Brandon Ho 500727531

**COE 718 Final Project**

Introduction:

This project was implemented in two parts which consists of two seperate uVision Project files. The first project titled “Project\_Part1” has three main.c files titled “Part1\_Project.c”, “Part1b\_Project.c” and “Part1c\_Project.c” that implement three different solution to the priority inversion problem introduced in Lab4 using semaphore and mutux concepts (See appendix Block Diagram 1). The second uVision Project file is titled “Final\_Project” which uses the “main.c” file to demonstrates the concept of joinable threads (See appendix Block Diagram 2). For Part 1, each of these three main files “Part1\_Project.c”, “Part1b\_Project.c” and “Part1c\_Project.c” uses the “Mars Rover Priority Inversion” scenario introduced in Lab 4, where the required execution of a “Low- Priority” thread P3 required to complete a task is preempted by the “Higher priority” thread P2, thus permanently blocking the execution of the “Low-Priority” thread P3 and the complete execution of the program. Additionally, each of the three “main.c” files executes the same demonstration project, where the three different input directions of the joystick (LEFT, RIGHT and CENTER) illuminate LED(6)/P2.5, LED(4)/P2.3 or LED(5)/P2.4 respectively, followed by a strobe light pattern on LED(0)/P1.28, LED(1)/P1.29 and LED(2)/P1.31, then subsequently turning off the LED. The illumination of the LED is accomplished by thread P3 (Low Priority), which switch control to thread P1(High priority) to execution the LED light strobe pattern and subsequently switch back to thread P3(Low Priority) to turn off the LED initial illumination. The priority inversion, to stay true to the MARS Rover example, is introduced when thread P2(Normal Priority) tries to preempt thread P1(Low Priority) during the context switch between threads P3 (High Priority) and P1(Low Priority) after the LED strobe pattern before the LED can be turned off. To avoid thread P2 blocking thread P1 during the context switch from thread P3 to thread P1, three different thread management concepts were used. The first is the semaphore concept (using RTOSV1 osSemaphore function) represented by the “Part1\_Project.c” file, the second is the mutux concept (using RTOSV1 osMutuex function) represented in “Part1b\_Project.c” file and the third is a manual coded version of the semaphore concept without RTOSV1 implementation represented by the “Part1c\_project.c” file.The “Part1\_project.c” main file solves the priority inversion problems using the built in osSemaphore functionality in the RTOSV1 API. Four different binary semaphore either allow or block access to each of the four operating threads during program execution to control the execution order of each thread based on the requirements of the program (specifics of implementation will be explained further in methodology section). This prevents thread P2 from blocking thread P1. The “Part1b\_project.c” main file solves the priority inversion problem using the built in osMutux functionality in the RTOSV1 API. A single mutex ensures the appropriate thread management and prevents thread P2 blocking thread P1 (specifics of implementation will be explained further in methodology section). Last, the “Part1c\_project.c” main file solves the priority inversion problem using a custom-made semaphore function in place of the osSemaphore functionality built in RTOSV1. This implementation of the semaphore attempts to correct priority inversion in the demonstration program using a conditional infinite while loop to prevent the higher priority thread P2 from blocking the lower priority thread P1. However, this implementation of the priority inversion demonstration was entirely successful. The reasons for failure of this custom semaphore implementation will be explained in the methodology section. The second Uvision project file titled “Final\_Project” demonstrations the concept of joinable threads, by first creating a new thread titled “main\_thread1”, to which two “sub” threads titled “THREAD1” and “THREAD2” are subsequently created from the “main\_thread1”. “THREAD1” illuminates all the LEDS on Port 1 and Port2 sequentially, then proceeds to increment a counter titled “counter1”. “THREAD2” joins “THREAD1” and turns off all the LED previously illuminated and proceeds to increment a second counter titled “counter2” using the results from “counter1”. Between the switch from “Thread1” execution to “Thread2” execution, the results from the “Thread1” counter function are multiplied with the results from the “Thread2” counter function and stored as the variable “counter3”. After execution of both “Thread1” and “Thread2” are complete, “counter3” is divided by two. It should be noted these two steps occur in the “main\_thread1” thread outside of the “Thread1” and “Thread2” sub threads. Methodology/Review:

How the semaphore as implemented in the Project functions (Part1\_Project.c) functions:

One way of implementing thread management is with the use of a “Semaphore”. A semaphore as is implemented by the osSemaphore function in this project works as a (binary counter) / (flag mechanism) which can only hold values between zero and one. The execution of code can only happen past the threshold of the semaphore function if the semaphore function holds an internal value count of one, otherwise the semaphore function blocks code execution (See Figure 1).

Text, letter

Description automatically generated

Figure 1: Semaphore Logic

The mechanism of the semaphore works as follows (See Figure 2):

1. The “semaphore wait” command checks the internal count of the semaphore before entering a critical section of code. If the internal semaphore count is one, the critical section runs, where the internal counter is decremented to zero. The implication of this logic is that any other threads trying to execute a section of code block by a semaphore function sharing the same internal counter will be blocked.
2. The “semaphore release” command increments the internal count of that particular semaphore function after the critical section is complete, allowing other threads once blocked to enter their respective critical sections of code.

Diagram

Description automatically generated

Figure 2: Semaphore blocking mechanism

In regard to the project, a semaphore could be used to block execution of the code of a thread, such is the case for the priority inversion demonstration when thread P2 tries to preempt thread P1 similar to what is illustrated in Figure 2 (further explanation on how this corrects priority inversion in “Part1\_Project” in Design section).

How the mutex implemented in the Project functions (Part1b\_Project.c) functions:

Another thread management device is the mutex. In the case of this project, the osMutex function works very similarly to the semaphore show in Figure 2, however as implemented in RTOSV1 it has less implementation flexibility. The osSemaphore function has the possibility to be implemented as a counting semaphore allowing a certain number of threads to be run before blocking. In addition, the semaphore can be instantiated to zero, blocking all threads from executing until so deemed necessary (See figure 3). This implementation strategy is used to manage the threads for “Part1\_Project” and correct the priority inversion.

Line chart

Description automatically generated

Figure 3: Alternative logic to implement binary semaphore.

The mutex function is in fact a specific case of a binary semaphore (internal value can only be zero or one) with an initialization of interval value one (ie: execution of code will be allowed to past first instance of the mutex mutex function). However, in the RTOSV1 environment the osMutex function is initialized to a value of one. In this arrangement, the mutex can be described to work like a lock and key scenario and functions identically to the semaphore shown in Figure 1.

The mechanism of the mutex works at follows (See Figure 4):

1. The “mutex wait” command allows access to the critical section, where upon crossing the threshold of the function, the mutex locks. This implies, that no other thread with the same mutex identification will allow access to its critical section.
2. The “mutex release” command unlocks the mutex, allowing other threads to access critical sections of code previously blocked.

Diagram, text

Description automatically generated

Figure 4: Contrast Mutex logic to Binary Semaphore (initialized to count one).

How the semaphore as implemented in the Project functions (Part1c\_Project.c) functions:

The custom semaphore function found in “Part1c\_Project.c” implements the demonstration for the priority inversion using the exact same code implementation/structure as in “Part1\_Project.c” functionality wise except for instead of using the built in osSemaphore function, the custom semaphore function is alternatively called. The custom semaphore function was implemented as a counting semaphore, with two functions that mirror the functionality of the osSemaphore function. The first function titled “wait\_P” which functions like the osSemaphore wait command and the “release\_V” command which functions like the osSemaphore release command.

The two commands “wait\_P” and “release\_V” function as follows (See Figure 5):

1. The “wait\_P” command is initialized to an internal value of zero. When code execution crosses the threshold of the function. The function checks if the internal value is one or zero. Two things can occur (See Figure 6 and 7)
2. If the count is one, code execution is allowed to pass, and the count is decremented by one. This has the effect of blocking any other thread from entering a critical section after the count has been set to zero.
3. If the count is zero, code execution is blocked by use of a conditional while loop that repeats until the count is again set to one.
4. The “release\_V” command increments the internal counter of the semaphore. This implies that after incrementation, the “wait\_P” command potentially blocking a critical section of code in another thread will now all code execution to pass the threshold of its “wait\_P” function.

