ELEC422

Microprocessor Systems
Assignment 3
Real-Time Operating System Implementation on FRDM-K64F

Khanthapak Thaipakdee

Department of Electrical Engineering and Electronics,
University of Liverpool,
Brownlow Hill, Liverpool L69 3GJ, UK

Table of Contents

Contents **Page**

Table of Figures

Figure **Page**

1. Introduction

This project aims to develop an extensive subsystem for the Kinetis K64F microcontroller, encompassing display, LED, and ADC subsystems to enable efficient user interaction, LED control, and analogue-to-digital conversion functionality.

In Part A begins by establishing four threads and integrating two ISRs. Thread 1 manages bidirectional PC-K64F communication via a Virtual Communications Port, handling keyboard input, echoing messages, and decoding signals. Thread 2 handles K64F-PC communication using a Zero Copy Mailbox. ISRs linked with Thread 3 and Thread 4 monitor switch inputs (SW2 and SW3), signaling Thread 2 via the Mailbox upon status changes, enhancing user feedback.

Part B expands the system with Thread 5, tasked with controlling RGB LEDs based on commands from Thread 1. LED states toggle according to user input, enabled by flag-based communication between threads.

Part C further extends functionality with two additional threads: Thread 6 for periodic ADC readings from Pot 1 and Thread 7 for on-demand ADC readings from Pot 2. Thread 1 coordinates Thread 6's sampling rate and requests readings from Thread 7. Inter-thread communication utilizes flags and message passing, ensuring efficient data exchange without reliance on global variables.

The project leverages Real-Time Operating System (RTOS) features on the FRDM-K64F Board, employing inter-process communication mechanisms provided by the RTX5 Operating system under the CMSIS-RTOS2 wrapper, enhancing software efficiency compared to Assignment 1.

2. Inter thread communications

Figure 1 depicts a diagram of inter-thread communications, involving Thread 1 through Thread 7, two ISRs, a mutex named ADCmutex, a memory block named memblock, and message queues named msgqueue and queue_sampleperiod. The msgqueue facilitates communication between Thread 1, Thread 3, Thread 4, Thread 6, or Thread 7 with Thread 2, used for displaying characters input by the user or strings to the PC, while queue_sampleperiod transfers the new sample period input by the user from Thread 1 to Thread 6.

When SW2 is pressed or released, it triggers ISR1, setting a flag to Thread 3, and similarly, when SW3 is pressed or released, it triggers ISR2, setting a flag to Thread 4. During the printstr function call, memblock is allocated, and after printing to the PC in Thread 2, memblock is freed, indicating every thread that uses the printstr function allocates memory.

Thread 1 echoes input characters back to the display via Thread 2, while Threads 3 and 4 send the switch state as a string to the display via Thread 2. Threads 6 and 7 transmit the ADC value of Pot 1 or Pot 2 to the display via Thread 2. In Thread 1, when the user inputs 'R' or 'r' or 'G' or 'g' or 'B' or 'b', it sets flags to Thread 5 to operate the LED based on the input character. Additionally, when the user inputs '2', Thread 1 sets a flag to Thread 7.

Moreover, in Thread 6 and Thread 7, they acquire ADCmutex before performing ADC conversion and release ADCmutex after finishing the conversion, ensuring only one thread performs ADC conversion at a time. This setup ensures smooth communication and coordination among threads, ISRs, and mutexes, facilitating the efficient operation of the embedded system.

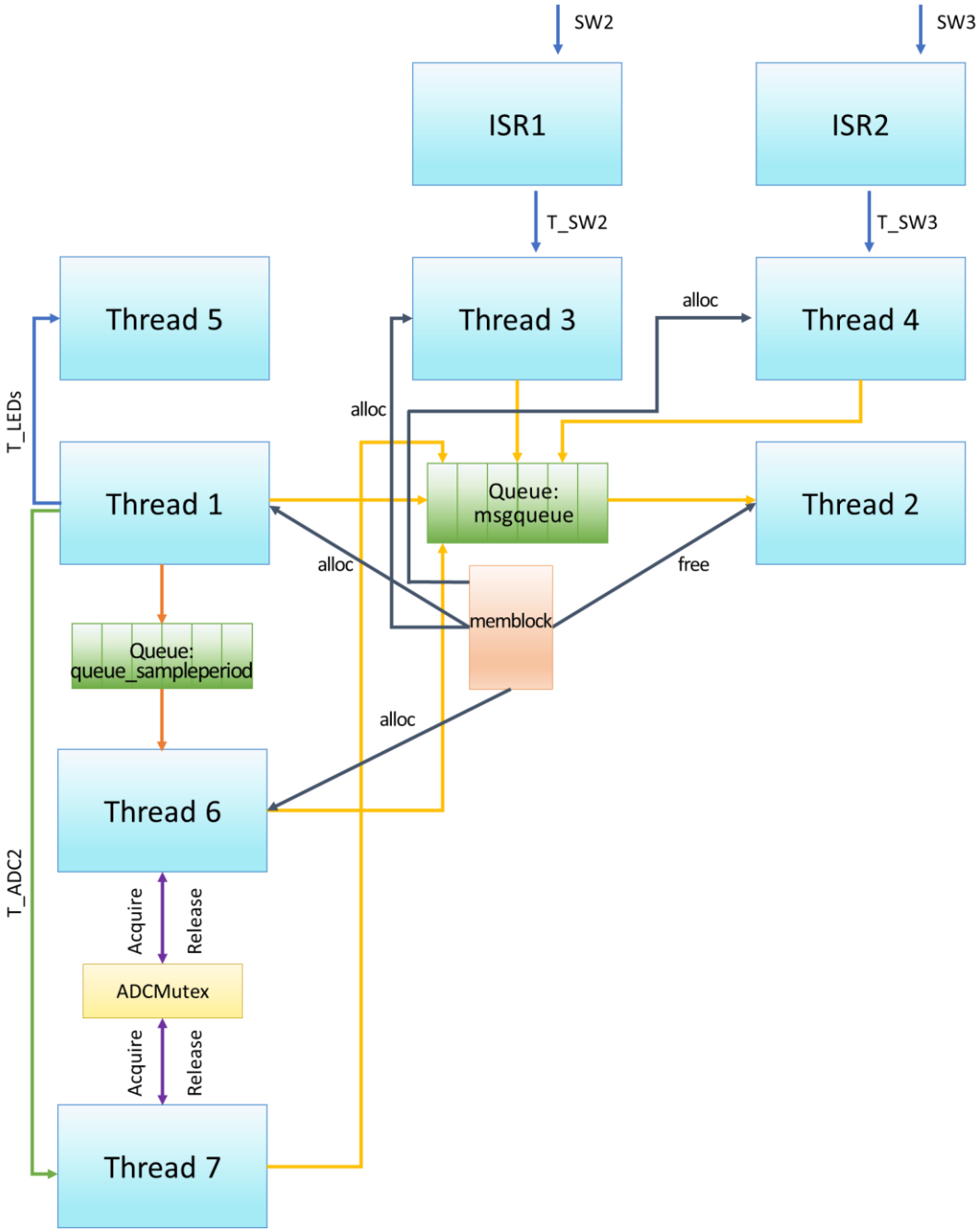

Figure 1 Diagram of inter thread communications

3. Flowcharts of each thread/ module

Figure 2 represents flow chart of printstr function. It starts by allocating a memory block from the operating system's memory pool, waiting indefinitely for space to become available if necessary. It then creates a message structure to hold the data to be printed. Next, it copies the string data passed as the status parameter to the canData field of the message structure using a string copy function like strcpy. Finally, it posts the pointer to the memory block containing the message structure to the message queue, again waiting indefinitely until space is available. Overall, the function efficiently prepares and queues up string data for printing, ensuring seamless communication within the embedded system.

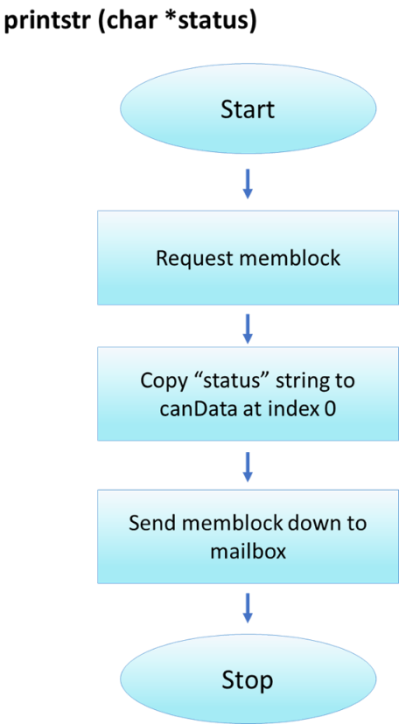

Figure 2 Flowchart of printstr function

The flowchart for the printchar function (Figure3) begins by initializing a character array named str with a size of 2, designed to hold the character to be printed along with a null terminator. The input character c is then assigned to the first element of the str array, while the null terminator '\0' is added to the second element to denote the end of the string. Following this setup, the function proceeds to call the printstr function, passing str as its argument to facilitate the printing of the character.

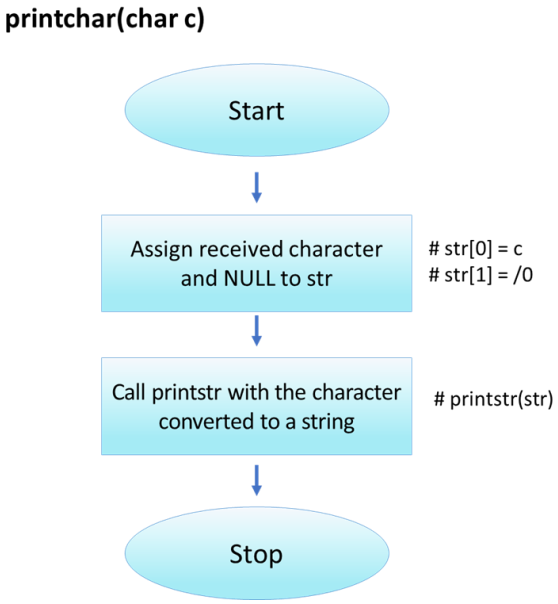

Figure 3 Flowchart of printchar function

The flowchart for the readnumber function (Figure4) illustrates the process of reading a numerical input from the user via a serial interface. Initially, a variable number of type uint32_t is initialized to 0 to store the numerical value. The function then reads a character c from the input using the GETCHAR function and prints it to the screen using the printchar function. If the character read is a space, indicating the start of the numerical input, the function enters a loop. Within this loop, it continues to read characters until a non-digit character or a backspace (ASCII code 127) is encountered.

If a backspace is encountered, the function adjusts the number variable accordingly by dividing it by 10 to remove the last digit. Otherwise, it multiplies the current value of number by 10 and adds the numerical value of the character (obtained by subtracting the ASCII value of '0') to it. The loop continues until a non-digit character is encountered. Finally, the function returns the accumulated numerical value stored in the number variable. This function is designed to allow users to modify numerical values by using the backspace key.

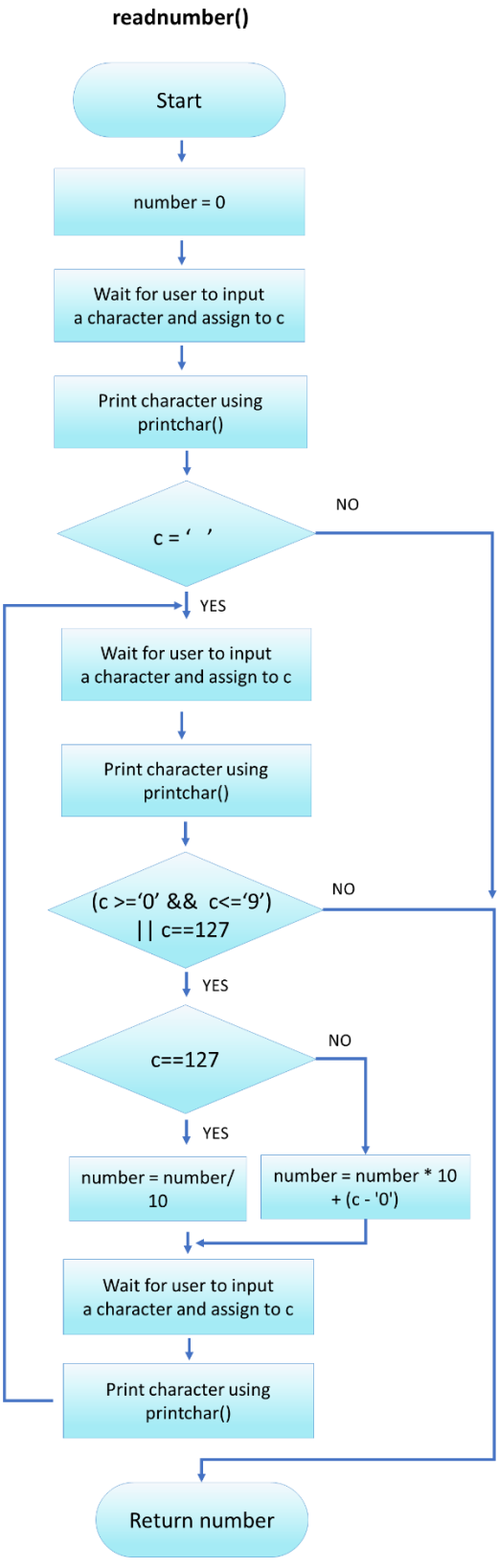

Figure 4 Flowchart of readnumber function

The flowchart for the producer_thread (Figure 5) clarify the operation of a thread responsible for producing input data and controlling flags for LED operation and ADC sampling. Initially, the thread enters an infinite loop, continuously awaiting user input. Upon receiving a character ‘c’ from the user via the GETCHAR function, the thread prints the character to the PC using the printchar function. If the character is ‘1’, indicating a request to change the sample period, the thread calls the readnumber function to read and store the new sample period. Following this, the thread checks if the sample period is greater than 0, and if so, it puts the sample period into the queue_sampleperiod message queue passing to thread 6.

Additionally, the thread checks the value of ‘c’ and sets corresponding flags for LED control and ADC sampling. If the user input ‘R’, it sets flag[0] of the LED switcher thread to 1 (T_LEDs = 0x01). If the user input ‘r’, it sets flag[1] of the LED switcher thread to 1 (T_LEDs = 0x02). If the user input ‘G’, it sets flag[2] of the LED switcher thread to 1 (T_LEDs = 0x04). If the user input ‘g’, it sets flag[3] of the LED switcher thread to 1 (T_LEDs = 0x08). If the user input ‘B’, it sets flag[4] of the LED switcher thread to 1 (T_LEDs = 0x10). If the user input ‘b’, it sets flag[5] of the LED switcher thread to 1 (T_LEDs = 0x20). If the user input ‘2’, it sets flag[0] of the ADC2 thread to 1 (T_ADC2 = 0x01). Then, it returns to wait for the user to input the character again.

