RYERSON UNIVERSITY

Faculty of Engineering and Architectural Science

Department of Electrical and Computer Engineering

Course Number	COE 718
Course Title	Embedded Systems Design
Semester/Year	F2020
Assignment Name	Final Project
Instructor Name	Saber Amini
Section No	03

Submission Date	12/08/2020
Due Date	12/08/2020

Name	**Student ID**	**Signature***
Vatsal Shreekant	500771363	[illegible]

Table of Contents

Introduction

This project was executed in two different project files. The first project titled “part1” has three main.c files titled “part1_a.c”, “part1_b.c” and “part1_c.c” that implement three different solutions to the priority. The second project file is titled “Project”. For Part 1 of this project, each of these three main files “part1_a.c”, “part1_b.c” and “part1_c.c” implement the concept of “Mars Rover Priority Inversion” from Lab 4. The execution of thread P3 required the execution of a pre-empted thread P2. This in turn would block the execution of thread P3 completely. It should be noted that thread P3 is labelled as a low priority and P2 as high priority.

The joystick directions left, right and center were implemented by lighting the following LEDs:

- LED(0) → P1.28
- LED(1) →P1.29
- LED(2) →P1.31
- LED(4) →P2.3
- LED(6) →P2.5
- LED(5) →P2.4

The priority inversion algorithm can be best described when a thread with a normal priority (P2) pre-empts a thread with a low priority (P1) when the program makes a transition between a high priority (P3) and a low priority thread. In order to bypass the process of P2 blocking P1, the three recommended solutions as per the lab requirements were implemented. The following are the recommended solutions:

- Semaphores
- Mutex
- Resemble Semaphores

The “Semaphore” file resolves the priority inversion by using the built in osSemaphore() function as provided in the API. Multiple semaphores implemented in this section either render or barricade access to each of the respective threads during program execution. The “Mutex” file resolves the priority inversion using the built in osMutex() function as provided in the API. One mutex leads to the correct thread organization and thus averts the thread P2 blocking procedure. Finally, the “Resemble Semaphores” file resolves the priority inversion by via a custom semaphore function unlike the osSemaphore() function as provided in the API. The custom semaphore tries to rectify the priority inversion issue by implementing a while loop. This blocks the thread with a higher priority P2 from blocking the thread with a lower priority.

The “Joinable Threads” part of the project is implanted in the project titled “Project”. The concept behind this part includes the creation of two threads from a main file. Upon the execution of the first thread, the LEDs light up on both the ports. This then updates the counter value. Upon the execution of second thread, the respective LEDs are turned off on both the ports. This then updates the counter value of the second counter. The final calculation is then performed using the values from both the counter variables that were updated upon the execution of both the threads.

Past Work/Review

Implementation of Semaphores:

Firstly, the wait command on the semaphore will review the internal semaphore count. It is vital to initiate this step before jumping to the main part of the code. If the count of the internal semaphore equates to ‘one’, the main function then initiates. The internal counter in this section is updated to the value ‘zero’. The significance of this section is that if either of the declared threads try to share the blocking functionality of the semaphore function, they will then be restricted and thus rendered blocked.

Secondly, the release command on the semaphore will augment the count of the respective semaphore. This is done once the main section of the code is executed and thus, the remaining threads that were previously blocked, would now be able to jump to their part in the code. To understand the structure of the semaphore implementation, please refer to figure 1.

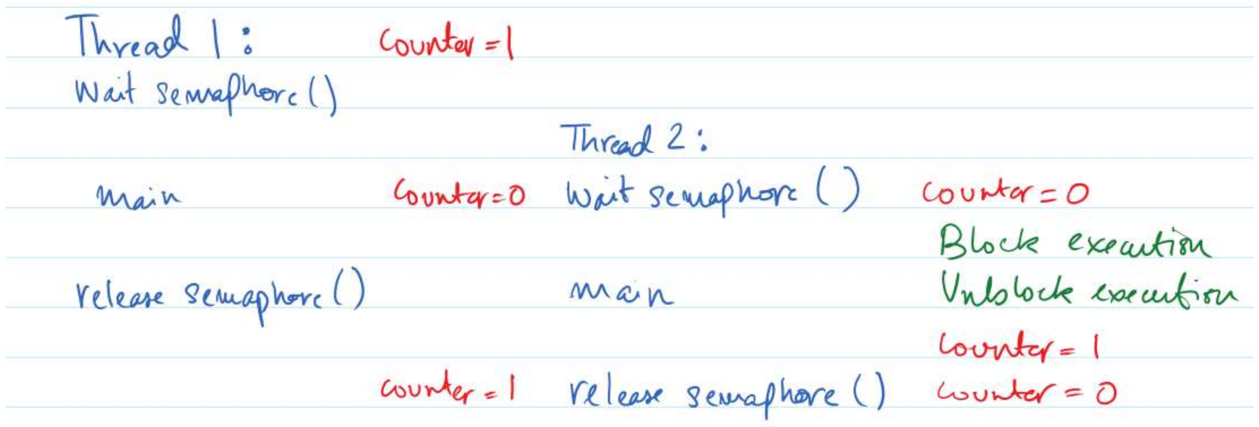


Fig. 1. Semaphore structure

Implementation of Mutexes:

Firstly, the wait command on the mutex will initiate control of the mutex to the access to the main section. The mutex then becomes immutable once it goes across the threshold of the program. Meaning that neither of the threads/processes with matching mutex declaration will render access to their main section.

Secondly, the release command on the mutex will expose the mutex, hence, rendering other threads/processes to gain control over the main sections of code that was previously unavailable. To understand the structure of the semaphore implementation, please refer to figure 2.