Text, letter

Description automatically generated

Figure 5: “wait\_P” and “release\_V” command logic

Text, letter

Description automatically generated

Figure 6: Semaphore Logic (Execution through critical section of code)

A picture containing text

Description automatically generated

Figure 7: Semaphore Logic (Blocking mechanism demo)

The principles of Joinable Threads:

The concept of joinable threads works as follow:

1. There is an initial main thread where multiple secondary threads may be instantiated. To synchronize their execution, the thread attributes are set to joinable, which allows the secondary thread to synchronize its operation with another secondary thread possibly dependent on data from the initial secondary thread. This is possible through signaling commands to make each thread aware of the other’s termination and execution.
2. The secondary thread is created and performs some task. Upon completion of the thread task, because the thread attributes of this secondary threads are set to joinable, the termination of an executing thread signals to the operation of another secondary thread to begin execution of the code. This synchronization allows the secondary threads to execute a task in a specific order, enabling subsequent threads to use data processed from previously executing threads during their own respective execution without chance of error.

Design:

PART1: Priority Inversion Solutions

In “Part1\_Project”, the code was implemented using semaphores with the following logic to correct the priority inversion error and demonstrate joystick and LED functionality in a creative way. The program initially exists in a state similar to that of Lab4. Four total threads exist which are P1(High Priority), P2(Low Priority), P3(Low Priority) and main(Low Priority). After LED, Joystick and Kernel initialization, four semaphore must be created each initialized to an interval value of zero. There is one semaphore function required to control code execution in each of the threads (See Figure 7)

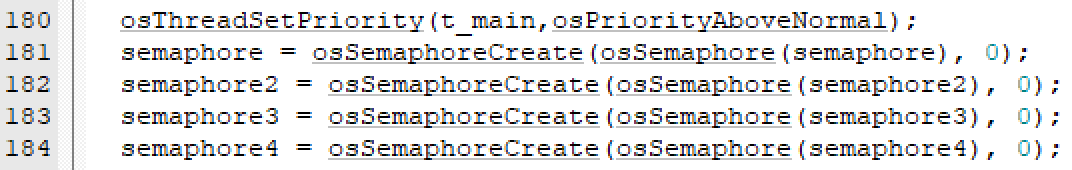


Figure 7: Declare Four semaphores—one for each thread.

In the main, the joystick state is continuously checked in a polling routine (Line 202-219) to determine the current state of the joystick using the “Joystick\_GetState()” command (See Figure 9). This is considered a “polling” routine because the program is coded in a manner in which the CPU is required to constantly check the I/O device (joystick) status. This contrasts an interrupt, where the I/O device interrupts the CPU, to request CPU service. In the main, the first semaphore is declared on line 197 (See Figure 8). This first osSemaphoreWait() command is protected by a flag titled “check” which is toggled after the first joystick command. This is done to ensure that code can only execute pass the semaphore wait function once, to start the “demo” as all the semaphores are initialized to zero. After, the “semaphore3” wait function on line 197 block entry back to the main thread titled “t\_main” (See Figure 9). As the “polling” for joystick input continues in the infinite “for loop” the osSemaphoreRelease() command increments the “semaphore4” internal count to one. This has the implication of providing access to threads in which their critical sections are protected by the semaphore with the identification of “semaphore4”.

Graphical user interface, text, application

Description automatically generated

Figure 9: Polling for joystick state🡪“main” thread access protected by semaphore3 & 4.

