Microprocessor Systems Lab 6c

Checkoff and Grade Sheet

Partner 1 Name:________________________________

Partner 2 Name:________________________________

Grade Component	Max.	Points Awarded		TA Init.s	Date
		Partner 1	Partner 2		
Performance Verification: Task 1	10 %				
Task 2	20 %				
Task 3	20 %				
Documentation and Appearance	50 %				
Total:					

→ Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. Create multiple FreeRTOS CMSIS_v2 threads and run them concurrently
2. Send data between threads using Queues
3. Notify events with event flags
4. Setup software timers and notify events to a single thread with thread flags
5. Use a mutex to control access to a hardware resource

→ Reading and References

R1. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapters 6 (ADC), 15 (DAC), 26 (GPIO), 66 (UART)

R2. RM0410-stm32f7_Reference_Manual.pdf: Chapters 6 (GPIO), 15 (ADC), 16 (DAC), 34 (USART)

R3. CMSIS_v2 API Overview

R4. Wikipedia RTOS Overview

R5. Lab06_RTOS_Template.zip: Project Template for Lab 6 RTOS

RTOS Overview

You likely have heard of an Operating System, which is a program that serves to provide application with basic functions, such as accessing memory, interacting with peripherals, and interacting with other applications. They allow applications to run on a variety of hardware by abstracting implementation details behind drivers, and also manage running multiple applications at once.

Operating systems use a scheduler to run multiple applications at once, by switching out the applications between a single (or multiple) processor core(s). By switching often enough, it appears that the applications are running concurrently.

Real time operating systems (RTOS) use schedulers designed in a way to result in deterministic performance. Therefore, an RTOS implementation is relevant to embedded systems as they often have real time requirements, such as interfacing with motors or sensors. It is important for the behaviour of the system to be consistent and repeatable.

FreeRTOS + CMSIS_v2

FreeRTOS is an open source RTOS implementation that can run on microcontrollers. As opposed to the commonly used general purpose operating systems like Windows, MacOS, or Linux[1], it does not provide any interfaces for interacting with a console, networks, or storage. It only includes functionality for real

[1]In refering to Linux as a "general purpose operating system", we are considering the typical laptop/desktop distributions. Embedded Linux does exist and is quite common

time scheduling of threads (called "tasks" in FreeRTOS), inter-task communication, and primitives for synchronization and mutual exclusion.

CMSIS is a generic RTOS interface for Arm Cortex devices. Using this interface allows programmers to write programs form Arm devies and switch out the RTOS used, as long as it supports CMSIS. Most of STM's examples make use of the CMSIS interface, and as a result this lab uses the CMSIS_v2 interface. The FreeRTOS API will not be accessed directly, but understanding the constructs defined in CMSIS_v2 will be applicable for any RTOS.

Notes about the FreeRTOS/CMSIS_v2 Implementation

Typically, the STM32 HAL makes use of the `SysTick_Handler` callback to increment the HAL internal ticks. This is necessary for the HAL to work properly, and is used since the SysTick has a very high priority. However, FreeRTOS makes use of this handler. As a result, Timer 7 has been repurposed for the HAL ticks. This means you cannot make use of TIM7 with this template for FreeRTOS/CMSIS_v2.

Very Important!!

While CMSIS_v2 functions can be used in interrupts and IRQs, it is very important that proper interrupt priority is observed. Any ISRs that make use of CMSIS_v2 functions cannot have a priority higher than 5, so any interrupts you add should have a priority of at least 6 (lower number is higher priority). This prevents the user implemented interrupts from blocking the RTOS functionality and possibly breaking the "real time" operation of the system.

CMSIS_v2 Program Structure

In typical embedded programs, you put some initialization code at the beginning of `main`, then have an infinite loop and/or interrupts to handle state. A CMSIS program instead begins with `osKernelInitialize()`, followed by setting up any RTOS objects you will use, followed by `osKernelStart()`. The processor should never return from this function. Instead, any threads you set up will be started, and each of those will have their own infinite loop that they will run in. Specific RTOS provided functions can be called within threads that will yields control to other threads.

