Experiment #4

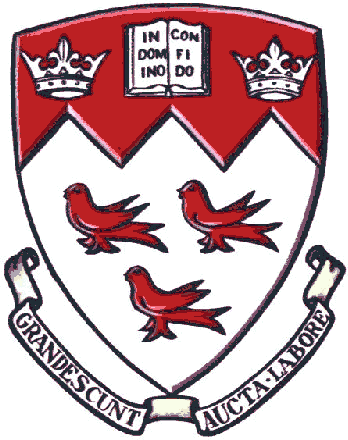
Multithreaded, interrupt-driven sensor reading and peripheral control

Maxim Goukhshtein ID: 260429739

Olivier Laforest ID: 260469066

Department of Electrical and Computer Engineering

McGill University, Montreal



March 23rd, 2015

Group 3

ECSE 426 Microprocessor Systems

# Abstract

The goal of the experiment presented in this report was to design and implement a multithreaded system using the real time operating system (RTOS) CMSIS-RTOS, capable of sensing the STM32F407 Discovery board's processor temperature and pitch angle and provide a visual display of those readings to the user. The report will show how a 4x4 alphanumeric keypad was used to provide the user with the means to select the desired mode of operation and how a 4 digits 7-segments display in combination with the on-board LEDs was used to provide visual feedback to the user. To that end, the work of experiments 2 and 3 were slightly modified and successfully combined using threads in an RTOS environment on the discovery board. This report will also show how the thread implementation was done so as to allow for the concurrent multithreaded computation to operate safely.

# Problem Statement

The goal of this experiment is to design and implement a system which can concurrently sense both the current processor's temperature and either the pitch or roll angle of the STM32F407 Discovery board. To that end, the on chip temperature sensor of the discovery board's processor is to be used in order to measure the processor's temperature and the off chip tri-axial MEMS Accelerometer sensor LIS3DSH is to be used to measure the gravitational acceleration in order to compute the board's pitch or roll angle. The system needs to provide the user with a means to input the desired mode of operation through the use of a 4x4 alphanumeric keypad. The system should also provide visual feedback using the 4 digits 7-segments display, as well as the on-board LEDs of the discovery board. There are two possible modes of operation in which the system may find itself in, temperature mode or accelerometer mode. In temperature mode, the system needs to display the real time current temperature of the board's processor in degrees Celsius on the 7-segments display. If the temperature exceeds an overheating threshold, the 7-segments display should flash on and off repeatedly as long as the processor's temperature remains above the threshold to denote danger level. In accelerometer mode, the system needs to display the real time current pitch or roll angle of the board on the 7-segment display and will allow the user to select one of the four user LEDS on the board and adjust the LED brightness using pulse width modulation (PWM) according to the board's angle. The LED should be completely off when the board's angle is and gradually get brighter as the board's angle gets wider all the way up to where it should be the brightest. The alphanumeric keypad's keys 1 to 4 should allow the user to select which one of the four user LEDs is to be lit at the current time. The user also needs to be able to switch from temperature mode to accelerometer mode and the other way around by pressing preselected key(s) on the keypad. Note that while in accelerometer mode, if the board's processor temperature exceeds the overheating threshold, the 7-segments display should also be flashing on and off to denote danger level. To allow for concurrent measurement of the processor's temperature and the board's tilt angle, the system should be a multithreaded system that uses CMSIS-RTOS. The work from experiments 2 and 3 should be modified and incorporated into threads in order to achieve the desired system and therefore, the requirements regarding sampling rates of the sensors, calibration and filtering are the same as those found in those respective experiments.

# Theory and Hypothesis

This experiment combines the work done in labs 2 and 3 into a multithreaded sensing system which concurrently performs measurements of the discovery board's temperature and tilt angle, displays the appropriate information through a four digits 7-segments display and on board LEDs, and records user inputs through a 4x4 alphanumeric keypad. The theory for the operation of the temperature sensor, ADC conversion and operation, MEMS sensor usage and calibration, Kalman filters, etc, has been discussed in the theory sections of experiments 1 to 3 lab reports and will therefore not be covered here.

