Purdue University Fort Wayne

ECE 485

Lab 1: Graphics, LCD, ADC, Timer and Interpreter

Yilei (Eddie) Li, Rishi Mitra

10/11/2021

1.0 Objective

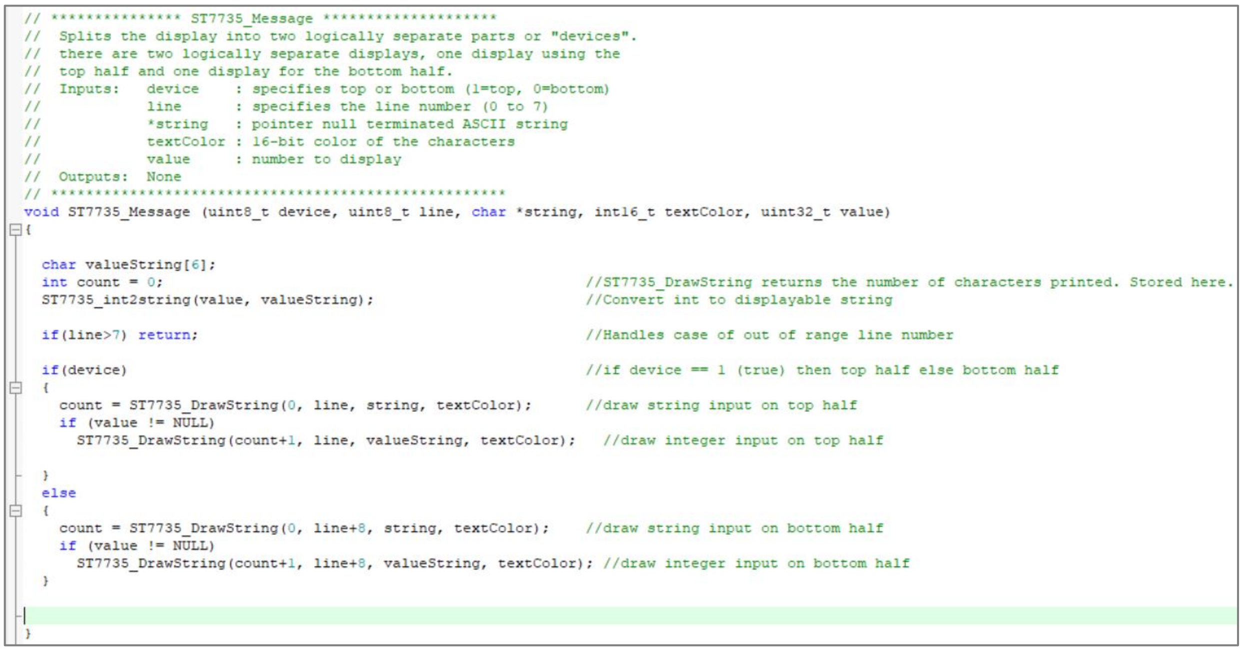
The objective of the lab is to get familiarized with the TI TM4C123G LaunchPad board, µVision development system and the TM4C123 ARM Cortex-M4 microcontroller.

- Extend the device driver for the LCD so that there are two logically separate displays, using the top half and the bottom half.
- Design and test an ADC driver that collects data on any one of the ADC inputs ADC0 to ADC3 with the sampling rate to vary from 100 to 10000 Hz.
- Develop a main program that implements an interpreter using the serial port and interrupting I/O.
- Design and test a system time driver using 24-bit system tick interrupt.

2.0 Software Design

a. Low level LCD drivers

Figure 1 contains the message function ST7735_Message() which splits ST7735 LCD screen into two logically separated parts – top and bottom – and displays messages depending on the inputted arguments (top/bottom, line number, string to be displayed, color of the text and the numerical value to be inputted). The prototype for the function is declared in the ST7735.h file and the function definition is declared in ST7735.c file (seen in Figure 1 below).