In general non-RTOS blocking functions should not be used in an RTOS. This is because blocking functions prevent the RTOS from changing which thread is currently running and therefore concurrency is lost. Use Interrupt or DMA mode when using peripherals.

All of the functions available in RTOS CMSIS_v2 are available at `https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS.html`. It is also recommneded to read the C function API overview at `https://www.keil.com/pack/doc/CMSIS/RTOS2/html/rtos_api2.html`

Threads

→ Threads in CMSIS_v2 RTOS

Each thread can be in one of 4 states: READY, BLOCKED, RUNNING, INACTIVE. Only one thread is in the running state at any time. Figure 1 shows the different states and how threads can move between states. A running thread can be moved to blocked, ready, or terminated depending on what happens. If it needs to wait for an event or time, it will move to the blocked state. It can also be terminated, or terminate itself. Finally, it can be pre-empted by a higher priority thread. A blocked thread that is waiting for some event c an move to ready once the event occurs, or directly to running if nothing else is running.

Using these different states, multiple tasks can run concurrently, with higher priority tasks getting chances to preempt and run over other tasks.

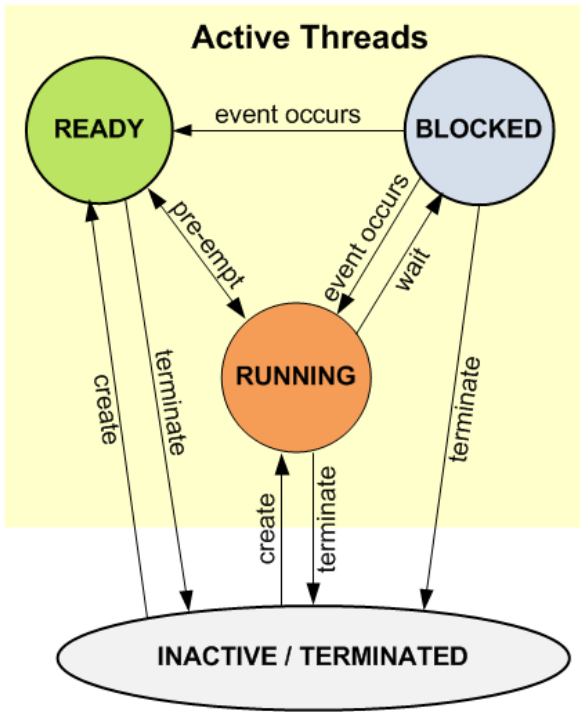

Figure 1: Thread States [keil.com]

A sample in Figure 2 shows an example of how two tasks may be run concurrently. At first, Task 1 is running. However, it has to wait for some external event, so yields to the RTOS. The RTOS then gives control to task 2. However, when the external event occurs, the RTOS hands execution back over to Task 1. This assumes that Task 1 has a higher priority than Task 2. Otherwise, the RTOS would handle the external event then give control back to Task 2.

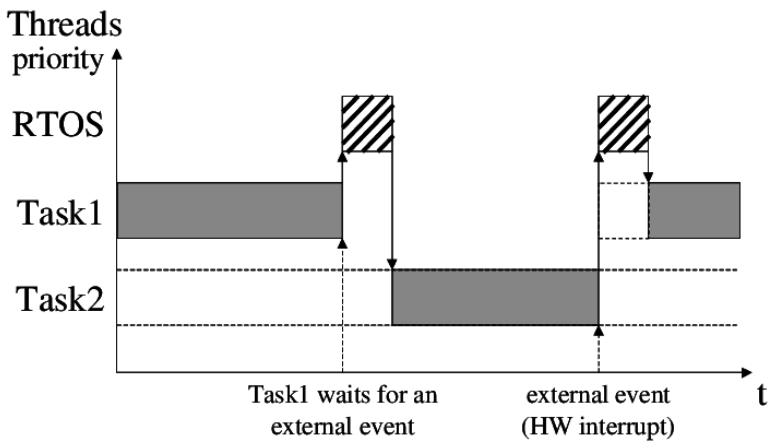
Threads priority
RTOS
Task1
Task2
Task1 waits for an external event
external event (HW interrupt)
t