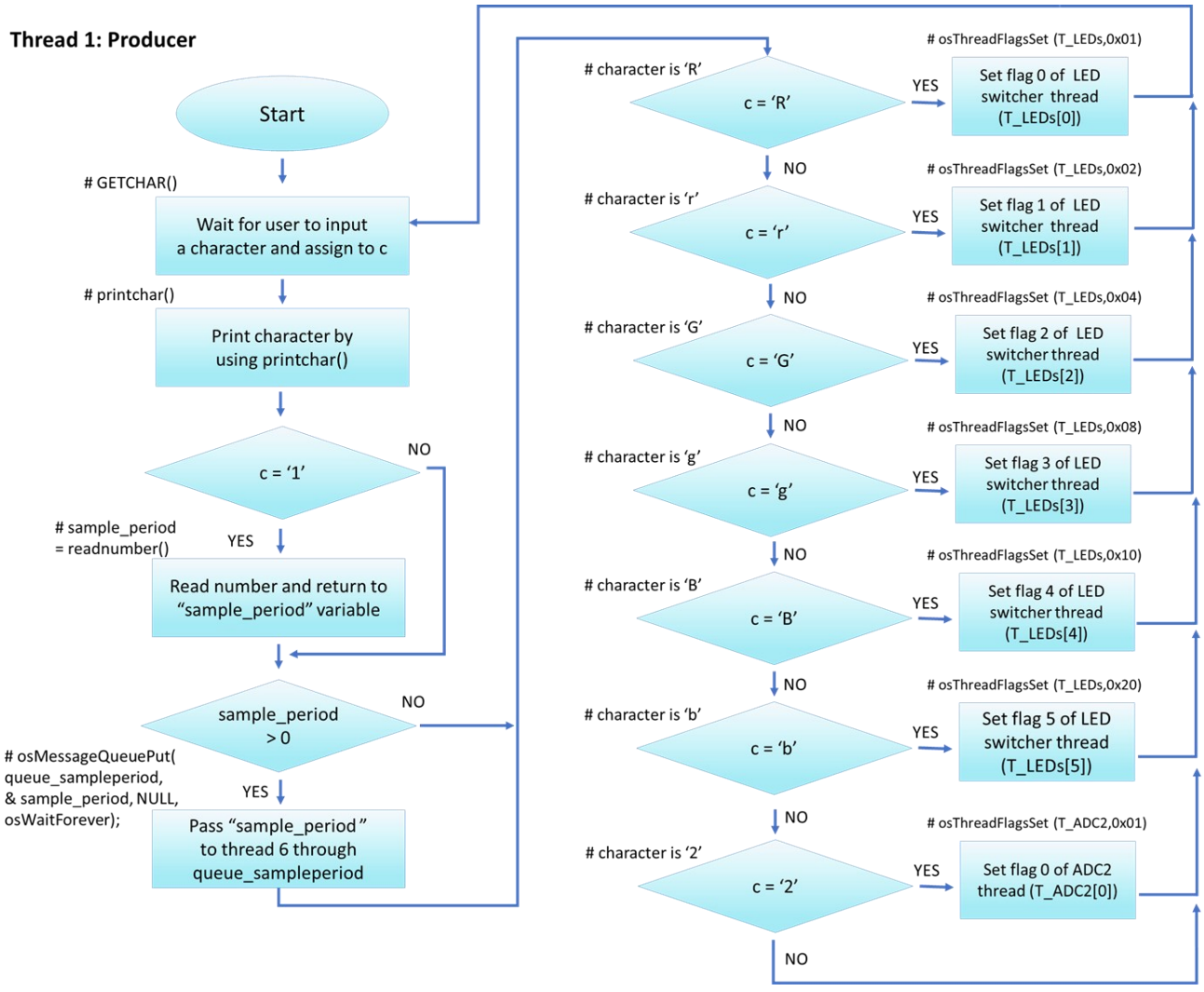

Figure 5 Flowchart of thread 1

The flowchart for the consumer_thread (Figure 6) depicts a thread continuously waiting for messages in a queue. Upon receiving a message, it prints the message to the screen and then frees the associated memory block. The thread then resumes waiting for the next message in the queue, ensuring a continuous process of message consumption and display.

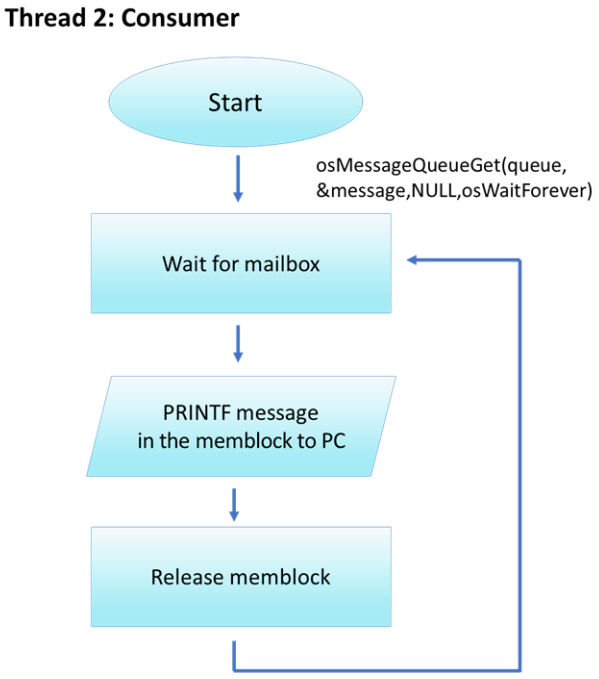

Figure 6 Flowchart of thread 2

The flowchart for the ISR1 (Figure 7) outlines the process of handling an interrupt triggered by the SW2 pin on the board. Initially, the function clears the external interrupt flag associated with SW2. Subsequently, it reads the current state of the SW2 pin and stores the value in the sw2_value variable. Depending on the value of sw2_value (0 or 1), the function sets either flag0 or flag1 of the T_SW2 thread.

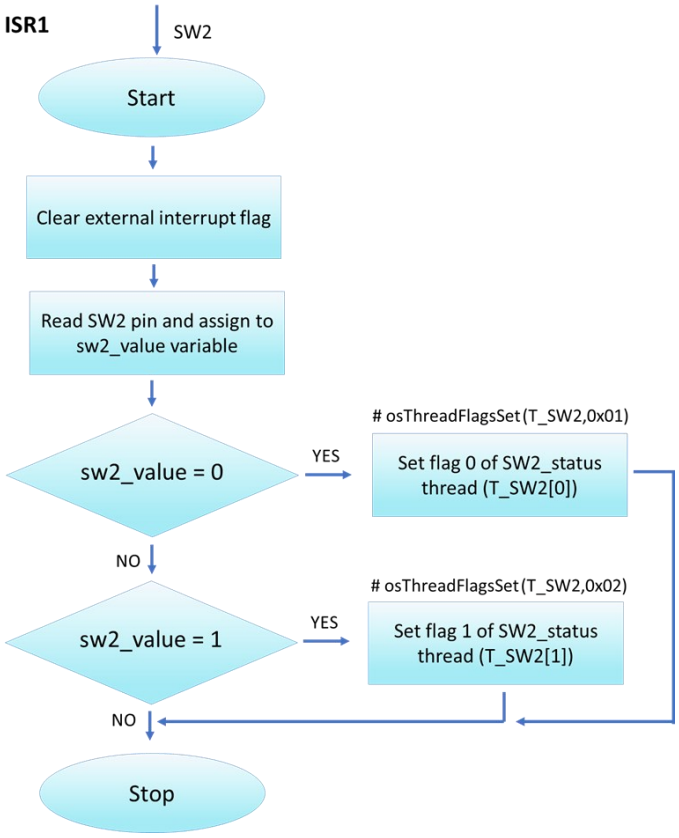

Figure 7 Flowchart of ISR1

The flowchart for the ISR2 (Figure 8) outlines the process of handling an interrupt triggered by the SW3 pin on the board. Initially, the function clears the external interrupt flag associated with SW3. Subsequently, it reads the current state of the SW3 pin and stores the value in the sw3_value variable. Depending on the value of sw3_value (0 or 1), the function sets either flag0 or flag1 of the T_SW3 thread.

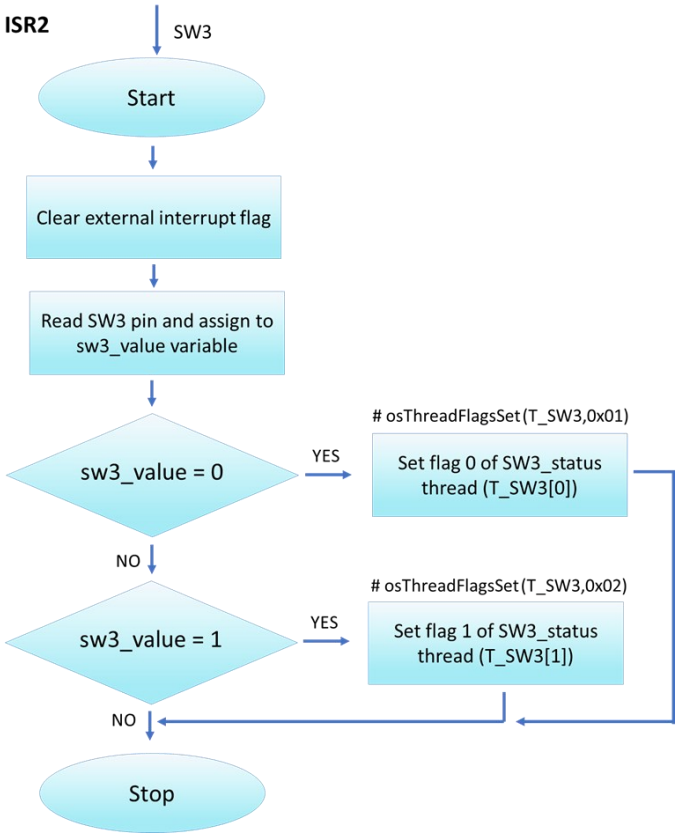

Figure 8 Flowchart of ISR2

The flowchart for the SW2_status thread (Figure 9) depicts its continuous monitoring of the SW2 switch status. Within a loop, the thread waits for a flag[0] or flag[1] indicating either SW2 pressed or released. Upon flag reception, it selects the corresponding status string and prints it to the output device using the printstr function.

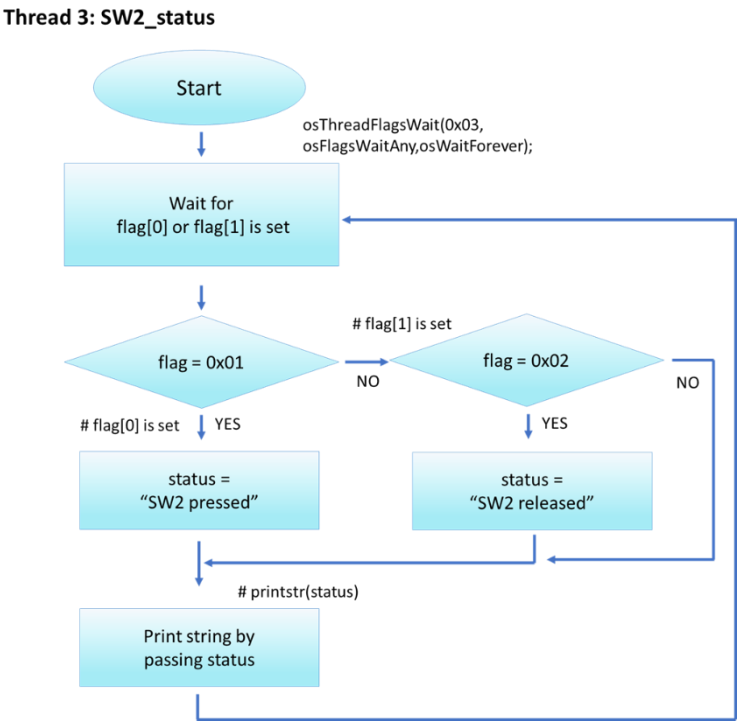

Figure 9 Flowchart of thread 3

The flowchart for the SW3_status thread (Figure 10) depicts its continuous monitoring of the SW3 switch status. Within a loop, the thread waits for a flag[0] or flag[1] indicating either SW3 pressed or released. Upon flag reception, it selects the corresponding status string and prints it to the output device using the printstr function.

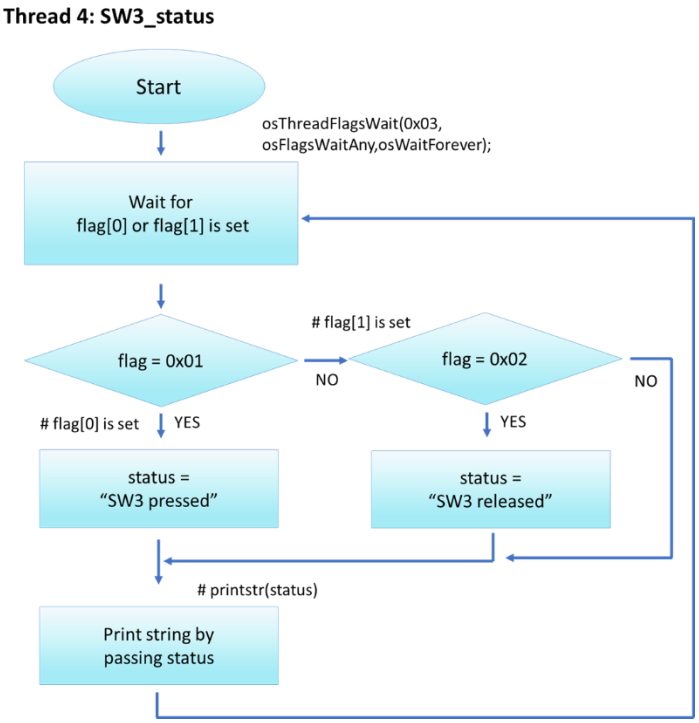

Figure 10 Flowchart of thread 4

The flowchart for the LEDs_switcher thread (Figure 11) outlines its operation in toggling LEDs based on flags received. Within an infinite loop, the thread waits for a flag, indefinitely waiting for any of the specified flags to be set. Upon flag reception, the thread checks the value of the flag and performs the corresponding actions. If the flag indicates turning on an LED (0x01, 0x04, or 0x10), the respective LED (Red, Green, or Blue) is turned on. If the flag indicates turning off an LED (0x02, 0x08, or 0x20), the respective LED is turned off.