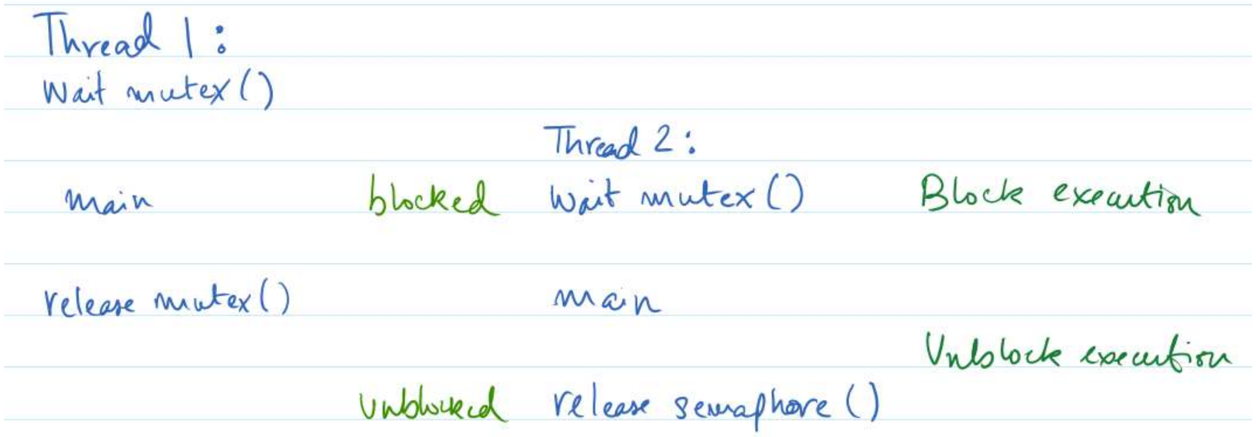


Fig. 2. Mutex Structure

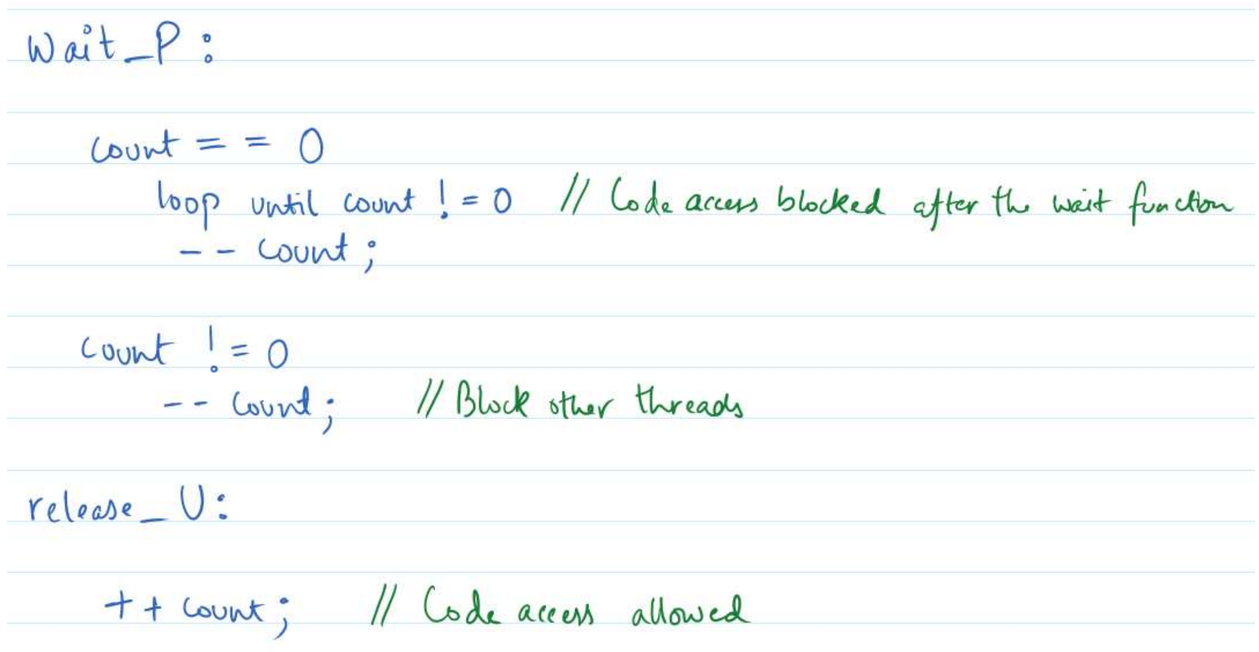
<u>Implementation of Resemble Semaphores:</u>

The custom semaphore function created in "part1_c.c" has identical functionality as "part1_a.c". The only difference between the two semaphore representations is that in "part1_c.c", the osSemaphore() function is executed, whereas in "part1_a.c", a custom made semaphore function is executed. In the case of 'Resemble Semaphores', two functions in relation with the "wait" and "release" functionalities were initiated: "wait_P" (similar to osSemaphore() wait) and "release_V" (similar to osSemaphore() release). The implementation of the two functions are as follows:

The "wait_P" function is run, and the initial value is assigned as "zero" for the internal count. Upon exceeding the baseline of the function, the program then checks if the initial value is still fixed at either zero or one. For the two possible scenarios, the following occurs:

- When the count value is equal to one, the execution of the function is rendered as 'okay' to proceed. Subsequently, the value of the count decreased by one. This process disables either of the threads from accessing the main section of the code.
- When the count value is equal to zero, the execution of the function is not allowed and thus is not allowed to proceed. This is done with the implementation of a while loop that reiterates itself until the count value is set back to one.