Figure 2: Sample Execution [keil.com]

→ Creating Threads in CMSIS_v2

Threads are referred to by their thread handle, of type `osThreadId_t`. Each thread also has attributes, defined by a struct `osThreadAttr_t`. Below shows a sample declaration for the thread handle and definition of the attributes struct. We also define a function that this thread will run.

```
osThreadId_t myThreadHandle;
const osThreadAttr_t myThread_attributes = {
        .name = "myThread",
        .priority = (osPriority_t) osPriorityNormal,
        .stack_size = 128 * 4
}
void StartMyThread(void *argument);
```

The `name` field of the attributes struct is used for debugging. The `priority` field is of type `osPriority_t` and defines the priority of the thread. See the CMSIS documentation for a full list of priorities. `stack_size` defines how many bytes are allocated to the stack for the thread. If the thread will make many function calls and/or have many local variables, the stack size may need to be increased.

In the main function, after `osKernelInitialize()`, but before `osKernelStart()` the threads should be created. Create the threads with `osThreadId_t osThreadNew(osThreadFunc_t func, void * argument, const osThreadAttr_t * attr)`. This returns the thread handle object for this function, and needs to be passed a callback function, argument for the callback function, and attributes.

As example:

```
int main()
{
...
    osKernelInitialize();
...
    myThreadHandle = osThreadNew(StartMyThread, NULL, &myThread\_attributes);
...
    osKernelStart();
...
}

// This is the thread function
void StartMyThread(void *argument)
{
    while(1)
    {
        osDelay(1) // Delay for 1 tick
    }
    // Should not return
}
```

Use the above process to create multiple threads. In order to switch between threads, we make use of RTOS provided functions. The simplest is `osDelay(uint32_t ticks)`, which delays for the passed number of ticks. (Here 1 tick is 1ms). This will yield to another thread while delaying.

◇ Task 1: Multiple LEDs Blinking With Threads

Create two threads to blink the two User LEDS on the board. The Red LED should blink at 2Hz, and the Green LED should blink at 1Hz. The GPIO for each LED should be controlled by a seperate thread.

Queues

Queues can be used to transmit data between threads. While queues are generally a first in-first out (FIFO) data structure, CMSIS_v2 queues have priorities on the message, meaning that a higher priority message can be received before an earlier, lower priority message.

Queues are created similarly to threads, with a handle and attributes.

```
osMessageQueueId_t myQueueHandle;
const osMessageQueueAttr_t myQueue_attributes = {
    .name = "myQueue"
};
```

Create the queue with `osMessageQueueId_t osMessageQueueNew(uint32_t msg_count, uint32_t msg_size` `const osMessageQueueAttr_t *attr)`. `msg_count` is the number of elements in the queue, `msg_size` is the maximum size of a single element in by te, and `attr` is the queue attributes.

For example:

```
myQueueHandle = osMessageQueueNew(32, sizeof(uint8_t), &myQueue_attributes);
```

Finally data can be added to the queue, and pulled from the queue, using `osMessageQueuePut` and `osMessageQueueGet`. Both of these functions take the queue handle, and a time out, as arguments. If the timeout is `osWaitForever` then the thread will block until there is space in the queue/until there is something in the queue. While the thread is block other threads will be allowed to run. `osMessagePut` also takes arguments for the message data and message priority.

See all the available queue functions at https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__Message.html

◇ Task 2: UART Loopback with Queue

The USB_UART has already been set up in interrupt mode. Add this functionality to your code from task 1. Create a queue, and add received characters to the queue in the interrupt. Create a new thread that reads from this queue and prints the characters back to the console.