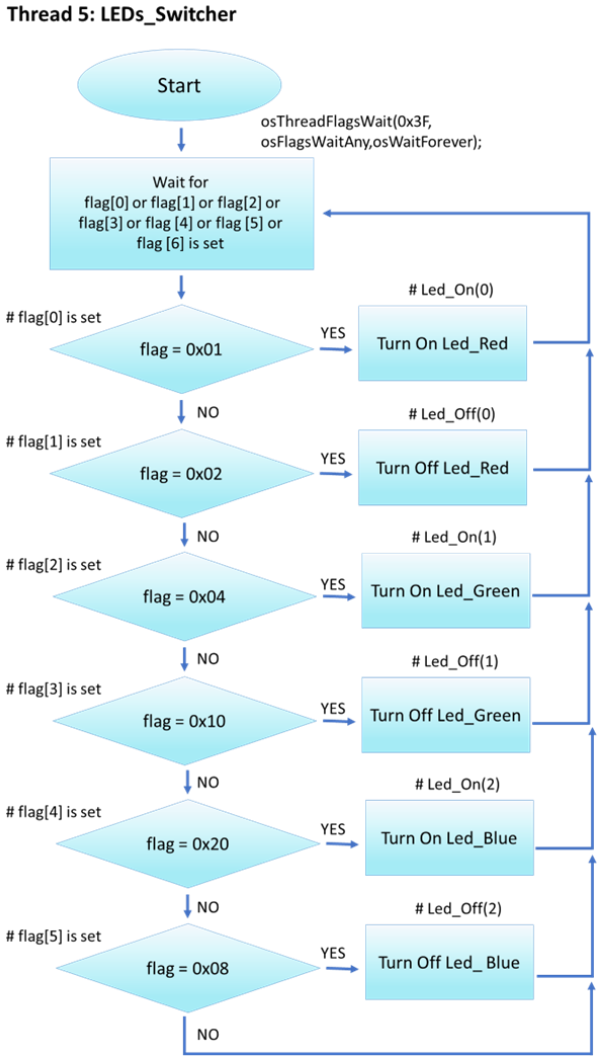

Figure 11 Flowchart of thread 5

The flowchart for the ADC1 thread (Figure 12 (left)) outlines its operation in managing ADC sampling intervals. Initially, the thread creates and initializes a timer and performs the ADC1_callback function periodically. The timer is set with a default interval of 10,000 milliseconds (10 seconds). Subsequently, the thread enters an infinite loop where it waits for messages containing new sampling periods from the queue_sampleperiod message queue. Upon receiving a new period, the thread stops the current timer and starts a new timer with the updated period. This flowchart represents the steps to adjust the ADC sampling interval based on user input within the ADC1_Thread6 thread.

The flowchart for the ADC1_callback function (Figure 12 (right)) details its process. Initially, it configures ADC channel settings and acquires a mutex (ADCMutex) to ensure exclusive ADC access. The function then initiates and waits for ADC conversion to complete. Afterward, it retrieves the ADC value, releases the mutex, and converts the value to voltage. Finally, the voltage is formatted into a string and printed using printstr. This flowchart offers a concise overview of ADC conversion and result printing within the ADC1_callback function.

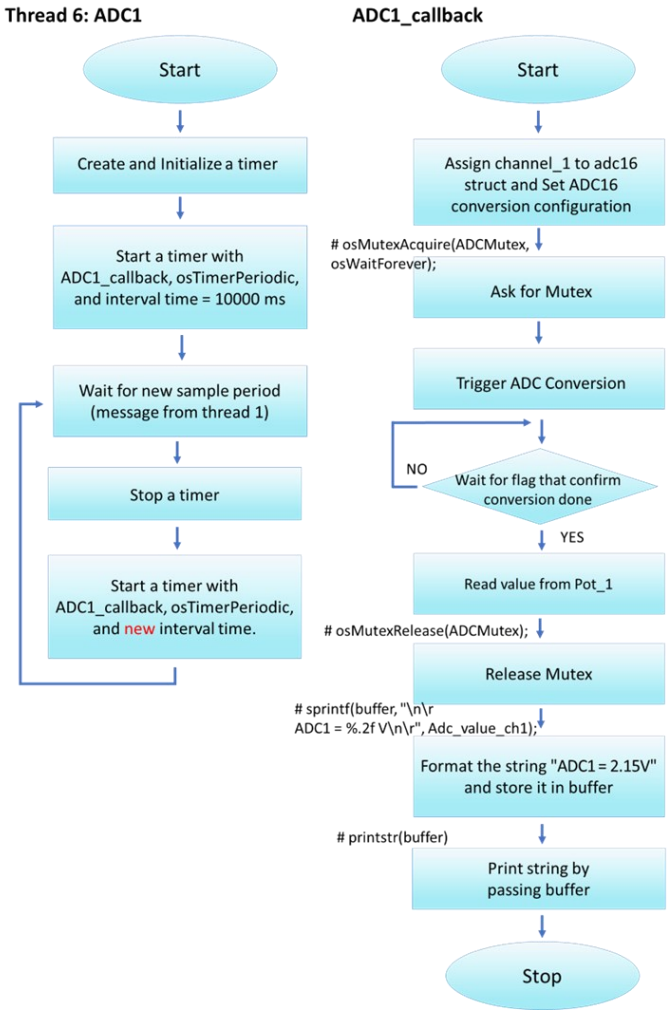

Figure 12 Flowchart of thread 6 (left) and ADC1_callback function (right)

The flowchart for ADC2_Thread7 (Figure 13) simplifies its operation in managing ADC conversions for the second channel. The thread begins with an infinite loop, where it waits for a flag indicating the need for ADC conversion. Upon flag reception, the thread configures the ADC channel settings for channel 2 and requests the mutex (ADCMutex). Then, it initiates the ADC conversion followed by waiting for the conversion to complete.

Once the conversion is finished, the thread retrieves the ADC value and releases ADCMutex to allow other threads to access the ADC. The ADC value is then converted to voltage and formatted into a string. Finally, the formatted string containing the ADC value is printed using the printstr function. This flowchart provides a representation of the steps involved in reading Pot 2 within the ADC2_Thread7 thread.

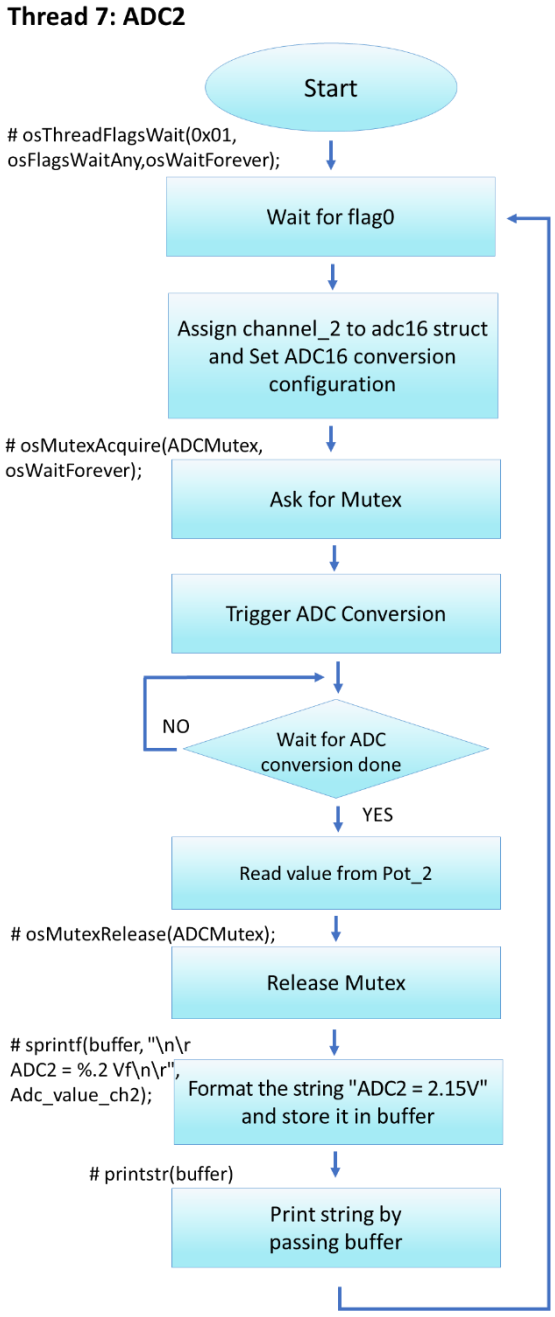

Figure 13 Flowchart of thread 7

4. Description of each module

printstr function

The printstr function takes a character pointer status as its argument. Within the function, a pointer to a message_t structure named message is declared. Memory is then allocated for message using the osMemoryPoolAlloc, which requires two input parameters: the memory pool ID and the timeout. In this code snippet, the memory pool used for allocation is memblock, and it waits indefinitely until memory becomes available. Following memory allocation, the content of the status string is copied to the canData member of the message structure using strcpy function.

Subsequently, a message is put into a queue by posting a pointer to the memory block using osMessageQueuePut. This function requires the message queue ID, a pointer to the buffer for the message to be retrieved from the queue, a pointer to the buffer for the message priority, and the timeout. In this snippet, the pointer to the message structure is placed into a message queue named “queue”, with the message priority set to NULL, indicating default priority, and it waits indefinitely until space becomes available in the queue, as referenced by the timeout.

Overall, this function is designed to copy a string (status) to a canData field within a message structure and then post a pointer to this structure into a message queue, likely for subsequent processing and printing on PC terminal in thread 2.

```
void printstr (char *status)
{
        message_t *message;

        //Allocate a memory block
        message = (message_t*)osMemoryPoolAlloc
        (memblock,osWaitForever);

        //Copy status to canData at index 0
        strcpy((char*)message->canData,(status));

        //Post pointer to memory block
        osMessageQueuePut(queue, &message,NULL
        , osWaitForever);

}
```

printchar function

The function printchar accepts a single input parameter, c, representing the character intended for output. Within the function, c is utilized to generate a string containing solely the character c. This process involves initializing a character array named str with a size of 2, accommodating the character itself and a null terminator. Subsequently, c is assigned to the first element of str, while the null terminator '\0' is appended to the second element to denote the termination of the string. Following this setup, the function invokes another function, printstr, passing str as its argument. This function avoids code duplication of requesting the memblock and sending to the mailbox by calling the printstr function.

```
void printchar(char c) {
    char str[2]; // String to hold the character
    and null terminator
    str[0] = c;
    str[1] = '\0'; // Null terminator

    printstr(str); // Call printstr with the character
converted to a string
}
```

readnumber function

The function readnumber reads a numerical input from the user via a serial interface. Initially, it initializes a variable number to store the numerical value. It reads a character c from the input using the GETCHAR function and prints it to the output device using the printchar function. If the character is a space, indicating the start of the numerical input, the function enters a loop. Within this loop, it continues to read characters until a non-digit character or a backspace (ASCII code 127) is encountered. If a backspace is encountered, the function adjusts the number variable accordingly by dividing it by 10 to remove the last digit. Otherwise, it multiplies the current value of number by 10 and adds the numerical value of the character to it. The loop continues until a non-digit character is encountered, at which point the function returns the accumulated numerical value stored in the number variable.

```
uint32_t readnumber(void){
    uint32_t number = 0;
    char c;
    c = GETCHAR();
    printchar(c);
    if(c == ' ')
        {
            c = GETCHAR();
            printchar(c);
            while ((c >= '0' && c <= '9')|| c == 127)
            {
                if(c==127)
                    number = number / 10 ;
                else
                    number = number * 10 + (c - '0');
                c = GETCHAR();
                printchar(c);
            }
        }
            return number;
}
```

5. Description of each thread

5.1 Thread 1: Producer

The provided code defines a thread function named producer_thread, responsible for taking user input, allocating memory blocks for messages, and posting these messages to a message queue. Within an infinite loop, the thread continuously waits for user input using the GETCHAR() function. Once a character is input by the user, the thread calls printchar function to print the character. If the input is '1', it calls readnumber function to get number and assigns it to sample_period variable. If the sample_period is more than 0, it sends the sample_period to thread 6 through queue_sample_period queue without any memory block.

Additionally, based on the input character, the thread sets flags using the osThreadFlagsSet function to signal other threads about specific conditions. If the input character is 'R', 'r', 'G', 'g', 'B', 'b', or '2', corresponding flags are set to inform other threads. Specifically, if 'R' is input, flag 0 is set to thread T_LEDs using osThreadFlagsSet. For 'r', flag 1 is set to thread T_LEDs. Similarly, for 'G', 'g', 'B', and 'b', flags 2, 3, 4, and 5 are respectively set to thread T_LEDs. Moreover, if '2' is input, flag 0 is set to thread T_ADC2. These flags enable other threads to communicate and respond to the input conditions appropriately, facilitating coordinated behavior across the system.