The "release_V" function upon execution, increases the semaphore's internal count value. Thus, after the value of the internal count of the semaphore is updated, the "wait_P" function that could have initiated the blocking procedure from either of the threads would now allow the passing of the threads to gain access to main section of the code. To understand the structure of the resemble semaphore implementation, please refer to figure 3.



```
Wait_P:

   count == 0
      loop until count != 0   // Code access blocked after the wait function
       -- count;

   count != 0
      -- count;      // Block other threads

release_U:

   ++ count;      // Code access allowed
```

Fig. 3. Resemble Semaphore Structure

<u>Implementation of Joinable Threads:</u>

For the purposes of Joinable threads, various threads will be declared within a main thread. To ensure that these threads run in conjunction with each other, the thread parameters are initiated to joinable. This leads to the sub threads to coordinating their functions with other threads. This can be done via certain operation so that all the threads become cognizant of their implementation or termination.

The subsequent threads are created; each performing certain tasks. Once the task is completed by the respective thread, the thread terminates its functions and indicates to the upcoming thread to initiate its execution. This ‘joinable’ function in between the threads leads to better organization and allows the subsequent threads to implement the given task in a set directive. This also reduces the percentage of any error arising from the data being passed on from the previous threads.

Methodology

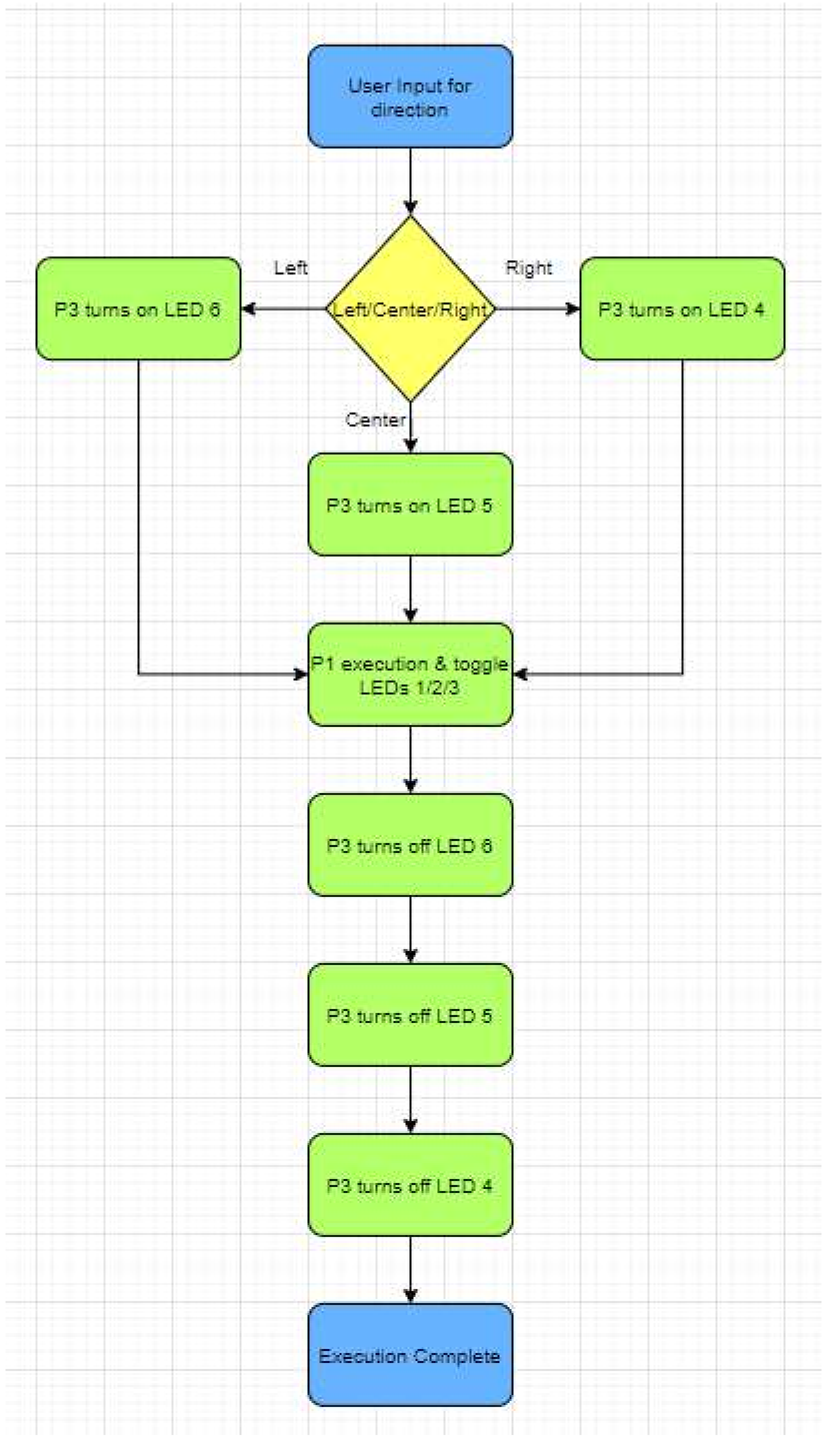


Fig. 4. Block diagram for Priority Inversion

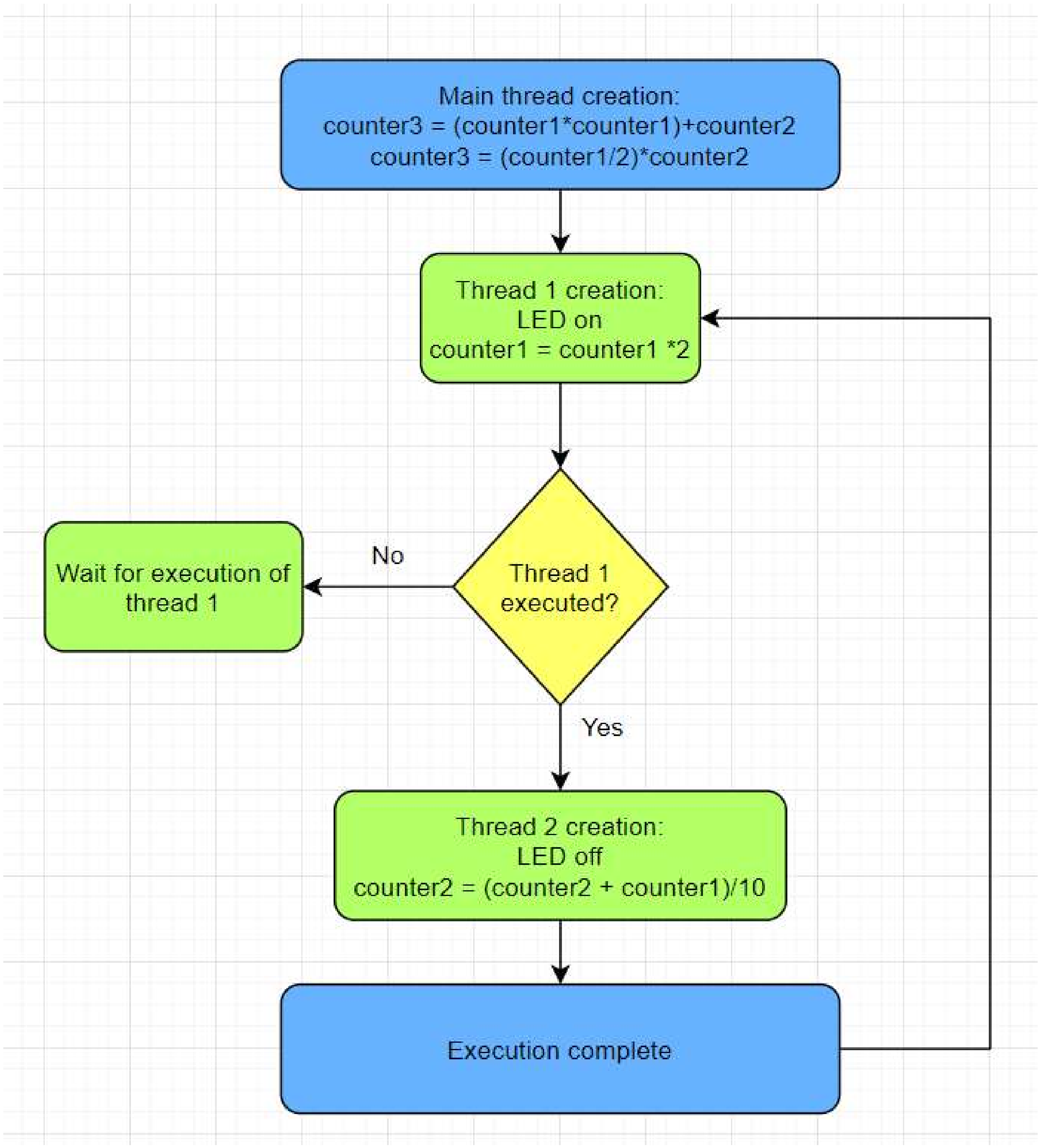

Main thread creation:
counter3 = (counter1*counter1)+counter2
counter3 = (counter1/2)*counter2
Thread 1 creation:
LED on
counter1 = counter1 *2
Thread 1 executed?
No
Wait for execution of thread 1
Yes
Thread 2 creation:
LED off
counter2 = (counter2 + counter1)/10
Execution complete

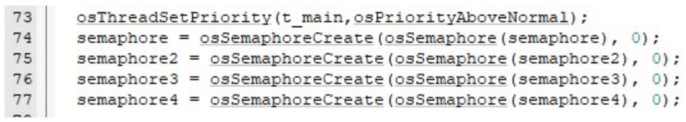
Fig. 5. Block diagram for Joinable Threads

Design

Priority Inversion

The “part1_a.c”, section was built with the help of semaphores to tackle the issues of priority inversion, joysticks and LEDs. A total of four threads were initialized with the following priority:

- P1→ (High)
- P2→ (Low)
- P3→ (Low)
- Thread main→ (Low)

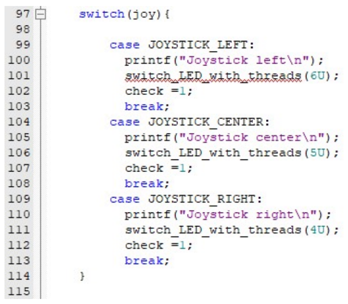


```
osThreadSetPriority(t_main,osPriorityAboveNormal);
semaphore = osSemaphoreCreate(osSemaphore(semaphore), 0);
semaphore2 = osSemaphoreCreate(osSemaphore(semaphore2), 0);
semaphore3 = osSemaphoreCreate(osSemaphore(semaphore3), 0);
semaphore4 = osSemaphoreCreate(osSemaphore(semaphore4), 0);
```