Note: while functions such as `osMessageQueuePut` can be used inside an IRQ, there are certain limitations on what parameters can be passed. View the documentation to see what parameters are acceptable inside an IRQ.

Event Flags

While queues can be used to send data between threads, event flags are used for signaling. Events are a set of 32 bits that can be used to signal. For example, the singal event generator thread in Figure 3 sends

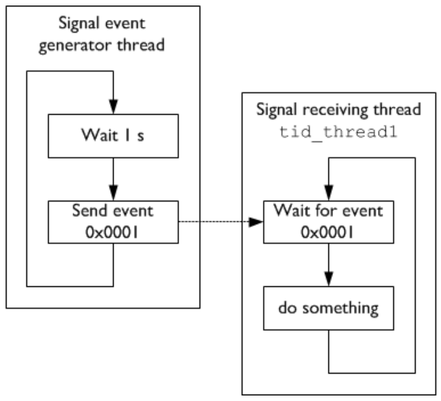

Figure 3: Event Example

0x1, or bit 1. The signal receiving thread waits for this event, then performs an action. If instead flag 0x2 was sent, bit 2 would be sent. Multiple flags can be sent, for example sending 0x3 would set flags 1 and 2.

Create an event with a handle (`osEventFlagsId_t`) and attributes(`osEventFlagsAttr_t`), just like with threads and queues, using `osEventFlagsId_t osEventFlagsNew(const osEventFlagsAttr_t *attr)`.

Set one or multiple flags with `osEventFlagsSet (osEventFlagsId_t ef_id, uint32_t flags)`. Flags can be checked and cleared with `osEventFlagsGet` and `osEventFlagsClear`. Alternatively, a thread can wait on a flag (and yield to other threads while waiting) using `osEventFlagsWait`.

◇ Task 3: ADC DAC Forwarding with Event Flags

Add another thread to your previous tasks. Setup the ADC and DAC in interrupt mode so that whenever the ADC reading is made, send an event from the ADC interrupt to the thread. This thread should read the ADC value, send it to the DAC, and start another ADC reading. Compare this method with event signaling to directly sending the value from the ADC to the DAC in the interrupt. How does the performance compare?

Thread Flags

Thread flags are very similar to event flags, except they can only be sent to one thread. The main advantage of using thread flags over event flags is that no new objects need to be allocated. Just like event flags, thread flags can be set with `osThreadFlagsSet`, and retrieved with `osThreadFlagsGet` and `osThreadFlagsWait`. Instead of passing a thread flag handle, the thread handle is used. Only the thread receiving the flag can get, wait on, or clear the flag.

RTOS Timers

Most RTOS' provide software timers that can be used for periodic tasks, or providing one shot timer delays. Figure 4 shows a sample of how a periodic timer may be used. Every time the timer expires, the callback function is called. It can also be manually restarted or stopped.