In the app_main, the osThreadNew function is used to start a thread and set it to the READY state. More details can be found in Section 5.11.

```
static const osThreadAttr_t ThreadAttr_producer =
{.name = "producer",};
__NO_RETURN void producer_thread (void  *args) {

//Define pointer for memory block
message_t *message;
char c;
int sample_period;
    while (1)
        {
            //Wait for the user to input a character
            c = GETCHAR();
            printchar(c);
            if (c == '1'){
                sample_period = readnumber();
            }

            /* Check that time change more than 0*/
            if(sample_period > 0)

            osMessageQueuePut(queue_sampleperiod,
            &sample_period, NULL, osWaitForever);

            /* Check condition of c and set flags. */
            if (c == 'R')
                    osThreadFlagsSet (T_LEDs,0x01);

            else if (c == 'r')
                    osThreadFlagsSet (T_LEDs,0x02);

            else if (c == 'G')
                    osThreadFlagsSet (T_LEDs,0x04);

            else if (c == 'g')
                    osThreadFlagsSet (T_LEDs,0x08);

            else if (c == 'B')
                    osThreadFlagsSet (T_LEDs,0x10);

            else if (c == 'b')
                    osThreadFlagsSet (T_LEDs,0x20);

            else if (c == '2')
                    osThreadFlagsSet (T_ADC2,0x01);

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.

}
```

5.2 Thread 2: Consumer

The provided code defines a thread function named consumer_thread, responsible for retrieving messages from a message queue and printing them to a PC. Within an infinite loop, the thread remains blocked until it receives a message, using the osMessageQueueGet function. This function retrieves a message from the specified message queue and blocks the thread until a message becomes available, indicated by the osWaitForever timeout parameter. Upon receiving a message, the content is printed to the PC using the PRINTF function. After printing, the memory allocated for the message is released using the osMemoryPoolFree function to ensure proper memory management. This loop operates repeatedly, ensuring that the thread remains active and responsive to incoming messages indefinitely.

```
static const osThreadAttr_t ThreadAttr_consumer =
{.name = "consumer",};
__NO_RETURN void consumer_thread (void  *args){

message_t *message;

    while (1)
    {
        osMessageQueueGet(queue,&message,
        NULL,osWaitForever);

        //PRINTF message to PC
        PRINTF("%s\n\r",message);

    //Release the memblock
        osMemoryPoolFree(memblock,
        message);

    }
}

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.
    // Add the new thread here

}
```

5.3 ISR1: BOARD_SW2_IRQ_HANDLER

The code is an ISR for SW2 button interrupts. Upon receiving an interrupt, it clears the flag and reads the button's current state. Depending on whether the button is pressed or released, the ISR sets a matching flag for a thread: flag 0 for pressed, flag 1 for released.

GPIO_PortClearInterruptFlags clears interrupt flags for a GPIO port. It requires the GPIO base address (base) and a bitmask (mask) indicating which flags to clear. In the code, it clears the interrupt flag for the SW2 button's GPIO port.

GPIO_PinRead is used to read the current state of a specific GPIO pin. In the provided code, it's used to read the state of the SW2 button. It takes two parameters: a pointer to the GPIO peripheral base address (base) and the pin number (pin) to be read.

osThreadFlagsSet is used to set flags for a specific thread. In the provided code, it's being used to set thread flags presumably for thread T_SW2 based on the state of the SW2 button.

```
void BOARD_SW2_IRQ_HANDLER (void)
{
    /* Clear external interrupt flag. */
    GPIO_PortClearInterruptFlags(BOARD_SW2_
    GPIO, 1U << BOARD_SW2_GPIO_PIN);

    /* Read SW2 pin.*/
    uint32_t sw2_value;

    sw2_value =
    GPIO_PinRead(BOARD_SW2_GPIO,
    BOARD_SW2_GPIO_PIN);

        /* Set either flag0 or flag1. */
        if (sw2_value == 0)
            osThreadFlagsSet  (T_SW2,0x01);
        else if (sw2_value == 1)
            osThreadFlagsSet  (T_SW2,0x02);
}
```

5.4 ISR2: BOARD_SW3_IRQ_HANDLER

The BOARD_SW3_IRQ_HANDLER closely resembles BOARD_SW2_IRQ_HANDLER, differing mainly in the GPIO port and pin connections for each button. It operates similarly, clearing the flag upon interrupt, then reading the button's state. Depending on the state (0 for pressed, 1 for released), the ISR sets the appropriate flag for thread coordination: flag 0 for pressed, flag 1 for released.

Furthermore, the initialization of port and pin configurations for each switch is performed in ELEC422.c and specifically in BOARD_InitPins. Additionally, given that both switches are configured with input pull-up, the GPIO_PinRead function interprets a button press as a state of 0 and a release as a state of 1. Moreover, the utilization of osThreadFlagsSet within this handler serves to set thread flags for thread T_SW3, further facilitating seamless communication and coordinated responses within the system.

```
void BOARD_SW3_IRQ_HANDLER (void)
{
    /* Clear external interrupt flag. */
    GPIO_PortClearInterruptFlags(BOARD_SW3_
    GPIO, 1U << BOARD_SW3_GPIO_PIN);

    /* Read SW3 pin.*/
    uint32_t sw3_value;

    sw3_value =
    GPIO_PinRead(BOARD_SW3_GPIO,
    BOARD_SW3_GPIO_PIN);

        /* Set either flag0 or flag1. */
        if (sw3_value == 0)
            osThreadFlagsSet  (T_SW3,0x01);
        else if (sw3_value == 1)
            osThreadFlagsSet  (T_SW3,0x02);
}
```

5.5 Thread 3: SW2_status

The provided code defines a thread function, SW2_status, responsible for monitoring the status of the SW2 button. This thread indefinitely awaits flag notifications using osThreadFlagsWait, indicating either a button presses or releases.

The osThreadFlagsWait function is utilized to wait for specific flags to be set for the current thread. It takes three parameters: flags, options, and timeout. Flags specify the flags to wait for, and the thread will be blocked it meets the specified option or time. Options determine the wait behavior, specifying whether the thread should wait for all specified flags to be set (osFlagsWaitAll) or any one of them (osFlagsWaitAny). Timeout specifies the maximum time the thread will wait for the flags to be set before returning. If set to osWaitForever, the thread will wait indefinitely until the flags are set. Otherwise, it specifies a timeout value in milliseconds. The parameters 0x03, osFlagsWaitAny, and osWaitForever signify indefinite waiting forever until either thread flag 0 or 1 is set.

If flag is equal to 0x01, the content of str1 ('SW2_pressed') is copied into status using strcpy(). If flag equals 0x02, str2 ('SW2_released') is copied into status. Following this task, the printstr function is called with the status string as an argument. The printstr function is used to print strings.

```
static const osThreadAttr_t ThreadAttr_SW2_IR
= { .name = "SW2",   };

__NO_RETURN void SW2_status (void  *argument)
{
    for (;;)
    {
        char str1[128] = "SW2_pressed";
        char str2[128] = "SW2_released";
        char status[128];

        /* Wait for flag and check flag. */
        uint8_t flag;
        flag = osThreadFlagsWait
        (0x03,osFlagsWaitAny,osWaitForever);

        if (flag == 0x01){
            strcpy(status, str1);
        }
        else if(flag == 0x02) {
            strcpy(status, str2);
        }
        // Call printstr function
        printstr(status);
    }
}

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.
    // Add the new thread here

}
```

5.6 Thread 4: SW3_status

SW3_status is a thread function responsible for monitoring the status of the SW3 button. Similar to SW2_status, this thread indefinitely awaits flag notifications using osThreadFlagsWait, indicating either a button presses or releases. Upon flag notification, it constructs a message reflecting whether SW3 was pressed or released. strcpy is employed to copy the button state into the status string due to its string data type. Then status string will be sent as argument of printstr function which is used to print strings.

```
static const osThreadAttr_t ThreadAttr_SW3_IR
= { .name = "SW3",   };

__NO_RETURN void SW3_status (void  *argument)
{
    for (;;)
    {
        char str1[128] = "SW3_pressed";
        char str2[128] = "SW3_released";
        char status[128];

        /* Wait for flag and check flag. */
        uint8_t flag;
        flag = osThreadFlagsWait
        (0x03,osFlagsWaitAny,osWaitForever);

        if (flag == 0x01){
            strcpy(status, str1);
        }
        else if(flag == 0x02) {
            strcpy(status, str2);
        }
        // Call printstr function
        printstr(status);
    }
}

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.
    // Add the new thread here

}
```

5.7 Thread 5: LEDs_switcher

The provided code establishes a thread function named LEDs_switcher, designed to manage the states of multiple LEDs based on flag notifications. The thread awaits flag notifications using the osThreadFlagsWait function. This function blocks the thread until at least one of the specified flags is set, enabling the thread to efficiently respond to changes in the system's state.

The parameters 0x3F, osFlagsWaitAny, and osWaitForever signify waiting indefinitely until either thread flag 0, 1, 2, 3, 4, or 5 is set. Once a flag is received, the thread examines its value to determine the corresponding LED action. Each flag corresponds to a specific LED and action combination. For instance, if flag 0 is set (flag = 0x01), the thread turns on LED 0 (LED_RED) using the LED_On(0) function. Similarly, if flag 1 is set (flag = 0x02), it turns off LED_RED using LED_Off(0). This logic extends to other flags, controlling the states of LED 1 (LED_GREEN) and LED 2 (LED_BLUE) accordingly. If flag 2 is set (flag = 0x04), LED_GREEN is turned on, while flag 0x08 results in LED_GREEN being turned off. Similarly, flag 0x10 triggers LED_BLUE to turn on, and flag 0x20 prompts LED_BLUE to turn off. The LED_On and LED_Off functions are described in ELEC422.c.

The app_main function initializes a new thread, T_LEDs, using osThreadNew. This function associates the LEDs_switcher function with the thread and specifies default attributes defined by ThreadAttr_LEDswitcher. By creating this thread, the application ensures continuous monitoring and control of the LEDs, providing a responsive and dynamic user experience.

```
static const osThreadAttr_t ThreadAttr_LEDswitcher =
{ .name = "LEDswitcher",          };

__NO_RETURN void LEDs_switcher(void *argument)
{
    for (;;) {
        /* Wait for flag and check flag. */
        uint8_t flag;
        flag =
osThreadFlagsWait(0x3F,osFlagsWaitAny,osWaitForev
er);
        if (flag     == 0x01)
          {
          LED_On(0);
          }
        else if (flag == 0x02)
          {
          LED_Off(0);
          }
        else if (flag ==  0x04)
          {
          LED_On(1);
          }
        else if (flag ==  0x08)
          {
          LED_Off(1);
          }
        else if (flag ==  0x10)
          {
          LED_On(2);
          }
        else if (flag ==  0x20)
          {
          LED_Off(2);
          }
    }
}

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.
    // Add the new thread here

}
```

5.8 Thread 6: ADC1

Initially, a timer with the name "timer_ADC1" is created and initialized with the osTimerNew function. osTimerNew has 4 input parameters; function pointer to callback function, type of timing behaviour, argument to the timer callback function, and timer attributes. In the snippet code, the timer is associated with the ADC1_callback function and set to trigger periodically (osTimerPeriodic). The timer is started with an initial period of 10000 milliseconds (10 seconds) using osTimerStart. The task will continuously do periodic. Inside an infinite loop, the thread waits indefinitely for a message on the queue_sampleperiod message queue using osMessageQueueGet. When a message is received, containing a new period for the timer, the thread stops the timer (osTimerStop) and then restarts it with the new period using osTimerStart.

```
osTimerId_t T_ADC1_periodic;

static const  osTimerAttr_t timerAttr_timerADC1
 = {.name = "timer_ADC1",};
static const osThreadAttr_t ThreadAttr_ADC1
= {.name = "ADC1",};

__NO_RETURN void ADC1_Thread6 (void  *argument)
{
    /*Create and initialize a timer with ADC1_callback
function and periodic timing setting*/
    T_ADC1_periodic =
    osTimerNew(&ADC1_callback, osTimerPeriodic,
    (void *)0, &timerAttr_timerADC1);

    uint32_t interval_ms = 10000;
    osTimerStart(T_ADC1_periodic , interval_ms);


    while(1){
            uint32_t newperiod;
            osMessageQueueGet(queue_sampleperiod,
            &newperiod, NULL, osWaitForever);
            osTimerStop(T_ADC1_periodic);
            osTimerStart(T_ADC1_periodic , newperiod);
    }
}

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.
    // Add the new thread here

}
```

5.9 ADC1_callback

The ADC1_callback function serves as the callback for the ADC conversion process triggered by the timer. Within this function, the ADC channel configuration is set up, specifying the channel number (DEMO_ADC16_USER_CHANNEL) which is Pot 1 and disabling differential conversion. Mutex acquisition ensures exclusive access to the ADC hardware during conversion. The ADC conversion is initiated using ADC16_SetChannelConfig, and the function waits until the conversion is completed. Once the conversion is done, the ADC value is retrieved, scaled to obtain the corresponding voltage, and formatted into a string. This string represents the ADC value for channel 1. The printstr function is then called to print this string.

```
#define DEMO_ADC16_BASE                 ADC0
#define DEMO_ADC16_CHANNEL_GROUP        0U
#define DEMO_ADC16_USER_CHANNEL         12U
/* PTB2, ADC0_SE12 *//* Pot_1*/
#define DEMO_ADC16_USER_CHANNEL_2       13U
/* PTB3, ADC0_SE13 *//* Pot_2*/

void ADC1_callback (void  *argument) {

    //Assign channe1_1 to adc16 struct.
    adc16_channel_config_t adc16ChannelConfigStruct;
    adc16ChannelConfigStruct.channelNumber
    = DEMO_ADC16_USER_CHANNEL;
    adc16ChannelConfigStruct.enableDifferentialConversion
    = false;

    //Ask for Mutex to prevent a collision in ADC
conversion.
    osMutexAcquire(ADCMutex, osWaitForever);

    //ADC Conversion

    ADC16_SetChannelConfig(DEMO_ADC16_BASE,
    DEMO_ADC16_CHANNEL_GROUP,
    &adc16ChannelConfigStruct);

    while (0U == (kADC16_ChannelConversionDoneFlag &
    ADC16_GetChannelStatusFlags(DEMO_ADC16_BASE,
    DEMO_ADC16_CHANNEL_GROUP)))
    {
    }
    int g_Adc16ConversionValue_ch1;
    g_Adc16ConversionValue_ch1 =
    ADC16_GetChannelConversionValue
    (DEMO_ADC16_BASE,
    DEMO_ADC16_CHANNEL_GROUP);

    //Release Mutex after finishing ADC conversion.
    osMutexRelease(ADCMutex);

    float Adc_value_ch1;
    Adc_value_ch1 =
    3.3*g_Adc16ConversionValue_ch1/4096;

    char buffer[128];
    // Format the string "ADC1 = <value>"
    sprintf(buffer, "\n\rADC1 = %.2f V\n\r", Adc_value_ch1);

    // Call printstr function
    printstr(buffer);

}
```

5.10 Thread 7: ADC2

The thread waits for a specific thread flag (0x01) using osThreadFlagsWait, and osWaitForever signifies that it waits until this flag is set without a time limit. This flag serves as a trigger to start the ADC conversion process. Upon receiving the thread flag, the ADC channel configuration is set up. This includes specifying the channel number (DEMO_ADC16_USER_CHANNEL_2), which refers to Pot 2, disabling differential conversion. Before starting ADC conversion, the thread acquires a mutex (ADCMutex) using osMutexAcquire. The osWaitForever parameter in osMutexAcquire ensures that if this mutex is not released, other processes requiring this mutex must wait until the end of the ADC conversion process. This is to prevent concurrent access to the ADC hardware, ensuring that only one thread performs ADC conversion at a time.