## Real-Time Operating Systems (RTOS)

Real-time operating systems (RTOS) make embedded programming similar to desktop programming and they usually include a large amount of pre-made services such as process management, file management, memory management, date and time, user management, networking, etc [1]. They allow real-time operation which means that they process information coming from sensors connected to it in real-time, or with minimum delay. RTOS can be separated into two categories, soft and hard time RTOS. In soft time RTOS, the operating system generally meets a deadline and in hard time RTOS, the system deterministically meets the deadline [2]. RTOS allow multi-tasking to take place through the use of threads. They provide tools to synchronize those threads and utilities to share resources among them. The CMSIS-RTOS is an API which provides a generic interface all ARM Cortex-M processor devices [3]. The CMSIS-RTOS API interfaces with an existing real-time Kernel and provides several attributes and functionalities which are listed below [3].

### Threads

Threads are different processes or tasks which run concurrently within a system/application and may have interdependencies with each other. Systems which runs on single core processor may only run one thread at a time. Such systems need a scheduler, which is a mechanism to decide which thread should currently be running. Each thread possess its own stack and state buffer and each is assigned a priority to allow proper scheduling [2]. Each thread may be in one of the following four states:

* Running: Only one thread at a given time can be in that state on single core systems. The thread in the running state is the task which is currently executing.
* Ready: Threads which are ready to be executed are in this state. Once the thread which is in the running state terminates, the thread which is ready and has the highest priority will switch to the running state and start execution.
* Waiting: A thread in the waiting state is a thread which is waiting for an event to occur (e.g. signal flag to be set, semaphore or mutex to be released, message or mail to be received, etc.)
* Inactive: Threads which have not been created or terminated are inactive and therefore do not use any system resources [4].

The following figure shows the state transition diagram for threads in CMSIS RTOS.

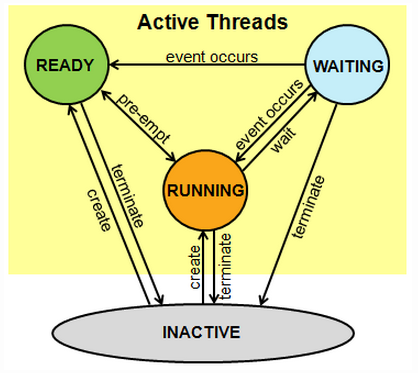


Figure 1: State transition diagram for threads in CMSIS-RTOS

### Scheduler

The CMSIS-RTOS (RTX version) uses a pre-emptive scheduler which schedules which thread should be running when an event happens. The different possible events can be a resource release, the tick of a timer, arrival of a message or mail, status change of a signal flag, etc [2] [3]. The thread with the highest priority is the one which will be in the running state [2]. If two or more threads have the same priority, the scheduler will have each of them run alternatively following a round-robin scheme. The equal priority tasks will alternate at tick events which are generated by the SysTick which is consequently no longer available to the programmer. The frequency of the tick event can be configured. A low frequency leads to less frequent change in the threads which is currently running and therefore reduces the overhead computational cost of context switching. A high frequency allows each equal priority threads to run more frequently and therefore lead to a more even distribution of computational resources [2]. Since the thread priority dictates which thread is going to be running and when it is going to be running, the assignment of priorities by the designer is very important to ensure the proper operation of the system.

### Communication between threads

Communication between threads can be done via signals, message queues and/or mail queues.

* Signals: Signal functions allow a thread to communicate with other threads by setting, clearing or waiting for signal flags. Signal flags are assigned to each threads [4].
* Message queues: Message queue functions allow to define a memory pool associated with a message queue, send, receive and/or wait for messages. Integers or pointer value may be sent through as messages to a thread or an interrupt service routine [4].
* Mail queues: Mail queue are very similar to message queues except that they are used to send a block of memory instead of an integer or a pointer value [4].

