mbed.com/handbook/CMSIS-RTOS for additional information. You may also find a specific implementation of CMSIS-RTOS interface of STM32 through this document https://www.st.com/resource/en/user_manual/dm00105262.pdf

If you prefer not to use CMSIS-RTOS API, you can call function FreeRTOS API directly.

Your tasks:

1. ***Simple Multitask***

In this task, you will create 4 threads, as follow:

- The first thread blinks red led every 50ms period
- The second thread blinks red led every 18ms period
- The third thread blinks red led every 128ms period
- The forth thread blinks red led every 64ms period

You can use ***osDelay*** to delay with a given specific time in number of millisecond. Note that this may not be possible your tickrate is less than 1000. You can set tickrate In Configuration->FreeRTOS->Config Parameter->TICK_RATE_HZ. Make sure that it is greater than 1000.

2. ***Simple Multitask using Timer***

In this task, you will be doing exactly the same as the first one except that you will be using timer for your implementation.

3. ***Mutex/Semaphore***

Starting from either the first or the second task, i.e. your leds are still blinking. Create two additional threads: each thread use UART to transmit as follow:

```
int threadID = 0; // threadID is 0 for one thread and 1 for another
int idx = 0;
char buffer[32];
while(1) {
  sprintf(buffer, "TID: %d %d\r\n", threadID, idx);
  idx ++;
  HAL_UART_Transmit(&huart2, buffer, strlen(buffer), 1000);
}
```

While this will send out messages through UART, it may skip the number because the UART is busy. Add either mutex or semaphore to fix this problem, so that the number running from each thread has continuous index.

** If you are adventurous, you may try to use `HAL_UART_Transmit_IT or HAL_UART_Transmit_DMA.`

4. ***Message Queue/Mailbox***

Starting from task 3, you will create one more thread that handle all the UART transmission. Then you will modify the threads you create in the previous task to submit message through mailbox or messagequeue.