The ADC conversion is triggered using ADC16_SetChannelConfig, which configures the ADC module with the specified channel configuration. The thread then waits for the conversion to complete by polling the conversion done flag. After retrieving the value of Pot 1 to the g_Adc16ConversionValue_ch1 variable, the mutex is released using osMutexRelease. This allows other threads to access the ADC hardware. The converted ADC value (g_Adc16ConversionValue_ch2) is scaled to obtain the corresponding voltage value (Adc_value_ch2). This voltage value is then formatted into a string "ADC2 = 2.5 V" using sprintf and stored in the buffer array. Finally, the printstr function is called with the buffer containing the formatted string as an argument. This presumably sends the string for printing. Then, the thread will wait until the thread flag is set again.

```
#define DEMO_ADC16_BASE                 ADC0
#define DEMO_ADC16_CHANNEL_GROUP        0U
#define DEMO_ADC16_USER_CHANNEL         12U
/* PTB2, ADC0_SE12 *//* Pot_1*/
#define DEMO_ADC16_USER_CHANNEL_2       13U
/* PTB3, ADC0_SE13 *//* Pot_2*/

static const osThreadAttr_t ThreadAttr_ADC2
= {.name = "ADC2",};
__NO_RETURN void ADC2_Thread7 (void *argument)
{
    for(;;){
            osThreadFlagsWait(0x01,
            osFlagsWaitAny,osWaitForever);

            //Assign channel_2 to adc16 struct
            adc16_channel_config_t
            adc16ChannelConfigStruct;

            adc16ChannelConfigStruct.channelNumber
            = DEMO_ADC16_USER_CHANNEL_2;

            adc16ChannelConfigStruct.enableDifferentialC
            onversion = false;

            adc16ChannelConfigStruct.enableInterrupt
            OnConversionCompleted = false;

            //Ask for Mutex to prevent a collusion
            in ADC conversion.
            osMutexAcquire(ADCMutex,
            osWaitForever);

            //ADC Conversion
            ADC16_SetChannelConfig(
            DEMO_ADC16_BASE,
            DEMO_ADC16_CHANNEL_GROUP,
            &adc16ChannelConfigStruct);

            //Wait for conversion done
            while (0U ==
            (kADC16_ChannelConversionDoneFlag
            &ADC16_GetChannelStatusFlags
            (DEMO_ADC16_BASE,
            DEMO_ADC16_CHANNEL_GROUP)))
            {
            }

            int g_Adc16ConversionValue_ch2;
            g_Adc16ConversionValue_ch2 =
            ADC16_GetChannelConversionValue
            (DEMO_ADC16_BASE,
            DEMO_ADC16_CHANNEL_GROUP);

            //Release Mutex after finishing
            ADC conversion.
            osMutexRelease(ADCMutex);

            float Adc_value_ch2;
            Adc_value_ch2 =
            3.3*g_Adc16ConversionValue_ch2/4096;

            char buffer[128];

            // Format the string "ADC2 = <value>" and store
            it in buffer
            sprintf(buffer, "ADC2 = %.2f V", Adc_value_ch2);

            // Call printstr function
            printstr(buffer); }
}

void app_main (void *argument) {

    // Create a thread and add it to Active Threads.
    // Add the new thread here

}
```

5.11 app_main

The osMemoryPoolNew function creates and initializes a memory pool with attributes including block_count (maximum number of memory blocks), block_size (memory block size in bytes), and memory pool attributes. It returns the memory pool ID "memblock" for reference by other functions. Likewise, the osMessageQueueNew function initializes a message queue object with parameters including msg_count (maximum number of messages), msg_size (maximum message size in bytes), and message queue attributes. This function returns message queue IDs, msgqueue, and queue_sampleperiod, for reference by other functions. When invoking the printstr function, it utilizes memblock because only the pointer is copied to msgqueue. This zero-copy mailbox offers the advantage of not copying all the data, thereby reducing memory usage. Conversely, when passing an integer like sample_period, it transmits its value through queue_sampleperiod without a memory block.

The osThreadNew function is employed to create a thread, which requires three attributions: the thread function, a pointer passed as a start argument, and thread attributes. In the provided code, the first thread is initialized with the producer_thread function, while the second thread is created with the consumer_thread function. The third and fourth threads are built with the SW2_status and SW3_status functions, respectively, returning thread IDs T_SW2 and T_SW3. The fifth thread is initialized with the LEDs_switcher function, utilizing T_LEDs as the thread ID. Finally, the sixth and seventh threads are created with the ADC1_Thread6 and ADC2_Thread7 functions, respectively, with T_ADC2 serving as the thread ID for the seventh thread. Additionally, a mutex is created and initialized as a mutex object, returning ADCMutex as the mutex ID.

```
    static const osMemoryPoolAttr_t
    memorypoolAttr_memblock =
    {.name = "memory_pool",};

    static const osMemoryPoolAttr_t
    memorypoolAttr_mem_sampleperiod =
    {.name = "memory_sampleperiod",};

    static const osThreadAttr_t ThreadAttr_main =
     {.name = "main",};

    static const osThreadAttr_t ThreadAttr_app_main =
    {   .name = "app_main",   };

    static const osMutexAttr_t MutexAttr_ADCMutex =
    {.name = "ADCMutex",};

void app_main (void *argument) {

    memblock = osMemoryPoolNew(16,
    sizeof(message_t),&memorypoolAttr_memblock );

    msgqueue = osMessageQueueNew(16, 4, NULL);

    queue_sampleperiod = osMessageQueueNew(16, 4,
    NULL);

    osThreadNew(producer_thread, NULL,
    &ThreadAttr_producer);        //Thread1: Producer

    osThreadNew(consumer_thread, NULL,
    &ThreadAttr_consumer);        //Thread2: Consumer

    T_SW2 = osThreadNew(SW2_status, NULL,
    &ThreadAttr_SW2_IR);          //Thread3: SW2_status

    T_SW3 = osThreadNew(SW3_status, NULL,
    &ThreadAttr_SW3_IR);          //Thread4: SW3_status

    T_LEDs = osThreadNew(LEDs_switcher, NULL,
    &ThreadAttr_LEDswitcher); //Thread5: LEDs_switcher

    ADCMutex = osMutexNew(&MutexAttr_ADCMutex );
    //Mutex is used to prevent ADC conversion collision

    osThreadNew(ADC1_Thread6, NULL,
    &ThreadAttr_ADC1);            //Thread6: ADC1

    T_ADC2 = osThreadNew(ADC2_Thread7, NULL,
    &ThreadAttr_ADC2);            //Thread7: ADC2

        app_mainid = osThreadGetId();

        osThreadTerminate(app_mainid);

}
```

5.12 ELEC422.c

he provided code initializes essential components for an embedded system. It first configures GPIO pins for input switches SW2 and SW3, enabling interrupts on either edge for both switches to detect when they are pressed or released. Next, it configures the ADC module with default settings and disables hardware triggers. Additionally, it performs auto-calibration for the ADC module, printing a message to indicate the success or failure of the calibration process. This ADC configuration ensures accurate ADC conversion, enabling the system to perform precise analog-to-digital conversions. In addition, the code includes the header file "fsl_adc16.h", providing necessary declarations and definitions for the ADC16 module from the Freescale Software Library (FSL). This inclusion ensures that the functions and configurations related to the ADC16 module are correctly recognized and utilized within the code.

```
#include "fsl_adc16.h"
#define DEMO_ADC16_BASE              ADC0

void elec422_startup(void)
{
    /* Init input switch SW2 GPIO. */
    PORT_SetPinInterruptConfig(BOARD_SW2_PORT,
    BOARD_SW2_GPIO_PIN,
    kPORT_InterruptEitherEdge); //Change to EitherEdge
    EnableIRQ(BOARD_SW2_IRQ);
    GPIO_PinInit(BOARD_SW2_GPIO,
    BOARD_SW2_GPIO_PIN, &sw_config_SW2);

   /* Init input switch SW3 GPIO. */
    PORT_SetPinInterruptConfig(BOARD_SW3_PORT,
    BOARD_SW3_GPIO_PIN,
    kPORT_InterruptEitherEdge); //Change to EitherEdge
    EnableIRQ(BOARD_SW3_IRQ);
    GPIO_PinInit(BOARD_SW3_GPIO,
    BOARD_SW3_GPIO_PIN, &sw_config_SW3);

    /* ADC Config */
    adc16_config_t adc16ConfigStruct;
    ADC16_GetDefaultConfig(&adc16ConfigStruct);

    /* Initializes the ADC16 module. */
    ADC16_Init(DEMO_ADC16_BASE,
    &adc16ConfigStruct);
    ADC16_EnableHardwareTrigger
    (DEMO_ADC16_BASE, false);

    /* Calibration */
    if (kStatus_Success ==
    ADC16_DoAutoCalibration(DEMO_ADC16_BASE))
   {
      PRINTF("ADC16_DoAutoCalibration() Done.\r\n");
   }
   else
   {
      PRINTF("ADC16_DoAutoCalibration() Failed.\r\n");
   }

}
```

5. Testing schemes

- readnumber function

The provided code snippet employs the sprintf function to format messages and utilizes the printstr function. Initially, in the loop, it displays the ASCII value of each digit that the user types. This approach aids in displaying the ASCII value because backspace has no symbol to represent it, making it easier to spot. When the user types the ending character, the code prints the number extracted from the user's input.

```
uint32_t readnumber(void){
    uint32_t number = 0;
    char c, buffer[128];
    c = GETCHAR();
    printchar(c);
    if(c == ' ')
        {
            c = GETCHAR();
            printchar(c);
            while ((c >= '0' && c <= '9')|| c == 127)
            {

                sprintf(buffer, "\n\rASCII = %d\n\r", c);
                printstr(buffer);

                if(c==127)
                    number = number / 10 ;
                else
                    number = number * 10 + (c - '0');
                c = GETCHAR();
                printchar(c);
            }
            sprintf(buffer, "\n\rNumber = %d\n\r", number);
            printstr(buffer);
        }
            return number;
}
```

Figure 14 illustrates a screenshot of the readnumber function testing scheme, where the user inputs '1' followed by spacebar, then types '1023', and uses backspace to change the value to '1024', and finally 'enter'. The resulting number is '1024', which is correct as intended by the user. Additionally, the ADC1 value is displayed at intervals of 1024 ms, as this number is passed to Thread 6. The code shown in red within the code snippet is added to demonstrate that the readnumber function functions correctly with backspace and is user-friendly.

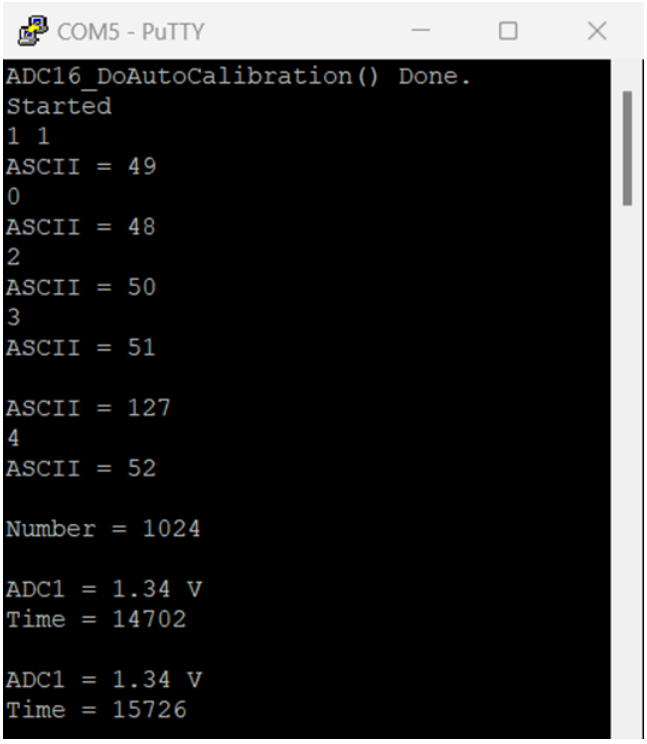

Figure 14 Screenshot of readnumber function testing

- Thread 1

Figure 15 displays the system and thread viewer for Thread 1, indicating that the thread is currently executing a GETCHAR() function, which is a blocking operation. Despite this blocking call, the thread is marked as being in the osThreadRunning state, signifying that it is actively running. Additionally, the flag value is reported as 0x00, indicating that no flags are currently being waited for in this thread.

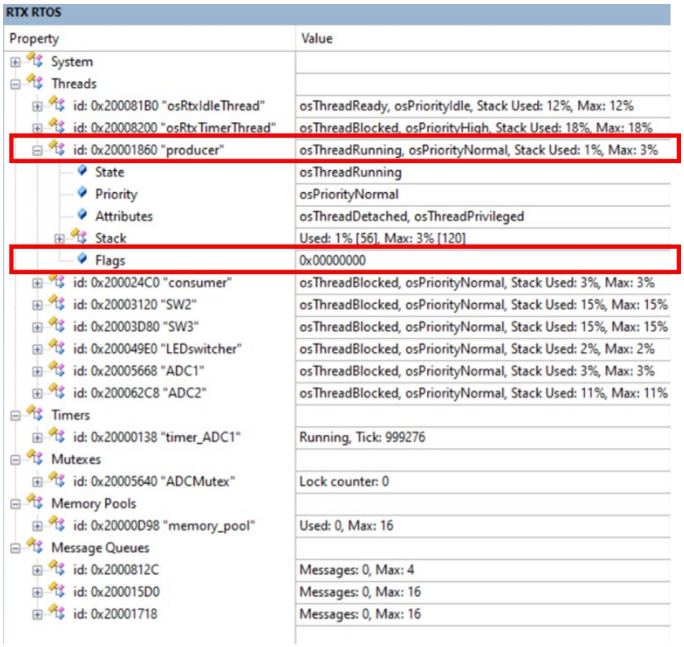

Figure 15 Screenshot of RTX RTOS (Thread 1)

- Thread 2

Figure 16 illustrates the system and thread viewer for Thread 2, where the thread is depicted as waiting for a message get operation from the message queue (msgqueue). This operation is blocking, as indicated by the osThreadBlocked state, suggesting that the thread is currently inactive until a message is received. The timeout for this operation is set to osWaitForever, implying that the thread will wait indefinitely until it receives a message. Additionally, the flag value is reported as 0x00, indicating that no flags are currently being waited for in this thread.