### Synchronization

To allow synchronization between the executions of threads sharing a resource, CMSIS-RTOS provides mutex and semaphore management functions.

**Mutex**

A mutex ensures that a resources which is used by more than one thread can only be accessed by one of those threads at a time. When a thread acquires the resource, this resource becomes unavailable to other threads until it is released by the thread which is currently using it. The other threads needing the shared resource need to wait in the mean time, which is why threads requiring a shared resource should minimize their critical section (i.e. a section of code in which a shared resource is being used) as much as possible in order to avoid starving other threads [4]. The following figure illustrates the interaction between the threads and the mutex.

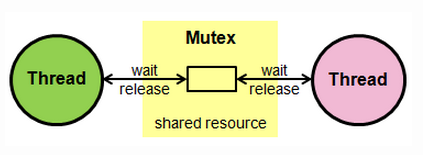


Figure 2: Interaction between threads and a mutex [4]

**Semaphore**

A semaphore works in a very similar way to a mutex, except that they protect the access to shared resources which have several identical instances of the resource (e.g. ports). Therefore, a semaphore has a fixed number associated to it which corresponds to the number of instances of the resource available. When a thread gets access to one instance of the resource, it decreases the semaphore. When a thread is done using the shared resource, it releases it and increases the semaphore. When the semaphore reaches 0, no more instances of the resource are available and subsequent threads requiring the shared resource need to wait for one of the instances to be released [4]. The following figure illustrates the interaction between several threads and a semaphore.

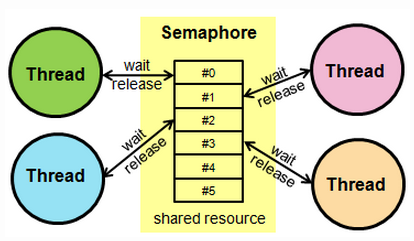


Figure 3: Interaction between several threads and a semaphore [4]

### Timer

The CMSIS-RTOS AP includes several timer management functions which allows creating, starting, stopping, restarting, etc, operating system timers. CMSIS-RTOS allows both on-shot and periodic timers to be created [4]. Timers are assigned the highest priority by default, however they can still be interrupted by hardware interrupts and are therefore less accurate than hardware timers. Operating system timers are usually a multiple of the SysTick frequency for the Round-Robin scheduler [2].

### Interrupt Service Routines

The CMSIS-RTOS can use interrupt service routines, however, interrupt handler cannot wait for anything [2]. Consequently, mutexes cannot be used, wait functions are not permitted in the interrupt service routine and all timeouts in method parameters need to be set to 0 [4].

# Implementation

The implementation process for this experiment consisted of putting together the work done in the previous 2 experiments, making some minor adjustments to reflect on the new requirements, as well as packaging the whole system to work using the real-time operating system framework. In particular, the various components were to be implemented in 4 separate threads (i.e. temperature sensor, MEMS, keypad and display threads), with message queues used for implementing a thread-safe inter-thread communication. Unless otherwise specified, the initializations and configurations of the various components, as well as their method of operation (including calibration and filtering) remains exactly as in the previous labs. Please refer to the reports for experiments 2 and 3 for more information.

## Interrupts and signals

In the previous 2 experiments, interrupt handler routines were set-up to control the flow of execution of the various components. In particular, global variables were monitored using a constant polling of their values in the body of the main program loop. Once an interrupt occurred, the global variables were flagged, indicating to the main program that an interrupt occurred and therefore, for example, a new temperature sensor reading was available, which was then retrieved, filtered, etc. Unlike the previous experiments, to properly implement an interrupt-driven multi-threaded system, signaling was used instead. In particular, each of the 4 threads were set to wait indefinitely (i.e. using the *‘osWaitForever’* argument) until a signal was set. The signals were set inside the body of each of the interrupt handlers to indicate to the waiting thread(s) that it can move into a ready state and execute its code once the scheduler makes it run. In order to distinguish between the signals, each thread was set to wait to a different signal (i.e. a total of 4 different signals, one per thread). Although not strictly necessary for a proper functioning of the system, the level of the priority of the various interrupts were set to be in the same order as the thread priorities (see the various thread subsections for further information).