Each joystick position illuminates a different LED by creating a series of threads in a manner that replicates the “MARS Rover” priority inversion problem. The LED is illuminated by passing an unsigned integer argument (6U, 5U or 4U based on selected joystick direction) to the switch\_LED\_withthreads() function. Any old threads are terminated using the osThreadTerminate() command (Lines 152 -153) to avoid execution conflicts with the variables during subsequent joystick inputs. The first thread P3 is created (Line 155) with low priority and illuminate the requested LED input provided from the unsigned integer argument passed from the function call (Figure 10).

Graphical user interface, text, application, email

Description automatically generated

Figure 10: Create thread function

Entry to the critical section of Thread P3 is controlled by the semaphore with ID “semaphore3” previously incremented to value one during main thread code execution (Line 120). Consequently, thread execution is forced to execute in this order, regardless of the creation of any higher priority threads because the “semaphore3” osSemaphoreWait() function is the only current semaphore with a value that is not zero and consequently is the only thread allowed to execute (See Figure 11). The LED is subsequently turned “ON” (Line 124) in thread P3 before incrementing the semaphore with ID “semaphore” (Line 128). This implies that threads protected by the semaphore wait functions with IDs of “semaphore” are the only threads that can execute. Again, this enforces thread execution order even if a thread of higher priority attempts to run because the execution of that higher priority thread will be blocked by the respective osSemaphoreWait() command. Subsequently, thread P3 is set to “waitforever” using the osSignalWait() command with signal ID 0x01 (Line 129). The next thread P1 is subsequently created (See Figure 10) after a small delay and executes due to its highest priority status in relation to all other threads.

Text

Description automatically generated

Figure 11: Thread P3 (Low Priority) executes and switch context to Thread P1(High Priority)

Thread P1 execution passes the threshold of the osSemaphoreWait() function with ID “semaphore” as the internal count had been previously incremented to one to purposefully control the order of tread execution to ensure that only thread P1 can execute. Thread P1 runs a strobe light pattern by sequentially illuminating the LEDs on Port 1 after which context is switched back to thread P3 to finish the task of toggling the LED from “ON” to “OFF” (See Figure 12).

Graphical user interface, text, application

Description automatically generated

Figure 12: Thread P1 executing strobe pattern and switching context back to thread P3

Consequently, to complete the context switch, the execution of thread P1 is set to wait (Line 87-89). Referring back to Figure 10, a third thread P2 has been created with a higher priority than that of thread P3. This is the point where without semaphore functions managing thread execution, thread P2 would execute due to its higher priority even though the program requires the execution of thread P1 to occur first inverting the required execution order of the threads. The result of this priority inversion is that thread P2 execution permanently blocks thread P1 execution. To prevent this, thread P2 code must be protected by a semaphore function and consequently a osSemaphoreWait() command blocks access to thread P2 until the execution of thread P3 has completed (See Line 100 Figure 13).

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 13: semaphore wait function with ID “semaphore2” block thread P2 execution

As a result, the semaphore with ID “semaphore2” blocks thread P2 code execution, because the internal count has a value of zero. As all other threads have their execution blocked by their respective semaphores, the only thread that can execute is the low priority thread P3. Thread P3 resumes execution from Line 129 (See Figure 11) and finishes its task of toggling the LED it previously toggled “ON” to “OFF”. Referring to Figure 11, by line 141, the complete task of toggling “ON” and “OFF” the LED selected by the joystick has been completed by thread P3. Consequently, other tasks can now be executed as the risk of priority inversion is no longer present. The semaphore with ID of “semaphore2” is incremented with the command osSemaphoreRelease() which sets the internal value of “semaphore2” to one and consequently allows the execution of the higher priority thread P2 previously blocked by the semaphore wait command with ID “semaphore2”. Thread P2 executes a LED light pattern to demonstrate functionality of the thread and increments the semaphore with ID “semaphore3” command with the osSemaphoreRelease() command. Thread P2 only executes once, because after crossing the threshold of its semaphore, the interval count is decremented and is never incremented again. At this moment, the semaphore with ID “semaphore3” is the only semaphore with a value other than zero, therefore only threads in which their critical section is protected by the semaphore wait function with the “semaphore3” can execute. Thread P1 is set to “waitforever” because of the context switch signal and thread P2 and P3 execution is blocked by their respective semaphore wait functions. Referring to Figure 9, at this moment the only thread that can execute is the main thread. The demonstration program is now complete and consequently is operating in the main thread waiting for the user to enter the next LED to illuminate with a joystick input command. The program will now repeat the demonstration program and do so in a looping operation forever. In “Part1b\_Project”, the code was implemented using a mutex with the following logic to correct the priority inversion error and demonstrate joystick and LED functionality. “Part1b\_Project” demonstrates the same program as mentioned above in the semaphore example however thread management is handled by a mutex instead of a semaphore.