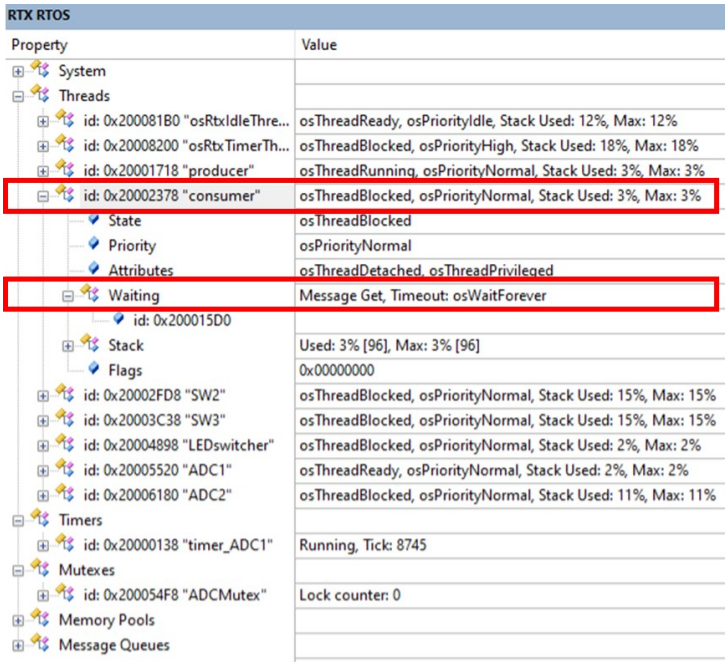

Figure 16 Screenshot of RTX RTOS (Thread 2)

- Thread 3 and Thread 4

Figure 17 depicts the system and thread viewer for Thread 3 and Thread 4, where both threads are observed to be executing an osThreadFlagsWait function, which represents a blocking operation. Despite this blocking call, both threads are marked as being in the osThreadBlocked state, suggesting that they are currently inactive. The timeout for this operation is set to osWaitForever, indicating that the threads will wait indefinitely until they receive a flag. Additionally, it is indicated that the threads are waiting for thread flags, with the flag value reported as 0x03 and osFlagsWaitAny, indicating that they are waiting for either flag[0] or flag[1] to be set.

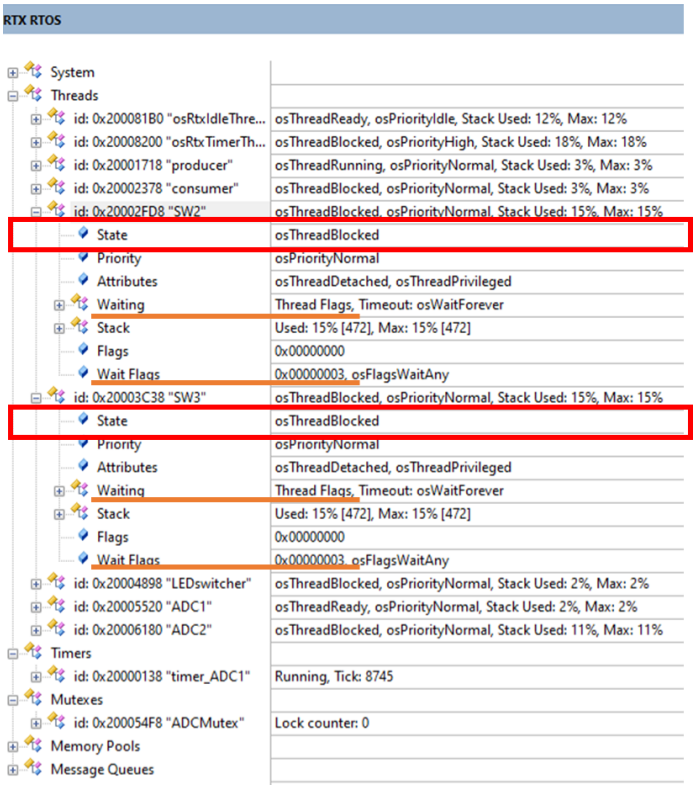

Figure 17 Screenshot of RTX RTOS
(Thread 3 and Thread 4)

Figure 18 displays a Putty screenshot demonstrating the behavior when SW2 and SW3 are pressed and released. When SW2 is pressed, the string "SW2_pressed" is displayed on the screen, and when SW2 is released, the string "SW2_released" is displayed. Similarly, when SW3 is pressed, the string "SW3_pressed" is shown, followed by "SW3_released" when SW3 is released. These results confirm that the system operates as intended, accurately detecting the press and release events of SW2 and SW3.

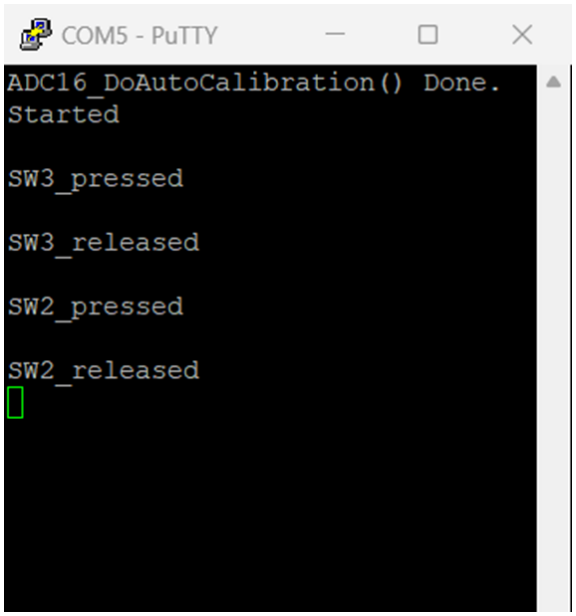

Figure 18 Screenshot of Putty
(Thread 3 and Thread 4)

- Thread 5

Figure 19 presents the system and thread viewer for Thread5, where the thread is executing an osThreadFlagsWait function, indicating a blocking operation. Despite being blocked, it is marked as osThreadBlocked, signifying inactivity. The timeout is set to osWaitForever, implying indefinite waiting for a flag. Additionally, the thread is waiting for thread flags, with flag value reported as 0x3F and osFlagsWaitAny, indicating it awaits any of flag[0] through flag[5] to be set.

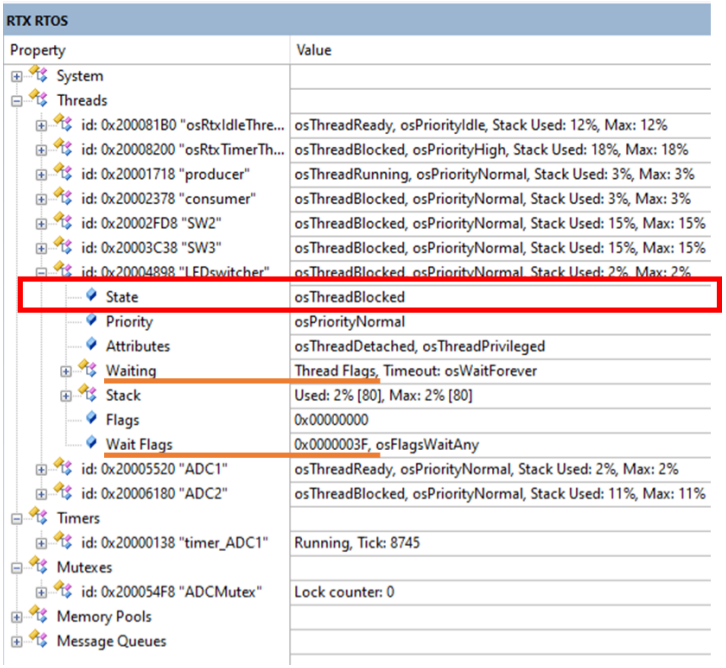

Figure 19 Screenshot of RTX RTOS (Thread 5)

Figure 20 showcases a capture of LEDs on the board alongside the Putty screen. Upon typing 'R', LED_RED turns on; typing 'G' results in LED_GREEN being on, displaying yellow; typing 'B' lights up LED_BLUE, showing white. Subsequently typing 'r' turns off LED_RED, displaying light blue; 'g' turns off LED_GREEN, leaving LED_BLUE displaying blue; 'b' turns off LED_BLUE, leaving no color. Typing 'G' again turns on LED_GREEN, displaying green. This sequence corresponds with the observed colors, indicating correct functionality.

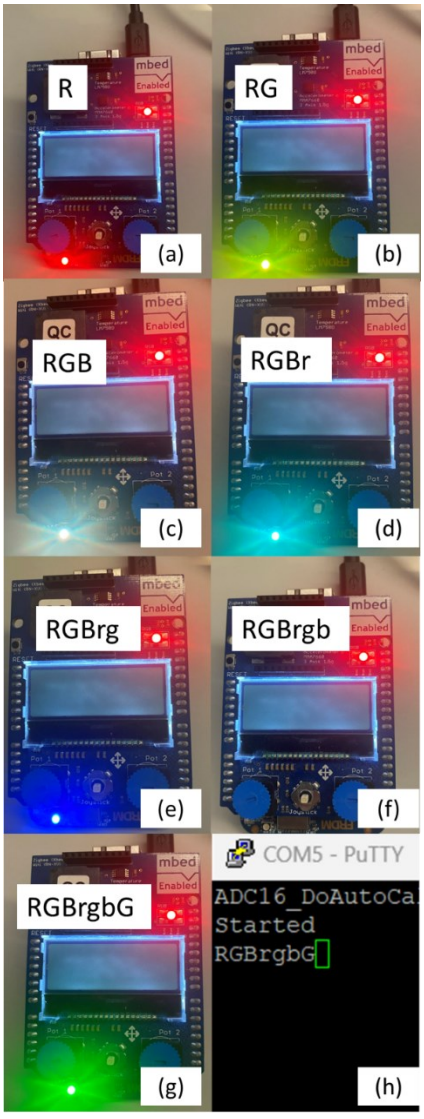

Figure 20 Capture of LED on board and Putty screen

- Thread 6

Figure 21 illustrates the system and thread viewer for Thread 6, where the thread is waiting for a message from the message queue (queue_sampleperiod). This operation is blocking, as indicated by the osThreadBlocked state, implying inactivity until a message is received. The timeout for this operation is set to osWaitForever, implying indefinite waiting until a message arrives. Concurrently, the "timer_ADC1" is running periodically, as denoted by osTimerPeriodic and the callback function, ADC1_callback, with an initial period of 10000 ms (0.1 Hz).

The provided code incorporates the osKernelGetTickCount function, which retrieves the current RTOS kernel tick count (1 tick = 1ms) at the initiation of the ADC_callback function and prints that value using the printstr function. Figure 22 displays a screenshot from Putty, showcasing the ADC1 value and kernel tick count. Before altering the sample period, the ADC1 value is printed at an interval of 10000 ms as the default. Subsequently, when the user inputs '1 20000', the display interval is adjusted to 20000 ms. These results demonstrate the correct operation of the timer periodic and the exchange of data with the message queue (queue_sampleperiod) between Thread 1 and Thread 6.

```
void ADC1_callback (void  *argument) {
    static uint32_t tick = 0U;
    tick = osKernelGetTickCount();

    //Assign channe1_1 to adc16 struct.
    adc16_channel_config_t
adc16ChannelConfigStruct;
    adc16ChannelConfigStruct.channelNumber
    = DEMO_ADC16_USER_CHANNEL;
    adc16ChannelConfigStruct.enableDifferentialConv
ersion = false;

    //Ask for Mutex to prevent a collision in ADC
conversion.
    osMutexAcquire(ADCMutex, osWaitForever);

    //ADC Conversion

    ADC16_SetChannelConfig(DEMO_ADC16_BA
    SE, DEMO_ADC16_CHANNEL_GROUP,
    &adc16ChannelConfigStruct);

    while (0U ==
    (kADC16_ChannelConversionDoneFlag &
    ADC16_GetChannelStatusFlags(DEMO_ADC16_B
    ASE, DEMO_ADC16_CHANNEL_GROUP)))
    {
    }
    int g_Adc16ConversionValue_ch1;
    g_Adc16ConversionValue_ch1 =
    ADC16_GetChannelConversionValue
    (DEMO_ADC16_BASE,
    DEMO_ADC16_CHANNEL_GROUP);

    //Release Mutex after finishing ADC conversion.
    osMutexRelease(ADCMutex);

    float Adc_value_ch1;
    Adc_value_ch1 =
    3.3*g_Adc16ConversionValue_ch1/4096;

    char buffer[128];
    // Format the string "ADC1 = <value>"
    sprintf(buffer, "\n\rADC1 = %.2f V\n\r",
Adc_value_ch1);

    // Call printstr function
    printstr(buffer);

    sprintf(buffer, "Time = %u\n\r", tick);
    printstr(buffer);

}
```

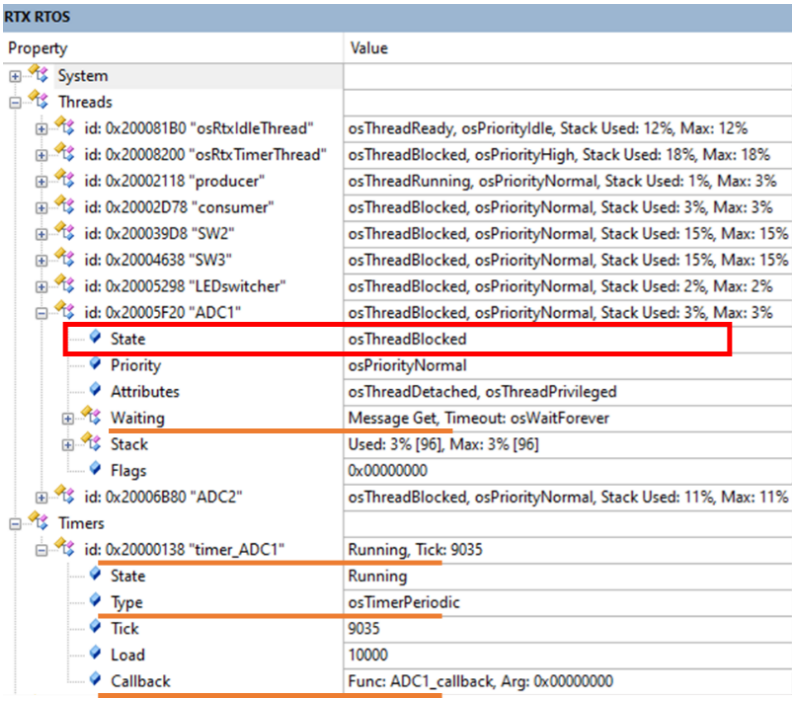

Figure 21 Screenshot of RTX RTOS (Thread 6)