## Temperature sensor thread

As in experiment 2, the built-in temperature sensor was used to monitor the processor temperature. The sampling of the temperature was done at a frequency of 50 Hz, as before. However, the major difference from the previous work, is the use of the TIM2 HW timer, as opposed to the previously used SysTick timer (i.e. which can no longer be used, as it is being utilized by the RTOS). The setting up of the TIM2 and NVIC initialization structures was done exactly as was previously done, in experiment 3, with the TIM3/NVIC structures (i.e. which were used for the keypad and 7-segment display). The only difference was in the rate, which was set to 50 Hz by setting the *TIM\_Period* field to 30 and the *TIM\_Prescaler* to 55999 (i.e. via , where 1500 Hz corresponds to the TIM2 counter clock).

This thread was set to wait on a signal, *TEMP\_SENSOR\_READY*, which was set inside the *TIM2\_IRQHandler()* function, to be called every 20 ms when a new temperature reading is ready. Once the reading is retrieved and filtered, it is sent to the display thread via the *temp\_queue* message queue.

This thread was set to have the highest priority (i.e. *osPriorityHigh*) to allow for the earliest possible detection of an alarm and with a custom stack size of a 1000 bytes (to avoid a stack overflow which occurs with the default, 200 bytes, stack size).

## MEMS thread

The core functioning of the MEMS sensor, which is used to calculate the board’s pitch angle, remains exactly the same. The modification of this component is limited to wrapping its functionality in a separate thread, as well as its use of a signal (i.e. *MEMS\_READY*) to wait for new readings, in the same fashion as described above. The read and filtered pitch angle value is sent to the display thread via the *pitch\_queue* message queue.

This thread was set to have the second highest priority (i.e. *osPriorityAboveNormal*) to allow for the earliest possible detection of a change in the orientation which can be reflected on the displays as soon as possible (note that the display thread runs at a much higher frequency then the MEMS updates). Moreover, it was set to use a custom stack size of a 400 bytes (once again, to avoid a stack overflow).

## Keypad thread

The setting up of this component remains unchanged when compared to the previous experiment. Similarly, the keypad scanning and de-bouncing mechanisms also remain as before. However, the keypad’s functioning was slightly altered to work in-line with the new requirements. In particular, the keypad functioning was set to detect a press of the 1-4 buttons (i.e. indicating which LED is to be lit, when in the accelerometer mode), as well as the keys ‘A’ and ‘B’ which are used to indicate that the user chooses the temperature mode and accelerometer mode, respectively. When a valid key is pressed, the choice is communicated to the display thread using the *keypad\_queue* message queue.

Like the other threads, the keypad thread is set to wait on a signal (i.e. *KEYPAD\_READY*), which is set once the TIM3 handler function is called.

The priority of this thread was set to be the second lowest (i.e. *osPriorityNormal)*, which was deemed to be appropriate when compared to the priorities of the other threads. The stack size for this thread was set to 400 bytes.

## Display thread

The display thread is used to display an appropriate message on the 7-segment display (depending on the user selected mode of operation). It is also used to light an LED (and control its brightness), when the system operates in the accelerometer mode. Furthermore, this thread is responsible for implementing the overheating alarm effect (regardless of the mode of operation), in the form of a repeatedly flashing 7-segment display. It therefore combines the functionalities that were implemented in experiment 2 (i.e. PWM-based LED operation) and experiment 3 (i.e. displaying values on a 7-segment display), with the required modifications. The initialization and configuration of the LED and 7-segment display component remain as before, with the expectation of the period of the TIM3 timer whose period (i.e. *TIM\_Period*) and prescalar (i.e. *TIM\_Prescalar)* were set to allow a (little bit too high) desired rate of 7.2 kHz, in order to allow for a nice looking brightness control effect.