Figure 14: Declare mutex function

Identical to “Part1\_Project”, “Part1b\_Project” implements the same main thread logic to select an LED to illuminate. Contrasting Part1 with the semaphore implementation, only one mutex is required to ensure priority inversion is avoided and the single mutex is initialized in the “main” (See Figure 14). After a LED to illuminate has been selected by the user and the threads are created to execute the LED “demo” task, thread P3 executes first (first thread created) and toggles “ON” the selected LED. Identical to “Part1\_Project”, “Part1b\_Project” switches thread context from thread P3 to thread P1 using the RTOSV1 signaling function osWait(). Subsequently, thread P1 executes next and strobes the LED light pattern (NOTE: thread P1 also has the highest priority and will execute next even if not signaled explicitly to do so). Again, identical in operation to “Part1\_Project”, “Part1b\_Project” requires the execution of thread P3 to occur next to toggle off the selected LED “OFF”, however thread P2 has higher priority than thread P3 causing thread P2 to execute first blocking the execution of thread P1. To correct the priority inversion, upon the initial execution of thread P3 when the LED is toggled “ON”, the addition of a mutex at the beginning of thread P3 execution locks out any other threads protected by a mutex of the same ID (See Figure 15).

Graphical user interface, text, application

Description automatically generated

Figure 15: Mutex implementation on thread P3

As a consequence, the lock will not be released until the mutex release command is called after the full execution of thread P3 has occurred (Line 140). Therefore, if we observe Figure 16, thread P2 will be blocked from execution due to the mutex wait function on Line 96 preventing the priority inversion error from occurring. Subsequently, thread P2 is free to run after the mutex release command is called and continues to operate in accordance with what was explained in the semaphore design explained above. At this point the program has implemented the full “demo” and returns back to the main thread waiting to implement another joystick selection.

Graphical user interface, text, application, email

Description automatically generated

Figure 16: Thread P2 Priority Inversion prevented by mutex lock.

The custom semaphore function implemented as “Part1c\_Project” is functionally identical to the implementation of the semaphore using the osSemaphore function in “Part1\_project”, except that the built-in function has been swapped out for the custom function (See Figure 17).

Graphical user interface, text, application

Description automatically generated

Figure 17: Custom Semaphore function

Function “wait\_P” is equivalent to the osSemaphoreWait() command. It works in according to the principles of a binary semaphore as explained in the Methodology section. When the internal count is one, code execution is allowed to pass the semaphore. However, if the internal count is zero, the custom semaphore blocks code execution using a conditional while loop that executes indefinitely until the interval count is incremented to one. The “release\_V” function is equivalent to the osSemaphoreRelease() command and incremented the interval count by one. However, as coded these custom semaphore functions do not properly implement the demo code. This is due to the fact that once the custom semaphore “wait\_P” function blocks code execution to protect a critical section, the thread scheduler doesn’t realize that the thread is being blocked, as opposed to running a loop operation. Unfortunately, without using the signaling function build into the RTOSV1 environment to explicitly put the threads to the “wait” or “run” condition, the “demo” program gets stuck in an executing thread every time a thread execution must be blocked. The logic required to explicitly signal threads in a generic way that can be used with the concept of the semaphore was unable to be determined for this Final Project. Further research would be required (RTOSVV2) into how a custom semaphore function could be implemented without explicitly controlling which threads execute in advance using signaling functions.