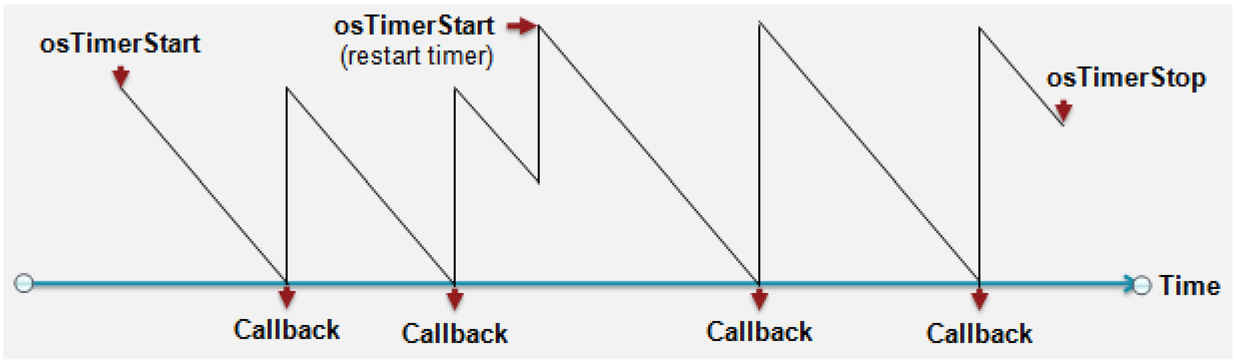

Figure 4: Timer Sample

A timer needs a handle(`osTimerId_t`) and attributes object(`osTimerAttr_t`). A callback function is also needed, which has the same function signature as a thread function (returns `void`, takes a single `void *` as argument). The timer can then be created with `osTimerId_t osTimerNew (osTimerFunc_t func, osTimerType_t type, void *argument, const osTimerAttr_t *attr)`. The `osTimerType` can be either `osTimerOnce` for a oneshot timer, or `osTimerPeriodic` for a periodic timer.

Once the timer has been created, it can be started with `osTimerStart (osTimerId_t timer_id, uint32_t ticks)`. `ticks` is the period of the function in ticks, which here are set to 1ms. Make sure that you only call `osTimerStart` inside a thread.

◇ Task 4: Timer Print

Create a new thread to print a message to the console every 2 second. Add this thread to the tasks above.

Mutexes

Mutexes, short for mutual exclusion, are used to guard access to a shared resource. They work under a very simple principle. A thread can only access a resource if it currently holds the mutex for that resource, shown in Figure 5. This could be a storage device, GPIO pin, a DMA stream, some hardware acceleration peripheral, or any other resource.

When a mutex is held by a thread, and a thread wants to access it, it will enter the blocked state. The second thread will unblock once the mutex is freed.

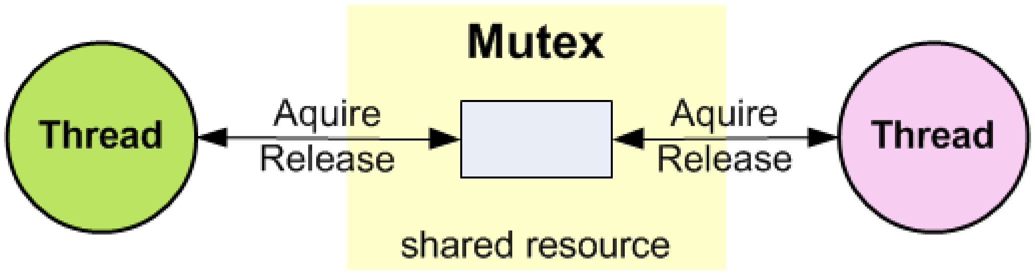

Figure 5: Mutex Scenario

Mutexes can cause unexpected behaviour with respect to priority. Assume there are three threads T1, T2, and T3 with priorities Low, Normal, and High. Let T1 access a mutex and perform some operation. It then gets pre-empted by T3 to do a task. However, T3 also needs that mutex, so blocks while it waits for T1. T1 will resume until it gets preempted by T2. Then T2 will have priority over T1, therefore blocking T1, and by extension T3. As a result, a Normal priority thread is blocking a High priority thread.

This can be rectified with priority inheritance. When a higher priority thread is blocked by a mutex on a lower priority thread, the lower priority thread will have it's priority upgraded to be equal to the priority as the higher priority thread. In the above case, T1 will be upgraded to high priority, and run instead of T2, eventually unblock T3 when it releases the mutex.

Mutexes are created like all other CMSIS_v2 RTOS objects, by calling `osMutexNew` and providing an `osMutexAttr_t` which will return a `osMutexId_t`. The `osMutexAttr_t` struct has a field `attr_bits` which can set several options for the mutex. `osMutexRecursive` means a therad can consume a mutex multiple times without locking itself. `osMutexPriorityInherit` implements priority inheritance as described above. `osMutexRobust` means that the mutex is released when the owning thread is terminated.

Once the handle is created, mutexes can be acquired and released within threads using `osMutexAcquire` and `osMutexRelease`. The mutex can only be released from the thread that acquires it.