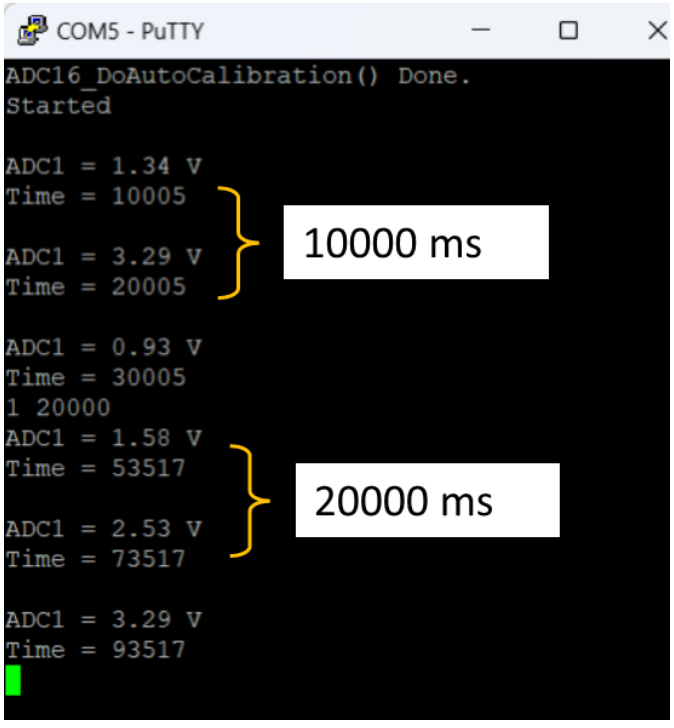

Figure 22 Screenshot of Putty (Thread 6)

- Thread 7

Figure 23 presents the system and thread viewer for Thread 7, where the thread is executing an osThreadFlagsWait function, indicating a blocking operation. Despite being blocked, it is marked as osThreadBlocked, signifying inactivity. The timeout is set to osWaitForever, implying indefinite waiting for a flag. Additionally, the thread is waiting for thread flags, with flag value reported as 0x01 and osFlagsWaitAny, indicating it awaits flag[0] to be set.

Figure 24 illustrates a screenshot captured from Putty, where the ADC value of Pot 2 is displayed on the screen upon user input '2'. This indicates the successful operation of the system in retrieving and displaying the ADC value of Pot 2 in response to user input.

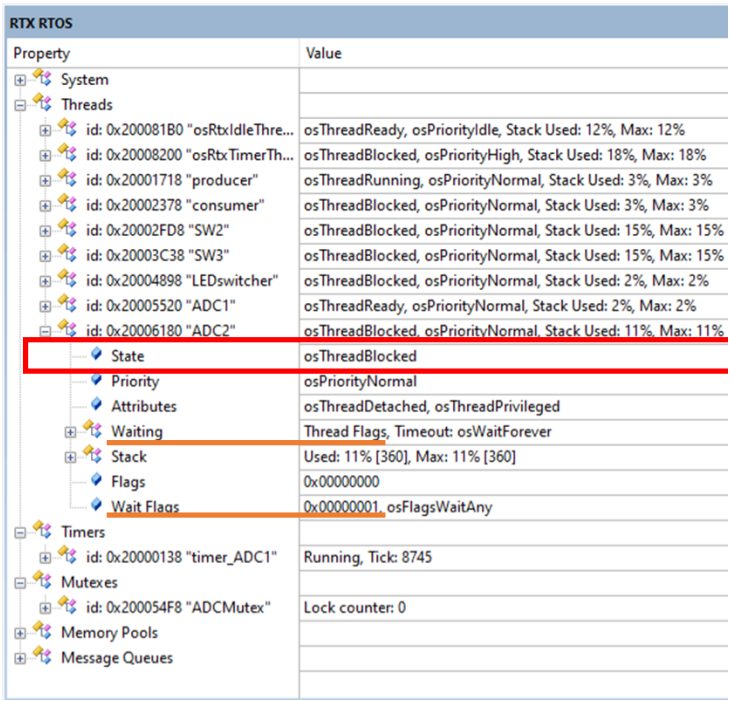

Figure 23 Screenshot of RTX RTOS (Thread 7)

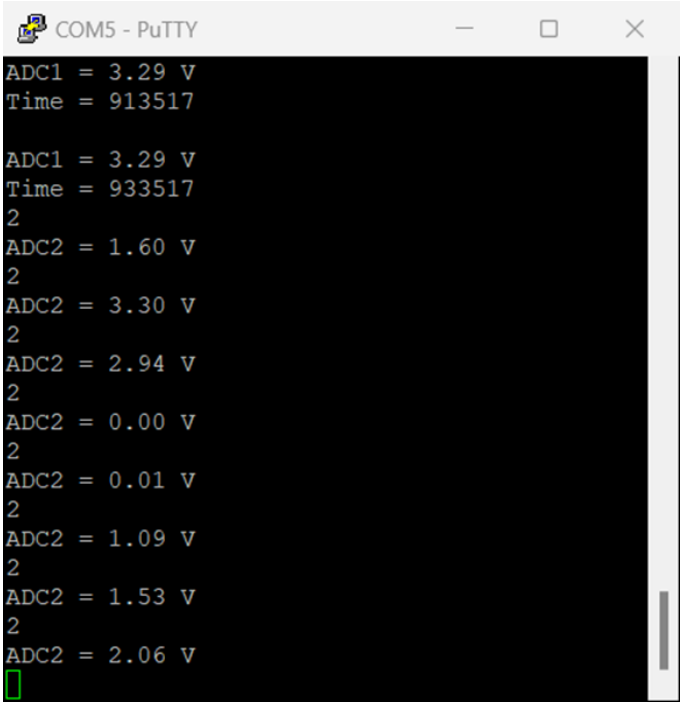

Figure 24 Screenshot of Putty (Thread 7)

- Mutex

Figure 25 displays the system and thread viewer, illustrating the status of the mutex named ADCmutex. When the program reaches the line osMutexAcquire(ADCMutex, osWaitForever) within the ADC1_callback function, the ADCmutex is locked, with the lock counter indicating a value of 0x01. This signifies that the ADCmutex is currently unavailable, ensuring that there will be no collision during ADC conversion.

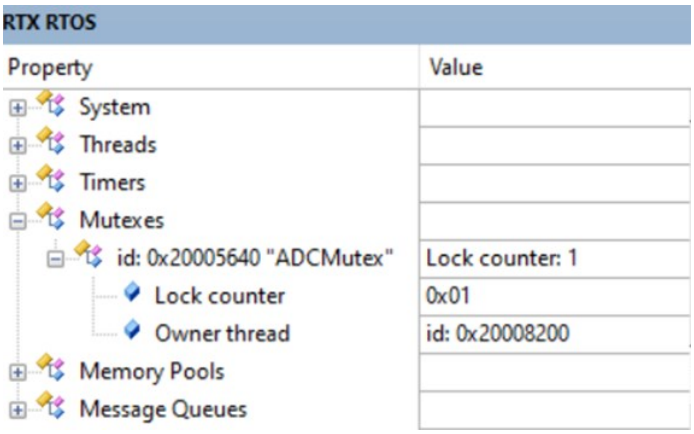

Figure 25 Screenshot of RTX RTOS (Mutex)

- Message Queue

Figure 26 depicts the system and thread viewer, showcasing the status of two message queues. One queue is observed to be waiting indefinitely in the consumer thread, while the other queue is also waiting indefinitely in the ADC1 thread. The first queue is utilized for passing strings or characters, while the second queue is used for passing numbers.

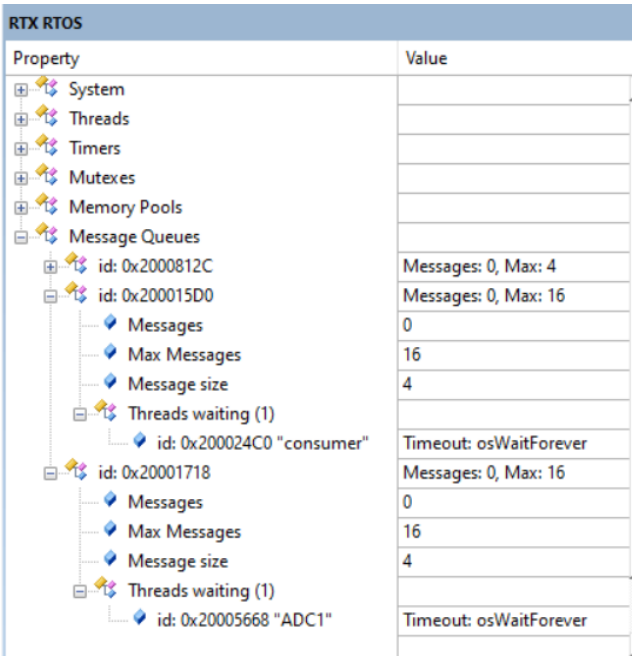

Figure 26 Screenshot of RTX RTOS (Message Queue)

- Overall system

Figure 27 provides a comprehensive overview of the system's operation captured through Putty. Each thread is observed to function correctly, performing its designated tasks without interference from other threads. Thread 1 and Thread 2 effectively handle user input and display messages on the PC terminal, while Thread 3 and Thread 4 successfully monitor switch inputs (SW2 and SW3) and display their states accurately.

Thread 5 demonstrates correct LED control based on user input, while Thread 6 conducts ADC conversions of Pot 1 periodically at an interval of 10000 ms initially. Furthermore, the ability to adjust the sample period for ADC conversions in Thread 6 functions as intended, with the system accurately responding to changes in the sampling rate.

Thread 7 performs ADC conversions of Pot 2 and displays the results correctly when the user inputs '2'. Importantly, each task operates independently within its designated timeframe, as evidenced by the accurate timing of ADC1 conversions despite other tasks occurring in between intervals. Overall, the system exhibits robust functionality across all threads, showcasing the effectiveness of the RTOS-based implementation in managing concurrent tasks efficiently.

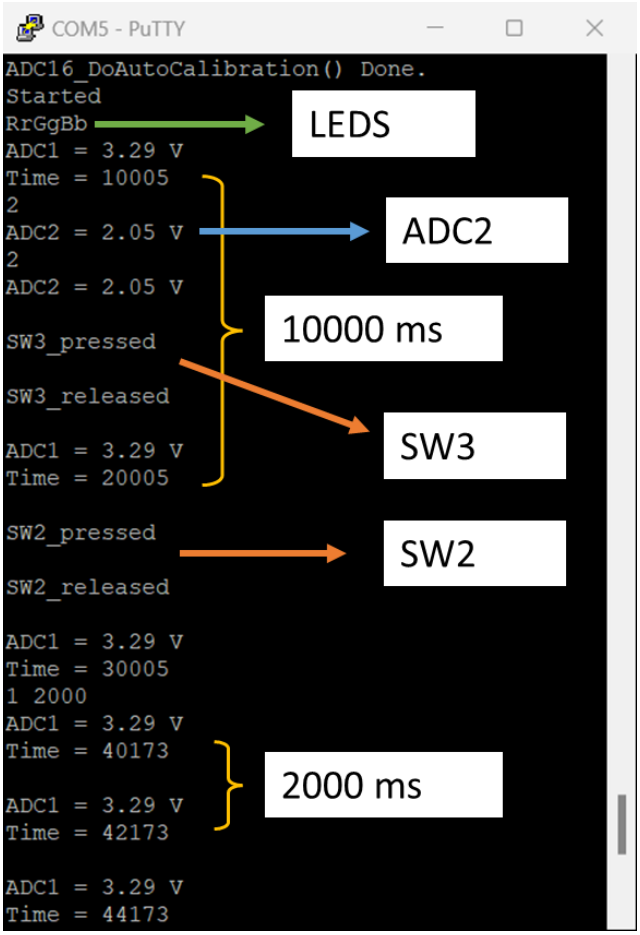

Figure 27 Capture of Putty (Overall system)

6. Discussion

Implementing an RTOS has significantly enhanced system responsiveness and efficiency by facilitating task scheduling, priority management, and inter-thread communication. This ensures timely execution of critical tasks such as ADC sampling, LED control, and display system operation, leading to smoother overall system performance. By prioritizing tasks based on their importance, the RTOS improves system reliability and ensures that essential operations are carried out without delay.

However, there are still some challenges that need to be addressed to further enhance the system's functionality. One such challenge is the difficulty for users to input data when the ADC sampling frequency is high, resulting in potential formatting issues. To mitigate this issue, separating the display panels for user input and data output could provide a more user-friendly experience. Additionally, integrating communication with external devices using protocols like I2C or SPI could expand the system's capabilities and facilitate information exchange with other peripherals.

Furthermore, the presence of noise in ADC conversion remains a problem that needs to be tackled. Implementing a low-pass filter could help mitigate this issue by filtering out unwanted noise, although this solution may require additional memory resources. Overall, addressing these challenges and implementing suggested improvements will further enhance the system's performance and usability, ensuring its effectiveness in real-world applications.

7. Conclusion

In conclusion, this project has successfully developed a comprehensive subsystem for the Kinetis K64F microcontroller, integrating display, LED, and ADC functionalities to enable efficient user interaction and system control. Through the implementation of multiple threads and ISRs, bidirectional communication with the PC, user feedback via switch inputs, LED control, and analogue-to-digital conversion capabilities have been effectively realized.

Part A laid the foundation by establishing threads for PC-K64F communication and integrating ISRs to monitor switch inputs, enhancing user feedback. Part B expanded the system by introducing LED control functionality, allowing for dynamic LED state changes based on user input signals.

In Part C, the system's capabilities were further extended with the addition of threads for periodic and on-demand ADC readings, facilitating data acquisition from Pot 1 and Pot 2. Thread coordination and inter-thread communication mechanisms, such as flags and message passing, ensured efficient data exchange without reliance on global variables.

Utilizing features provided by the Real-Time Operating System (RTOS) and the RTX5 operating system under the CMSIS-RTOS2 wrapper contributed to enhanced software efficiency compared to previous iterations. Overall, this project demonstrates effective utilization of RTOS capabilities and inter-process communication mechanisms to develop a robust and versatile subsystem for the FRDM-K64F board.