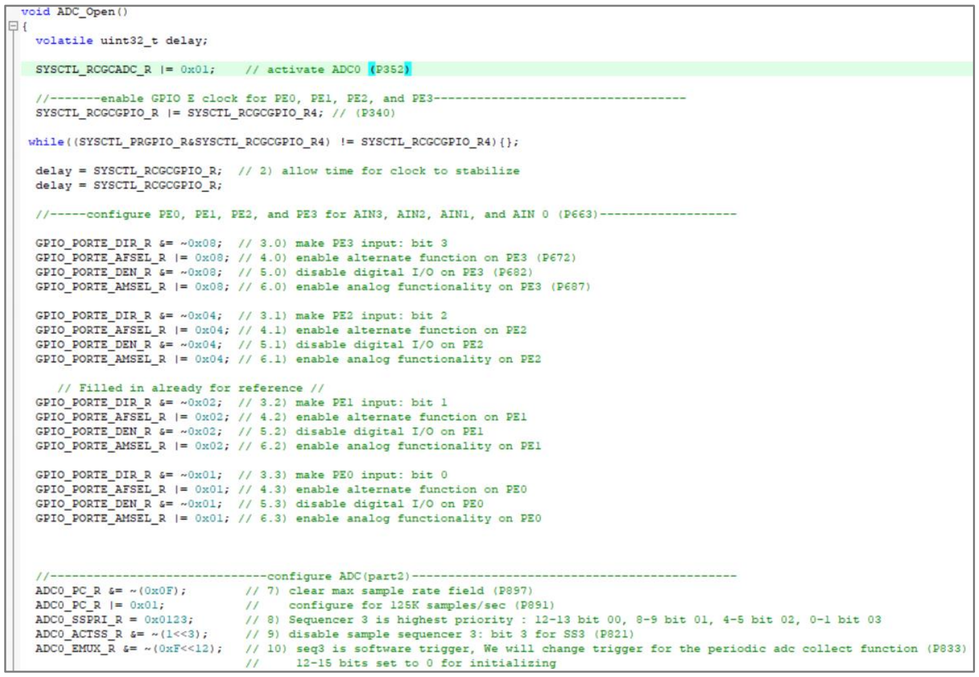


```c
// *************** ST7735_Message ********************
//  Splits the display into two logically separate parts or "devices".
//  there are two logically separate displays, one display using the
//  top half and one display for the bottom half.
//  Inputs:   device    : specifies top or bottom (1=top, 0=bottom)
//            line      : specifies the line number (0 to 7)
//            *string   : pointer null terminated ASCII string
//            textColor : 16-bit color of the characters
//            value     : number to display
//  Outputs:  None
// ****************************************************
void ST7735_Message (uint8_t device, uint8_t line, char *string, int16_t textColor, uint32_t value)
{

  char valueString[6];
  int count = 0;                                        //ST7735_DrawString returns the number of characters printed. Stored here.
  ST7735_int2string(value, valueString);                //Convert int to displayable string

  if(line>7) return;                                    //Handles case of out of range line number

  if(device)                                            //if device == 1 (true) then top half else bottom half
  {
    count = ST7735_DrawString(0, line, string, textColor);      //draw string input on top half
    if (value != NULL)
      ST7735_DrawString(count+1, line, valueString, textColor);   //draw integer input on top half

  }
  else
  {
    count = ST7735_DrawString(0, line+8, string, textColor);    //draw string input on bottom half
    if (value != NULL)
      ST7735_DrawString(count+1, line+8, valueString, textColor); //draw integer input on bottom half
  }

}
```

Figure 1: ST7735_Message function.

b. Low level ADC driver

The initialization function ADC_Open() activates ADC0, enables GPIO, configures ports PE3 and PE3 as input, and configures ADC before the moving on to sampling.