PART2: Joinable Threads

The joinable thread demonstration program “main.c”, fulfills the first required to create a main app where multiple secondary thread will execute as “joinable threads” by first creating a new thread titled “main\_thread1”. From “main\_thread1” two additional threads are created. However, to allow thread execution to occur sequentially in a synchronized order, the attribute parameter of these newly created thread must be set to joinable (See Figure 18).

Text

Description automatically generated

Figure 18: Thread attributes set to joinable for two sub threads

Once set to joinable, the complier is aware of the existence of each thread and can synchronize execution in a manner that ensures that the results produced by the execution of one thread can be used during the execution of a separate thread without errors occurring. Observing the “main\_thread1”, where sub threads titled, “THREAD1” and “THREAD2” are initialized, calling the osThreadJoin() command for each sub threads synchronizes threads execution. Each sub thread calls the same thread function titled “joinable\_thread” executes. Depending on the argument passed during the instantiation of a new sub thread, the LEDs on Port 1 and Port 2 will be sequentially turned “ON” and a counter function executed or turned “OFF” and a different counter function executed based on the results of the first counter. Therefore, the execution order of each thread is critical for the second counter function to execute as intended. This feature of the “demo” shows how sub “THREAD2” depends on sub “THREAD1”, therefore synchronization between sub thread execution is critical. In the “demo” for PART2 of the project, sub thread “THREAD1” sequentially illuminates all the LEDs on Port 1 and 2 and increments a “counter1” to show the first sub thread functioning. After the task of “THREAD1” is complete, the osThreadExit() command signals to the complier that indeed this thread is complete and another sub “THREAD2” can be called. “THREAD2” executes and turns “OFF” all the LEDS on Port 1 and 2 and increments a “counter2” based on “counter1” from sub “THREAD1”. Additionally, to show that indeed the two sub threads are executing from a main thread, there is a third “counter3” that multiples the results from each sub thread and divides the results by a factor of two. It should be noted that without executing the following sub threads using the “joinable” attribute to sync the execution of each sub thread, there is no guarantee that each sub thread would execute in the correct order in all situation possibly leading to program errors.

Experimental Results:

“Part1\_Project” ---semaphore implementation of the “demo”.

Graphical user interface, application

Description automatically generated

Figure A: Joystick position (Right) selected to illuminated LED

Joystick position “RIGHT” was selected by toggling PIN 24 on PORT 1. This illuminates LED 3 on PORT 2 (See Figure A).

Graphical user interface

Description automatically generated

Figure B: Execution Order of Threads using semaphore thread management

Observing the “Event Viewer”, the main run and creates thread P3 which executes the first thread P3. Thread P2 is subsequently created and attempts to run but it blocked. Thread P1 is created in runs as intended. Context is switched from Thread P1 to Thread P3 to finish execution of LED toggle. Thread P2 execution is again blocked to avoid priority inversion as thread P2 has higher priority in comparison to thread P3. Only after thread P3 has finish execution can thread P2 execute. At this point “demo” program is complete and returns to main to accept another joystick input command (See Figure B).

Graphical user interface, text, application

Description automatically generated

Figure C: Debug viewer—Semaphore Implementation

Observing the “debug viewer”, the user is prompted to select a joystick input. After selecting an LED to illuminate thread P3 executes and illuminates an LED to “ON”. Thread P2 attempts to execute but is blocked. Thread P1 executes as intended. Thread P2 is still blocked. Thread P3 finishes the task of toggling the LED by turning the LED “OFF”. The LED “demo” sequence is now complete.

“Part1b\_Project” ---mutex implementation of the “demo”.

The mutex implementation of the “demo” works identical to the semaphore implementation of the “demo”. Images of execution of mutex implementation provided for proof of mutex “demo” functionality (See Figure D, E and F).

Table

Description automatically generated

Figure D: Joystick position (Left) selected to illuminated LED

Joystick position “LEFT” was selected by toggling PIN 26 on PORT 1. This illuminates LED 5 on PORT 2 (See Figure D).