Fig. 6. Semaphore declaration for four threads

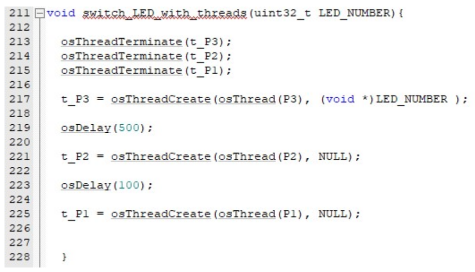
To follow the logic of priority inversion using semaphores in conjunction with the joystick function, refer to the following steps:

- Semaphore declaration.
- Initiation of osSemaphoreWait() function.
- Semaphores count initialized to ‘zero’ value.
- Before semaphore4 execution, semaphore3 blocks the code execution and jumps to the main.
- Semaphore4 internal count incremented to 1.
- Semaphore4 gains access to the remaining threads.
- Based on the joystick direction, the switch_LED_with_threads() function executes (figure 7).
- Expired threads removed via osThreadTerminate() (figure 8).
- Creation of thread P3 (figure 8).
- Semaphore 3 allowed to execute.
- Thread P3 turns the LED on the semaphore value incremented.
- Thread P3 will wait via the function osSignalWait() (figure 9).
- Creation and execution of thread P1 because of its priority level (figure 8).
- Thread P1 turns on the LEDs on Port 1 (figure 10).
- Switch to thread P3 to turn the LED off & P1 waits (figure 10).
- Thread P2 created and blocks the execution of P1 due to its priority.
- Switch to P3 for turning the LED off.
- Thread P2 executes by strobing the LEDs.
- Thread P1 waits; threads P2/P3 execution restricted by their semaphores.
- Simulation complete and user selects the next joystick direction.



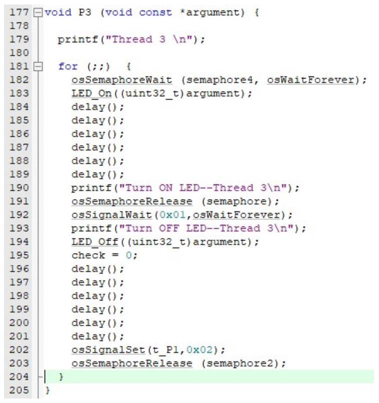
```
    switch(joy){

        case JOYSTICK_LEFT:
          printf("Joystick left\n");
          switch_LED_with_threads(6U);
          check =1;
          break;
        case JOYSTICK_CENTER:
          printf("Joystick center\n");
          switch_LED_with_threads(5U);
          check =1;
          break;
        case JOYSTICK_RIGHT:
          printf("Joystick right\n");
          switch_LED_with_threads(4U);
          check =1;
          break;
    }

```

Fig. 7. Joystick directions



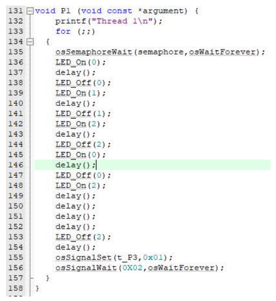
```
void switch_LED_with_threads(uint32_t LED_NUMBER){

  osThreadTerminate(t_P3);
  osThreadTerminate(t_P2);
  osThreadTerminate(t_P1);

  t_P3 = osThreadCreate(osThread(P3), (void *)LED_NUMBER );

  osDelay(500);

  t_P2 = osThreadCreate(osThread(P2), NULL);

  osDelay(100);

  t_P1 = osThreadCreate(osThread(P1), NULL);


  }
```

Fig. 8. Thread creation



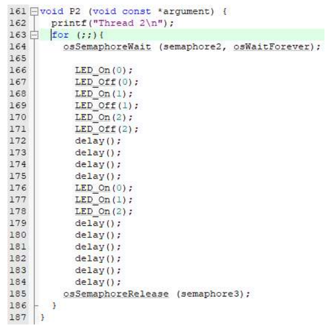
```
void P3 (void const *argument) {

  printf("Thread 3 \n");

  for (;;)  {
    osSemaphoreWait (semaphore4, osWaitForever);
    LED_On((uint32_t)argument);
    delay();
    delay();
    delay();
    delay();
    delay();
    delay();
    printf("Turn ON LED--Thread 3\n");
    osSemaphoreRelease (semaphore);
    osSignalWait(0x01,osWaitForever);
    printf("Turn OFF LED--Thread 3\n");
    LED_Off((uint32_t)argument);
    check = 0;
    delay();
    delay();
    delay();
    delay();
    delay();
    delay();
    osSignalSet(t_P1,0x02);
    osSemaphoreRelease (semaphore2);
  }
}
```

Fig. 9. Thread P3 execution and transition to thread P1



```
void P1 (void const *argument) {
    printf("Thread 1\n");
    for (;;)
  {
    osSemaphoreWait(semaphore,osWaitForever);
    LED_On(0);
    delay();
    LED_Off(0);
    LED_On(1);
    delay();
    LED_Off(1);
    LED_On(2);
    delay();
    LED_Off(2);
    LED_On(0);
    delay();
    LED_Off(0);
    LED_On(2);
    delay();
    delay();
    delay();
    delay();
    LED_Off(2);
    delay();
    osSignalSet(t_P3,0x01);
    osSignalWait(0X02,osWaitForever);
  }
}
```

Fig. 10. Thread P1 toggling with LEDs and transition to thread P3

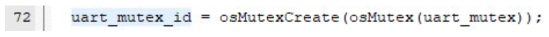


```c
void P2 (void const *argument) {
  printf("Thread 2\n");
  for (;;){
    osSemaphoreWait (semaphore2, osWaitForever);

      LED_On(0);
      LED_Off(0);
      LED_On(1);
      LED_Off(1);
      LED_On(2);
      LED_Off(2);
      delay();
      delay();
      delay();
      delay();
      LED_On(0);
      LED_On(1);
      LED_On(2);
      delay();
      delay();
      delay();
      delay();
      delay();
      delay();
    osSemaphoreRelease (semaphore3);
  }
}
```