Appendix A: Code Implementation

```
/*----------------------------------------------------------------------------
 * CMSIS-RTOS 'main' function template
 *---------------------------------------------------------------------------*/
#include <cmsis_os2.h>
#include "fsl_debug_console.h"
#include "ELEC422.h"
#include "board.h"
#include <string.h>
#include "fsl_adc16.h"
#include <stdio.h>

#define DEMO_ADC16_BASE                 ADC0
#define DEMO_ADC16_CHANNEL_GROUP        0U
#define DEMO_ADC16_USER_CHANNEL         12U /* PTB2, ADC0_SE12 */ /* Pot_1*/
#define DEMO_ADC16_USER_CHANNEL_2       13U /* PTB3, ADC0_SE13 */ /* Pot_2*/

//define the threads allow two switches (SW3, SW2) and LEDswitch(LEDs)
osThreadId_t app_mainid,T_SW3,T_SW2,T_LEDs;

//ADC2 thread and ADCMutex
osThreadId_t T_ADC2;
osMutexId_t ADCMutex;

typedef struct {
   uint8_t     canData[128];
} message_t;
osMemoryPoolId_t memblock, memblock_sampleperiod;
osMessageQueueId_t msgqueue, queue_sampleperiod;

void printstr (char *status)
{
        message_t *message;
        message = (message_t *)osMemoryPoolAlloc(memblock,osWaitForever); //Allocate a memblock
        strcpy((char *)message->canData,(status));                      //Copy status to canData at index 0
        osMessageQueuePut(msgqueue, &message, NULL, osWaitForever);//Post pointer to memory block
}

void printchar(char c) {
   char str[2]; // String to hold the character and null terminator
   str[0] = c;
   str[1] = '\0'; // Null terminator
   printstr(str); // Call printstr with the character converted to a string
}
```

```
uint32_t readnumber(void)
{       uint32_t number = 0;
        char c;
        c = GETCHAR();
        printchar(c);
        if(c == ' '){
                c = GETCHAR();
                printchar(c);
                    while ((c >= '0' && c <= '9')|| c == 127) {
                            if(c==127)
                                number = number / 10 ;
                            else
                                number = number * 10 + (c - '0');
                            c = GETCHAR();
                            printchar(c);
                        }
                }
            return number;
}
/*---------------------------------------------------------------------------
    Thread1: Producer thread for mailbox
 *---------------------------------------------------------------------------*/
static const osThreadAttr_t ThreadAttr_producer = {.name = "producer",};
__NO_RETURN void producer_thread (void  *args) {
message_t *message;                             //Define pointer for memory block
char c;
int sample_period;
    while (1){
            c = GETCHAR();                      //Wait for the user to input a character
            printchar(c);
            if (c == '1'){
                sample_period = readnumber();
            }

            /* Check that time change more than 0*/
            if(sample_period > 0)
                    osMessageQueuePut(queue_sampleperiod, &sample_period, NULL, osWaitForever);

            /* Check condition of c and set flags. */
            if (c == 'R')
                    osThreadFlagsSet (T_LEDs,0x01);
            else if (c == 'r')
                    osThreadFlagsSet (T_LEDs,0x02);
            else if (c == 'G')
                    osThreadFlagsSet (T_LEDs,0x04);
            else if (c == 'g')
                    osThreadFlagsSet (T_LEDs,0x08);
            else if (c == 'B')
                    osThreadFlagsSet (T_LEDs,0x10);
            else if (c == 'b')
                    osThreadFlagsSet (T_LEDs,0x20);
            else if (c == '2')
                    osThreadFlagsSet (T_ADC2,0x01);
    }
}
```

```
/*----------------------------------------------------------------------------
  Thread2: Consumer thread for mailbox
 *----------------------------------------------------------------------------*/
static const osThreadAttr_t ThreadAttr_consumer = {.name = "consumer",};
__NO_RETURN void consumer_thread (void  *args){
message_t *message;
    while (1) {
        osMessageQueueGet(msgqueue, &message, NULL, osWaitForever);
        PRINTF("%s",message->canData);
            //PRINTF message to PC
        osMemoryPoolFree(memblock, message);
        //Release the memblock
    }
}
/*----------------------------------------------------------------------------
  SW2_IRQ
 *----------------------------------------------------------------------------*/
void BOARD_SW2_IRQ_HANDLER(void)
{
   /* Clear external interrupt flag. */
   GPIO_PortClearInterruptFlags(BOARD_SW2_GPIO, 1U << BOARD_SW2_GPIO_PIN);
        /* Read SW2 pin.*/
        uint32_t sw2_value;
        sw2_value = GPIO_PinRead(BOARD_SW2_GPIO, BOARD_SW2_GPIO_PIN);
        /* Set either flag0 or flag1. */
        if (sw2_value == 0)
        {
        osThreadFlagsSet  (T_SW2,0x01);
        }
        else if(sw2_value == 1)
            {
        osThreadFlagsSet  (T_SW2,0x02);
        }
}
/*----------------------------------------------------------------------------
  SW3_IRQ
 *----------------------------------------------------------------------------*/
void BOARD_SW3_IRQ_HANDLER(void)
{
   /* Clear external interrupt flag. */
   GPIO_PortClearInterruptFlags(BOARD_SW3_GPIO, 1U << BOARD_SW3_GPIO_PIN);
        /* Read SW3 pin.*/
        uint32_t sw3_value;
        sw3_value = GPIO_PinRead(BOARD_SW3_GPIO, BOARD_SW3_GPIO_PIN);
        /* Set either flag0 or flag1 */
        if (sw3_value == 0)
        {
        osThreadFlagsSet  (T_SW3,0x01);
        }
        else if(sw3_value == 1)
        {
        osThreadFlagsSet  (T_SW3,0x02);
        }
}
```

```
*---------------------------------------------------------------------------
  SW2_status , SW3_status Threads (Thread3, Thread4)
 *---------------------------------------------------------------------------*/
static const osThreadAttr_t ThreadAttr_SW3_IR = {    .name =   "SW3",   };
static const osThreadAttr_t ThreadAttr_SW2_IR = { .name =     "SW2",   };

__NO_RETURN void SW2_status (void  *argument) {
    for (;;) {
        char str1[128] = "\n\rSW2_pressed\n\r";
        char str2[128] = "\n\rSW2_released\n\r";
        char status[128];

        /* Wait for flag and check flag. */
        uint8_t flag;
        flag = osThreadFlagsWait (0x03,osFlagsWaitAny,osWaitForever);
        /* Check value of flag */
        if (flag == 0x01){
            strcpy(status, str1);
        }
        else if(flag == 0x02) {
            strcpy(status, str2);
        }
        printstr(status);      // Call printstr function
    }
}

__NO_RETURN void SW3_status (void  *argument) {
    for (;;) {
        uint8_t index;
        char str1[128] = "\n\rSW3_pressed\n\r";
        char str2[128] = "\n\rSW3_released\n\r";
        char status[128];

        /* Wait for flag and check flag. */
        uint8_t flag;
        flag = osThreadFlagsWait (0x03,osFlagsWaitAny,osWaitForever);
        /* Check value of flag */
        if (flag == 0x01){
            strcpy(status, str1);
        }
        else if(flag == 0x02){
            strcpy(status, str2);
        }
        printstr(status);          // Call printstr function
    }
}
```

```
/*----------------------------------------------------------------------------
  LEDs_Swicher Thread (Thread5)
 *----------------------------------------------------------------------------*/
static const osThreadAttr_t ThreadAttr_LEDswitcher = { .name =   "LEDswitcher",};

__NO_RETURN void LEDs_switcher(void  *argument) {
    for (;;) {
        /* Wait for flag and check flag. */
        uint8_t flag;
        flag = osThreadFlagsWait(0x3F,osFlagsWaitAny,osWaitForever);
        if (flag    == 0x01)
            {
            LED_On(0);
            }
        else if (flag == 0x02)
            {
            LED_Off(0);
            }
        else if (flag ==  0x04)
            {
            LED_On(1);
            }
        else if (flag ==  0x08)
            {
            LED_Off(1);
            }
        else if (flag ==  0x10)
            {
            LED_On(2);
            }
        else if (flag ==  0x20)
            {
            LED_Off(2);
            }
    }
}
```

```
/*----------------------------------------------------------------------------
    Callback function of Thread6
 *----------------------------------------------------------------------------*/
void ADC1_callback (void  *argument) {
        //Assign channe1_1 to adc16 struct.
        adc16_channel_config_t adc16ChannelConfigStruct;
        adc16ChannelConfigStruct.channelNumber                     = DEMO_ADC16_USER_CHANNEL;
        adc16ChannelConfigStruct.enableDifferentialConversion = false;

        osMutexAcquire(ADCMutex, osWaitForever);  //Ask for Mutex to prevent a collusion in ADC conversion.

        //ADC Conversion
        ADC16_SetChannelConfig(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP,
&adc16ChannelConfigStruct);

        while (0U == (kADC16_ChannelConversionDoneFlag &
ADC16_GetChannelStatusFlags(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP)))
            {
            }
        int g_Adc16ConversionValue_ch1;
        g_Adc16ConversionValue_ch1 = ADC16_GetChannelConversionValue(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP);

        osMutexRelease(ADCMutex);  //Release Mutex after finishing ADC conversion.

        float Adc_value_ch1;
        Adc_value_ch1 = 3.3*g_Adc16ConversionValue_ch1/4096;

        char buffer[128];
        sprintf(buffer, "\n\rADC1 = %.2f V\n\r", Adc_value_ch1); // Format the string "ADC1 = <value>"
        printstr(buffer);      // Call printstr function
}
/*----------------------------------------------------------------------------
    Thread 6: ADC1
 *----------------------------------------------------------------------------*/
//ADC1 timer thread
osTimerId_t T_ADC1_periodic;

static const  osTimerAttr_t timerAttr_timerADC1 = {.name = "timer_ADC1",};
static const osThreadAttr_t ThreadAttr_ADC1 = {.name = "ADC1",};
__NO_RETURN void ADC1_Thread6 (void  *argument) {

    T_ADC1_periodic = osTimerNew(&ADC1_callback, osTimerPeriodic,(void *)0, &timerAttr_timerADC1);
//callback function of Thread6
    uint32_t interval_ms = 10000; //10000ms = 10s
    osTimerStart(T_ADC1_periodic , interval_ms);

    while(1){
            uint32_t newperiod;
            osMessageQueueGet(queue_sampleperiod, &newperiod, NULL, osWaitForever);
            osTimerStop(T_ADC1_periodic);
            osTimerStart(T_ADC1_periodic , newperiod);
    }
}
```

```
*-----------------------------------------------------------------------
    Thread 7: ADC2 to read value of ADC2
 *-----------------------------------------------------------------------*/
static const osThreadAttr_t ThreadAttr_ADC2 = {.name = "ADC2",};
__NO_RETURN void ADC2_Thread7 (void  *argument) {
    for(;;){
        osThreadFlagsWait(0x01,osFlagsWaitAny,osWaitForever);

        //Assign channel_2 to adc16 struct
        adc16_channel_config_t adc16ChannelConfigStruct;
        adc16ChannelConfigStruct.channelNumber                        = DEMO_ADC16_USER_CHANNEL_2;
        adc16ChannelConfigStruct.enableDifferentialConversion = false;

        //Ask for Mutex to prevent a collusion in ADC conversion.
        osMutexAcquire(ADCMutex, osWaitForever);

        //ADC Conversion
        ADC16_SetChannelConfig(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP,
&adc16ChannelConfigStruct);

        while (0U == (kADC16_ChannelConversionDoneFlag &
ADC16_GetChannelStatusFlags(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP)))
            {
    }
        int g_Adc16ConversionValue_ch2;
        g_Adc16ConversionValue_ch2 = ADC16_GetChannelConversionValue(DEMO_ADC16_BASE,
DEMO_ADC16_CHANNEL_GROUP);

        //Release Mutex after finishing ADC conversion.
        osMutexRelease(ADCMutex);

        float Adc_value_ch2;
        Adc_value_ch2 = 3.3*g_Adc16ConversionValue_ch2/4096;

        char buffer[128];
        // Format the string "ADC2 = <value>" and store it in buffer
        sprintf(buffer, "\n\rADC2 = %.2f V\n\r", Adc_value_ch2);

        // Call printstr function
        printstr(buffer);
    }
}
```

```
*---------------------------------------------------------------------------
 Start the RTOS
 *---------------------------------------------------------------------------*/
static const osMemoryPoolAttr_t memorypoolAttr_memblock ={.name = "memory_pool",};
static const osMemoryPoolAttr_t memorypoolAttr_mem_sampleperiod ={.name = "memory_sampleperiod",};
static const osThreadAttr_t ThreadAttr_main = {.name = "main",};
static const osThreadAttr_t ThreadAttr_app_main = {  .name = "app_main",  };
static const osMutexAttr_t MutexAttr_ADCMutex = {.name = "ADCMutex",};


void app_main (void *argument) {
   memblock = osMemoryPoolNew(16, sizeof(message_t),&memorypoolAttr_memblock );
   msgqueue = osMessageQueueNew(16, 4, NULL);
   queue_sampleperiod = osMessageQueueNew(16, 4, NULL);

    osThreadNew(producer_thread, NULL, &ThreadAttr_producer);              //Thread1: Producer
    osThreadNew(consumer_thread, NULL, &ThreadAttr_consumer);              //Thread2: Consumer

    T_SW2 = osThreadNew(SW2_status, NULL, &ThreadAttr_SW2_IR);         //Thread3: SW2_status
    T_SW3 = osThreadNew(SW3_status, NULL, &ThreadAttr_SW3_IR);         //Thread4: SW3_status

    T_LEDs = osThreadNew(LEDs_switcher, NULL, &ThreadAttr_LEDswitcher);  //Thread5: LEDs_switcher

    ADCMutex = osMutexNew(&MutexAttr_ADCMutex );  //Mutex prevents ADC conversion collision

        osThreadNew(ADC1_Thread6, NULL, &ThreadAttr_ADC1);                  //Thread6: ADC1
        T_ADC2 = osThreadNew(ADC2_Thread7, NULL, &ThreadAttr_ADC2);    //Thread7: ADC2

        app_mainid = osThreadGetId();
        osThreadTerminate(app_mainid);
}

    int main (void) {
    char c;
  SystemCoreClockUpdate();
  elec422_startup();
    PRINTF("Started\n\r");
  osKernelInitialize();                                 // Initialize CMSIS-RTOS
  osThreadNew(app_main, NULL, &ThreadAttr_main);        // Create application main thread
  if (osKernelGetState() == osKernelReady)
    {
   osKernelStart();                                     // Start thread execution
  }
  while(1);
}
```