Graphical user interface, table, Excel

Description automatically generated

Figure E: Execution Order of Threads using mutex thread management

Graphical user interface, text, application

Description automatically generated

Figure F: Debug viewer—Mutex Implementation

“Part1c\_Project” ---custom semaphore implementation of the “demo”.

This implementation of the “demo” didn’t work as intended as was discussed in the design. When the custom semaphore function attempts to block execution of a thread, it does so by establishing a conditional while loop, that loops until the value of the semaphore is not zero. However, when code execution enters the while loop, although the thread is effectively blocked, a context switch does not occur as the complier cannot distinguish between a while loop and a blocked area of code. As a result, the complier executes the while loop indefinitely since the thread is never put into a “sem\_wait” running state. Consequently, since the thread context never switches, the value of the semaphore can never be incremented and therefore the program becomes stuck (See Figure G).

Graphical user interface, application, table, Excel

Description automatically generated

Figure G: Thread execution becomes stuck on thread P2

Part2 of Final Project, “main.c” ---demo joinable threads.

Observing “Figure H”, sub thread “THREAD1” toggles all the LEDs on Port 1 and 2 to “ON”.

Graphical user interface, application

Description automatically generated

Figure H: LED switched “ON” by sub thread (THREAD1)

Observing “Figure I”, sub thread “THREAD2 toggles all the LEDs on Port 1 and 2 to “OFF”.

Graphical user interface, application

Description automatically generated

Figure I: LED switched “OFF” by sub thread (THREAD2)

Observing “Figure J”, sub “THREAD1” increments “counter1” and sub “THREAD2” increments “counter2” with the results of “counter1”. Between the execution of each sub thread, the main thread multiplies the results of “counter1” and “counter2”, then divides this value by a factor of two to produce “counter3”. Referencing Figure K, it can be seen in the “RTX RTOS” window that indeed each sub thread has the “joinable” attribute. Additionally, the complier is aware of the required synchronization between sub threads “THREAD1” and “THREAD2” with the “mainthread” and therefore executes in the specified order.

Table

Description automatically generated

Figure J: Counter variable incremented buy each sub thread.

Graphical user interface, text, application, email

Description automatically generated

Figure K: Thread Scheduler

Conclusion:

Implementation of the osSemaphore and osMutex function were able to correct the priority inversion example demonstrated in PART1 using threads of different priority to turn “ON” and “OFF and LED based on the input from the joystick. Although not completely functional, the custom semaphore function demonstrated the mechanism behind the semaphore discussed in the methodology/review section. In the future iteration of the “demo”, using the RTOSV2 environment instead of the RTOSV1 environment, it might be possible for the thread scheduler to be explicitly put into a “wait” state to force a thread context switch during the semaphore block routine (Explained in Methodology and Test Result section). Further research is needed. PART2 of the final project demonstrated the synchronization of threads by implementing threads as “joinable” threads, where two sub threads can be synchronized to accomplish tasks that rely on data processed independently in each thread. This was demonstrated by showing how one sub thread turned “ON” a series of LED sequentially, while another thread turned the same series of LEDs “OFF” without conflict. Similarly, since sub “THREAD2” is dependent on the result of sub “THREAD1”, a counter function was shown that emphasized this fact. Additionally, a third counter located in the thread where each sub thread was initialized showed the interaction amongst all three threads.

References:

1. RTOSV2, <https://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html>, 2020.
2. RTOSV1, <https://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html>, 2020.
3. Printf(), <https://www.keil.com/support/man/docs/jlink/jlink_trace_itm_viewer.htm>, 2020.
4. uVision Debugger, <https://www.keil.com/support/docs/4126.htm>, 2020.
5. Semaphores Functions, <https://www.geeksforgeeks.org/semaphores-in-process-synchronization/>, 2020.
6. Mutex Functions, <https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>, 2020.

Appendix:

Diagram

Description automatically generated

Block Diagram 1: PART1 priority inversion “demo” program flow

Diagram

Description automatically generated

Block Diagram 2: Part2 Joinable Threads execution flow