Fig. 11. OsSemaphoreWait() blocking thread P2 execution

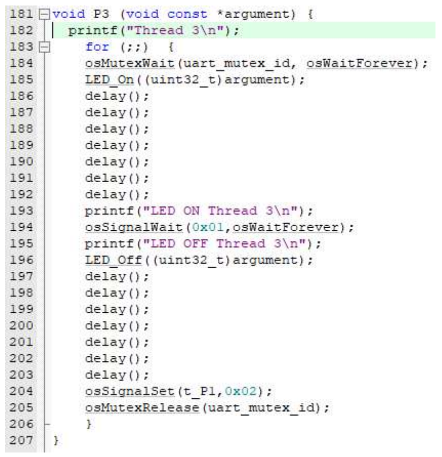
The “part1_b.c”, executes similar to “part1_a.c”. Refer to the following steps:

- Once the user selects any LED light, the thread P3 executes & LED is turned on.
- Thread P3 switches to thread.
- Thread P1 executes next and LEDs toggle.
- Thread P3 executes and turns the LED off.
- Thread P2 executes before thread P1.
- The mutex placed before thread P3 blocks the other threads (figure 13).
- The mutes block P2 execution & thus saving priority inversion (figure 14).
- Post mutex release, P2 executes.
- Simulation complete and user selects the next joystick direction.



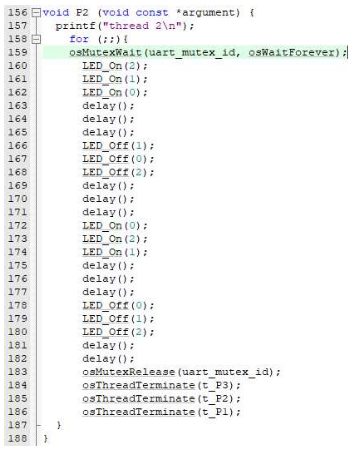
```c
uart_mutex_id = osMutexCreate(osMutex(uart_mutex));
```

Fig. 12. Declare mutex function



```c
void P3 (void const *argument) {
  printf("Thread 3\n");
    for (;;)  {
    osMutexWait(uart_mutex_id, osWaitForever);
    LED_On((uint32_t)argument);
    delay();
    delay();
    delay();
    delay();
    delay();
    delay();
    delay();
    printf("LED ON Thread 3\n");
    osSignalWait(0x01,osWaitForever);
    printf("LED OFF Thread 3\n");
    LED_Off((uint32_t)argument);
    delay();
    delay();
    delay();
    delay();
    delay();
    delay();
    delay();
    osSignalSet(t_P1,0x02);
    osMutexRelease(uart_mutex_id);
    }
}
```

Fig. 13. Mutex addition before thread P3

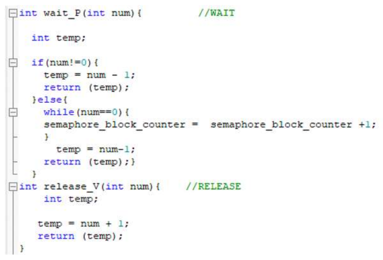


```c
void P2 (void const *argument) {
  printf("thread 2\n");
    for (;;){
    osMutexWait(uart_mutex_id, osWaitForever);
      LED_On(2);
      LED_On(1);
      LED_On(0);
      delay();
      delay();
      delay();
      LED_Off(1);
      LED_Off(0);
      LED_Off(2);
      delay();
      delay();
      delay();
      LED_On(0);
      LED_On(2);
      LED_On(1);
      delay();
      delay();
      delay();
      LED_Off(0);
      LED_Off(1);
      LED_Off(2);
      delay();
      delay();
      osMutexRelease(uart_mutex_id);
      osThreadTerminate(t_P3);
      osThreadTerminate(t_P2);
      osThreadTerminate(t_P1);
  }
}
```

Fig. 14. Priority Inversion on Thread P2 avoided due to mutex

The “part1_c.c”, executes similar to “part1_a.c”. Refer to the following steps:

- Creation of resembling semaphore function (figure 15).
- Internal count = 1 → code execution.
- Internal count = 0 → code execution blocked until count updated to 1.

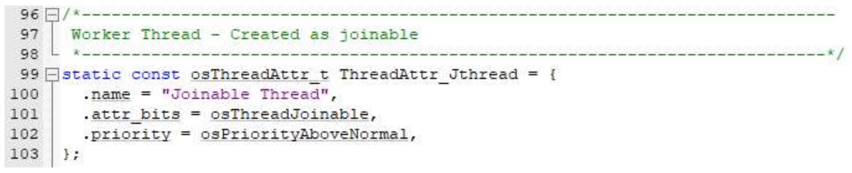


```c
int wait_P(int num){          //WAIT

  int temp;

  if(num!=0){
    temp = num - 1;
    return (temp);
  }else{
    while(num==0){
    semaphore_block_counter =  semaphore_block_counter +1;
    }
      temp = num-1;
    return (temp);}
  }
int release_V(int num){    //RELEASE
    int temp;

   temp = num + 1;
   return (temp);
}
```

Fig. 15. Resembling Semaphore function

Joinable Threads

The “part1_a.c”, section was built with the help of various threads that execute as joinable-threads. Thread main_thread1 is created and subsequently, two more threads are created. The thread parameters are set to osThreadJoinable (figure 16).