The values displayed on the 7-segment display are set in accordance to the most recently selected mode, as well as the most up-to-date values (i.e. for temperature/pitch angle). Similarly, when in accelerometer mode, the LED to be lit is chosen based on the most recently selected value by the user (or set to 1, corresponding to LED at GPIO\_Pin\_12, if no choice has yet to be made). These values are rendered available to the display thread via the 3 message queues that were mentioned above for each of the threads (see the following subsection for further details).

The alarm display, triggered when the detected temperature is above a certain pre-selected threshold, is implemented by turning on and off the 7-segment display for 1 second at a time. When an alarm is triggered, the display shows (during the second in which it is on) the current temperature or pitch angle depending on the current mode. When in the accelerometer mode, an LED is lit, with its brightness varying depending on the pitch angle. When the angle is 0 degrees, the LED is completely off, whereas when the angle is 180 degrees, the LED is the brightest. The LED’s brightness control was implemented using a pulse-width modulation (PWM) technique. The idea is similar, but slightly different from the one used in the second experiment. In particular, the pulse width period was set to 180, to allow 180 different brightness levels (i.e. one per each angle). The duty cycle is then equal to the current pitch angle. As a result, during a single PWM period the LED is on for the amount dictated by the duty cycle (and off for the rest of the period). For example, if the pitch angle was found to be 90 degrees, the LED alternates between being on and off for an equal amounts of time (i.e. half of the PWM period). The PWM is works with the TIM3 timer (as opposed to the SysTick timer, as was the case in experiment 2), which was set to run at 7.2 kHz. Therefore, using the same example as above, the LED would be on for 12.51 ms (i.e. seconds), followed by 12.51 ms of being off, repeatedly.

As this thread operates at a significantly higher frequency and based on the fact that it is best to display values that are the most up-to-date as possible, this thread was set to have the lowest priority of the 4 threads (i.e. *osPriorityBelowNormal*). Its default stack size was set to 400 bytes. Similar to the 3 other threads, the display thread waits on a signal (i.e. *DISPLAY\_READY*), to indicate that it can update the visual feedbacks.

## Inter-thread communication

As was mentioned above, the inter-thread communication was implemented using the message queues (and a memory pool). Each of the three “input” threads (i.e. temperature sensor, MEMS and keypad threads) was assigned their own message queue of size 1. The type of messages that the 3 queues can send is the same. Namely, the messages are of type *Message*, which is type definition of a struct that wraps a single variable of type float. When any of the input threads has a new value (i.e. temperature/angle/pressed key), this value is sent using the appropriate queue. The “output” thread (i.e. display thread), serves as the receiver of these messages. In particular, the thread attempts to receive a message from each of the three queues. In order to not block on the waiting for any of the messages (which would cause unwanted delays in the visual feedback), the reception of the message is set to have a 0 timeout value (i.e. the second argument of the osMessageGet() function is 0). Therefore, if a message had been sent earlier, the display thread will successfully receive it and update the required value to this up-to-date value. Otherwise, if no message had been sent between the current moment and the last time the message was checked, the older value is used.

In order to use message queue, it is necessary to allocate messages with memory (which should be freed after reading the queue). Given that the 3 queues were designed to share the same type of messages (i.e. *Message*), all 3 queues can and are allocated memory from a common memory pool, *mem\_pool*, which is a pool of memory for queues of messages of type *Message*.

Using message queues for communicating across the different threads has a number of benefits: it is rather easy to use, it is easy to extend to use more queues (i.e. have additional message sending threads), as well as more complex message types (i.e. adding additional variables to a message), it allows for easy synchronization (or execution flow) of the threads (i.e. ensuring that the display thread doesn’t block while waiting for values from the other threads) and it ensures the thread-safety of the system (i.e. by avoiding the use of any shared (global) variables).