◇ Task 5: [Depth] Error Status LED with Mutexes

Create a mutex that controls access to the Red LED. Create another thread that will rapidly blink the Red LED. When the character 'e' is entered onto the serial console, have the new thread rapidly blink the led to indicate an "error". When the 'n' character is entered, have the led go back to being controlled by the thread in Task 1. Make use of this mutex to ensure that only one thread is controlling this hardware resource (LED) at a time.

You will likely need to use thread and/or event signaling between your UART thread and your new LED Error thread, as well as add the acquisiton and release of the mutex to your thread from Task 1.

Semaphore Overview

Semaphores are another construct used to control accesses to shared resources, but are used differently than mutexes. They are used in two main ways, as a non-binary mutex (called a multiplex) and to control production/consumption of some resource. In addition, semaphores can be released by threads other than the one that acquired them. This makes them useful for passing data/resources around bewteen threads.

Figure 6 shows the multiplex use case. The semaphore limits the number of threads that can access some shared resource. In this case there are 6 available instances of the shared resource, so once all 6 have been acquired, the next thread to request one must wait. An example resource could be DMA resources, where larger chips have multiple DMAs that can all perform similar roles, and a thread can make use of any of them.

Another use case is a producer/consumer design pattern. In this design pattern, a producer thread is generating data in some manner and putting into memory buffers. A consumer is taking these buffers and doing some work with the data. When the buffer is used, the used buffer can be returned to the free available buffers. In this case, two semaphores are used, where the first counts down the available empty buffers and the second counts up the used buffers. This results in each thread blocking and passing control to other threads when there are no empty/filled buffers for them to use.

See https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__SemaphoreMgmt.html#details for examples and documentation.

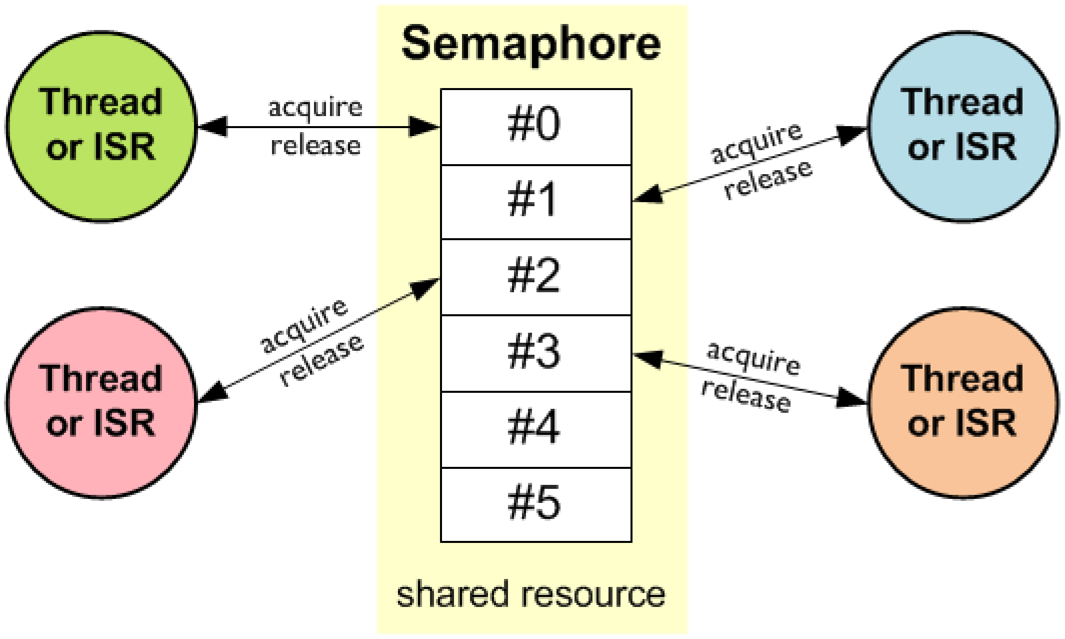

Figure 6: Semaphore Use Case [keil.com]