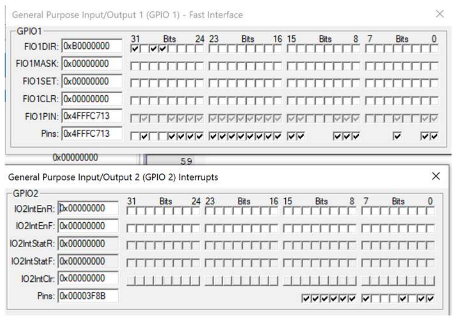
```c
/*----------------------------------------------------------------------------
 Worker Thread - Created as joinable
 *---------------------------------------------------------------------------*/
static const osThreadAttr_t ThreadAttr_Jthread = {
   .name = "Joinable Thread",
   .attr_bits = osThreadJoinable,
   .priority = osPriorityAboveNormal,
};
```

Fig. 16. Joinable thread parameters

Experimental Results

Priority Inversion

Part a:



General Purpose Input/Output 1 (GPIO 1) - Fast Interface

GPIO1

FIO1DIR:	0xB0000000
FIO1MASK:	0x00000000
FIO1SET:	0x00000000
FIO1CLR:	0x00000000
FIO1PIN:	0x4FFFC713
Pins:	0x4FFFC713

General Purpose Input/Output 2 (GPIO 2) Interrupts

GPIO2

IO2IntEnR:	0x00000000
IO2IntEnF:	0x00000000
IO2IntStatR:	0x00000000
IO2IntStatF:	0x00000000
IO2IntClr:	0x00000000
Pins:	0x00003F8B

Fig. 17. Joystick position set to demonstrate the 'right' direction

In figure 17 the joystick position is set to "RIGHT". This was done by toggling PIN 24 on PORT 1. LED 3 is turned on PORT 2.

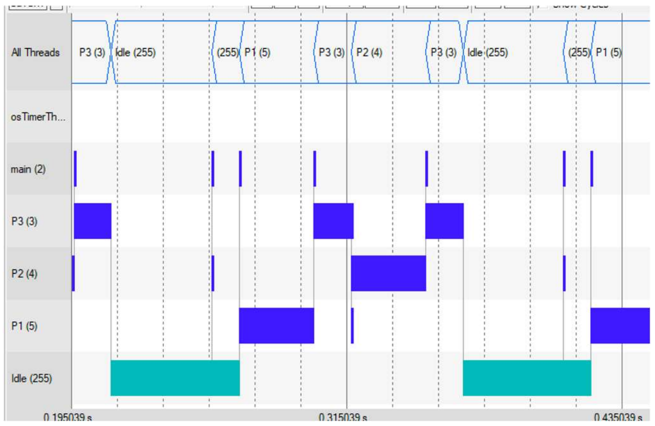
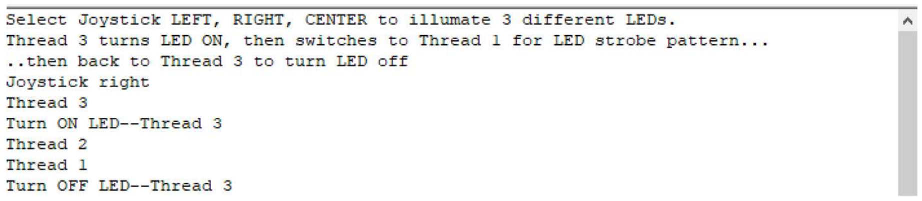


Fig. 18. Event viewer window demonstrating thread execution

In figure 18, thread P1 is created and executes. Switch between P1 and P3. P2 execution blocked. Thread P3 executes followed by P2 execution.

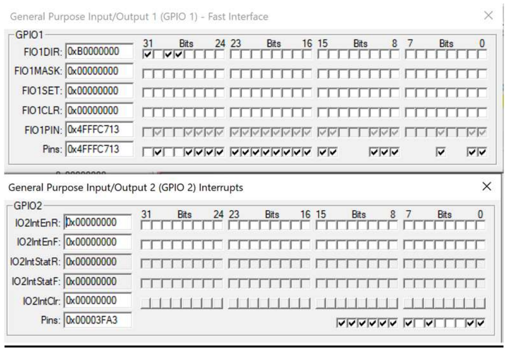


```
Select Joystick LEFT, RIGHT, CENTER to illumate 3 different LEDs.
Thread 3 turns LED ON, then switches to Thread 1 for LED strobe pattern...
..then back to Thread 3 to turn LED off
Joystick right
Thread 3
Turn ON LED--Thread 3
Thread 2
Thread 1
Turn OFF LED--Thread 3
```

Fig. 19. Debug window (Semaphore)

In figure 19, thread P3 executes (LED on); thread P2 tries to execute but gets blocked; thread P1 executes; thread P3 turns the LED off.

Part b:


General Purpose Input/Output 1 (GPIO 1) - Fast Interface

GPIO1

FIO1DIR:	0xB0000000
FIO1MASK:	0x00000000
FIO1SET:	0x00000000
FIO1CLR:	0x00000000
FIO1PIN:	0x4FFFC713
Pins:	0x4FFFC713

General Purpose Input/Output 2 (GPIO 2) Interrupts

GPIO2

IO2IntEnR:	0x00000000
IO2IntEnF:	0x00000000
IO2IntStatR:	0x00000000
IO2IntStatF:	0x00000000
IO2IntClr:	0x00000000
Pins:	0x00003FA3

Fig. 20. User selecting left direction on the Joystick

In figure 20, the mutex function simulates in conjunction with the semaphore function. The user selected the left direction by switching PIN 26 on.