# Testing and Observations

Testing during this experiment was done in roughly two stages: testing the correct functionality of each module separately (module testing) and testing the correct functionality of the complete system. Testing the functionality includes not only making sure that the modules exhibit the expected behavior, but also ensuring that the intended strict timing is met, by using the Keil real-time debugger to trace the timing of the different threads.

## Module testing

Before making any significant functional changes, each of the modules was minimally modified to work using the RTOS threads (i.e. setting up threads, creating thread handlers, adding signals, etc.). At that point, each of the modules was tested separately (while the other modules were “disabled”), to ensure that the behaviour exhibited by each module remained as before (using the same testing procedures as described in the previous experiments). Once it was observed that a module functions properly, the Keil’s Event Viewer was used in order to study the timing of the threads to ensure that the each module runs when and as often as expected (e.g. the 100 Hz MEMS sensor thread runs roughly every 0.01 seconds).

## System testing

Once it was confirmed that each module works as expected on its own, the required modifications to each module, as well as the inter-thread communication were implemented. The modifications were made incrementally and testing of the system was done to ensure that it exhibits the expected behavior. Once all the modifications were made and the entire system was implemented, the entire system was once again tested and appeared to behave as expected. In particular, using the keypad the user was able to correctly switch between the temperature and accelerometer modes. Once in the accelerometer mode the user was able to use the keypad to switch between the LEDs to be lit. It was observed that the brightness of the selected LED varied correctly, from completely off at 0 degrees to the brightest at 180 degrees. In both modes, the 7-segment display appeared to show the correct values in real-time (i.e. the processor temperature and pitch angle, depending on the selected mode). It was also verified that the displayed pitch angle and temperature appeared sensible (using the hair dryer to vary the processor’s temperature). Finally, it was confirmed that once the temperature reached a temperature equal or greater than the pre-selected threshold temperature, the alarm effect was correctly displayed on the 7-segment display. Moreover, when going below the threshold temperature, normal operation resumed as expected.

Additionally, once again using the Event Viewer, the timing of the thread executions were verified. It was indeed observed that the 4 threads ran at their designed frequencies (i.e. 50 Hz for the temperature sensor, 100 Hz for the MEMS and 7.2 kHz for the display and keypad threads).

# Conclusion

Leveraging work done in the previous experiments, multiple module (inputs/sensors and outputs/displays) were combined into a single multi-threaded system. The system was designed to operate in 2 modes: temperature mode and accelerometer mode. In temperature mode, the system displays the processor temperature on a 7-segment display. In accelerometer mode, the board’s pitch angle is displayed on the 7-segment display, in addition to an onboard LED being lit, with its brightness being a function of the pitch angle. A keypad provides the user with a means with which the modes can be switched. Furthermore, the keypad can be used to select which of the 4 LEDs is to be lit in the accelerometer mode. Using the RTX RTOS, the various modules (the temperature sensor, MEMS, keypad and display(s)) were each implemented in a separate thread. Message queues were used in order to communicate between the threads and ensure a thread-safe behavior. The behavior and timing of the modules were tested both separately, as well as after being combined into a complete system. The overall system appeared to showcase the expected behavior with the various sensors, inputs and outputs working concurrently, following the RTOS imposed strict timing and exhibiting the required functionality.

# References

|  |  |
| --- | --- |
| [1] | Z. Zilic, *ECSE 426 System Services and Final Project,* Montreal, QC, Canada: McGill University, 2015. |
| [2] | B. Nahill, A. Suyyagh, *Real-Time Operating Systems,* Montreal, QC, Canada: ECSE 426-Microprocessors Systems Tutorial 4, 2014. |
| [3] | http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html, "CMCIC-RTOS," ARM Ltd., 20 March 2015. [Online]. [Accessed 23 March 2015]. |
| [4] | "ARMmbed CMSIS RTOS," ARM, [Online]. Available: http://developer.mbed.org/handbook/CMSIS-RTOS. [Accessed 23 March 2015]. |