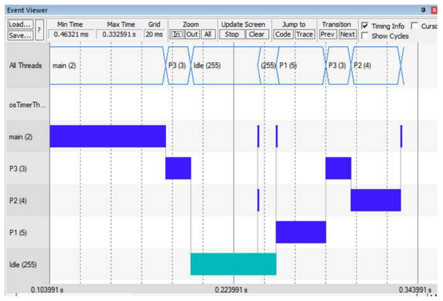
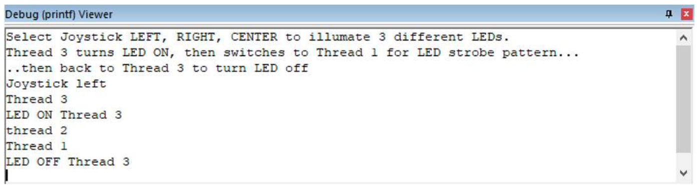


Fig. 21. Event viewer window demonstrating thread execution


Debug (printf) Viewer

```
Select Joystick LEFT, RIGHT, CENTER to illumate 3 different LEDs.
Thread 3 turns LED ON, then switches to Thread 1 for LED strobe pattern...
..then back to Thread 3 to turn LED off
Joystick left
Thread 3
LED ON Thread 3
thread 2
Thread 1
LED OFF Thread 3
```

Fig. 22. Debug window (Mutex)

Part c:

The event viewer window did not display the correct functionality of this part of priority inversion. Hence, the while loop keeps on running for an indefinite amount of time.

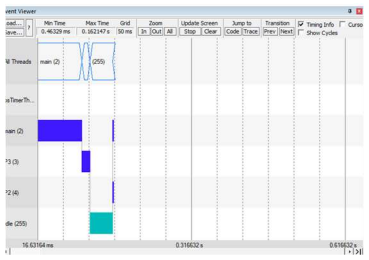
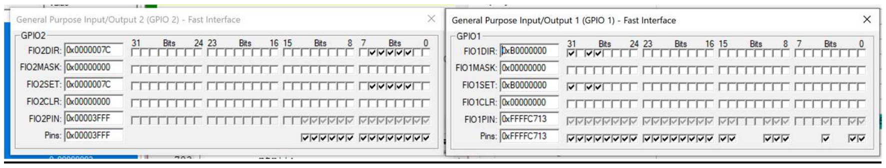


Fig. 23. Event viewer: P2 thread not executing further

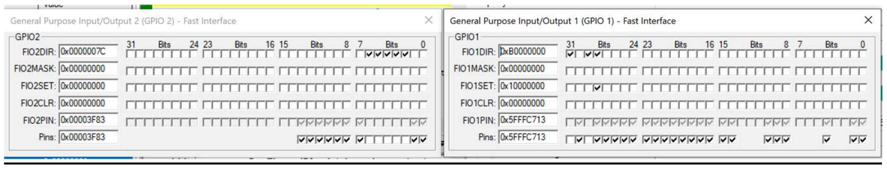
Joinable Threads



```
General Purpose Input/Output 2 (GPIO 2) - Fast Interface
GPIO2
FIO2DIR:  0x0000007C
FIO2MASK: 0x00000000
FIO2SET:  0x0000007C
FIO2CLR:  0x00000000
FIO2PIN:  0x00003FFF
Pins:     0x00003FFF

General Purpose Input/Output 1 (GPIO 1) - Fast Interface
GPIO1
FIO1DIR:  0xB0000000
FIO1MASK: 0x00000000
FIO1SET:  0xB0000000
FIO1CLR:  0x00000000
FIO1PIN:  0xFFFFC713
Pins:     0xFFFFC713
```

Fig. 24. LED switched "ON" by sub thread (THREAD1)

In figure 24, thread 1 switches the LEDs on, on both the ports.



```
General Purpose Input/Output 2 (GPIO 2) - Fast Interface
GPIO2
FIO2DIR:  0x0000007C
FIO2MASK: 0x00000000
FIO2SET:  0x00000000
FIO2CLR:  0x00000000
FIO2PIN:  0x00003F83
Pins:     0x00003F83

General Purpose Input/Output 1 (GPIO 1) - Fast Interface
GPIO1
FIO1DIR:  0xB0000000
FIO1MASK: 0x00000000
FIO1SET:  0x10000000
FIO1CLR:  0x00000000
FIO1PIN:  0x5FFFC713
Pins:     0x5FFFC713
```

Fig. 25. LED switched "OFF" by sub thread (THREAD2)

In figure 25, thread 2 switches the LEDs off, on both the ports.

Conclusions

The execution of both the Semaphore and the Mutex function calls were executed without any issues arising. This was observed upon the selection of the joystick direction, toggling of the LEDs from on to off. However, the resembling semaphore function did not function as per the specifications. It was observed from the event viewer window that the correct functionality of this part was not displayed and was not in compliance with the priority inversion issue. It was noted that the while loop kept on running for an indefinite amount of time.

For the execution of Joinable threads part of this project, it was noted that both the threads declared were functioning as prescribed in the lab manual. This section was built with the help of various threads that execute as joinable threads. Thread main_thread1 was created and subsequently, two more threads were created. The thread parameters were set to osThreadJoinable and hence the joinable threads parameter requirement was satisfied. As noted from the GPIO ports windows, it was observed that thread 1 switched the LEDs on, on both the ports. Simultaneously, thread 2 switched the LEDs off, on both the ports.

References

1) Mutex Functions, https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/, 2020.
2) Printf(), https://www.keil.com/support/man/docs/jlink/jlink_trace_itm_viewer.htm, 2020.
3) Semaphores Functions, https://www.geeksforgeeks.org/semaphores-in-process-synchronization/, 2020.