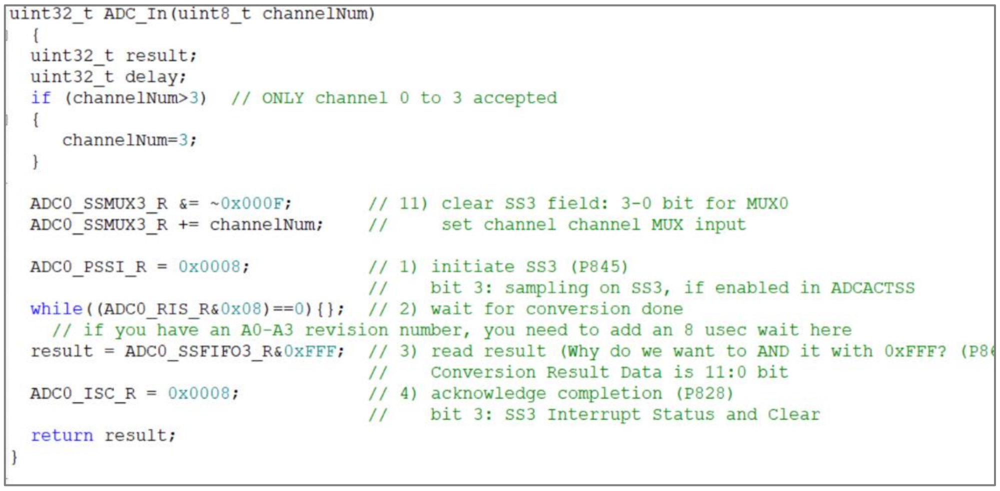


```
void ADC_Open()
{
  volatile uint32_t delay;

  SYSCTL_RCGCADC_R |= 0x01;    // activate ADC0 (P352)

  //-------enable GPIO E clock for PE0, PE1, PE2, and PE3----------------------------------
  SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R4; // (P340)

 while((SYSCTL_PRGPIO_R&SYSCTL_RCGCGPIO_R4) != SYSCTL_RCGCGPIO_R4){};

  delay = SYSCTL_RCGCGPIO_R;  // 2) allow time for clock to stabilize
  delay = SYSCTL_RCGCGPIO_R;

  //-----configure PE0, PE1, PE2, and PE3 for AIN3, AIN2, AIN1, and AIN 0 (P663)------------------

  GPIO_PORTE_DIR_R &= ~0x08;  // 3.0) make PE3 input: bit 3
  GPIO_PORTE_AFSEL_R |= 0x08; // 4.0) enable alternate function on PE3 (P672)
  GPIO_PORTE_DEN_R &= ~0x08;  // 5.0) disable digital I/O on PE3 (P682)
  GPIO_PORTE_AMSEL_R |= 0x08; // 6.0) enable analog functionality on PE3 (P687)

  GPIO_PORTE_DIR_R &= ~0x04;  // 3.1) make PE2 input: bit 2
  GPIO_PORTE_AFSEL_R |= 0x04; // 4.1) enable alternate function on PE2
  GPIO_PORTE_DEN_R &= ~0x04;  // 5.1) disable digital I/O on PE2
  GPIO_PORTE_AMSEL_R |= 0x04; // 6.1) enable analog functionality on PE2

    // Filled in already for reference //
  GPIO_PORTE_DIR_R &= ~0x02;  // 3.2) make PE1 input: bit 1
  GPIO_PORTE_AFSEL_R |= 0x02; // 4.2) enable alternate function on PE1
  GPIO_PORTE_DEN_R &= ~0x02;  // 5.2) disable digital I/O on PE1
  GPIO_PORTE_AMSEL_R |= 0x02; // 6.2) enable analog functionality on PE1

  GPIO_PORTE_DIR_R &= ~0x01;  // 3.3) make PE0 input: bit 0
  GPIO_PORTE_AFSEL_R |= 0x01; // 4.3) enable alternate function on PE0
  GPIO_PORTE_DEN_R &= ~0x01;  // 5.3) disable digital I/O on PE0
  GPIO_PORTE_AMSEL_R |= 0x01; // 6.3) enable analog functionality on PE0



  //------------------------------configure ADC(part2)--------------------------------------------
  ADC0_PC_R &= ~(0x0F);         // 7) clear max sample rate field (P897)
  ADC0_PC_R |= 0x01;            //    configure for 125K samples/sec (P891)
  ADC0_SSPRI_R = 0x0123;        // 8) Sequencer 3 is highest priority : 12-13 bit 00, 8-9 bit 01, 4-5 bit 02, 0-1 bit 03
  ADC0_ACTSS_R &= ~(1<<3);      // 9) disable sample sequencer 3: bit 3 for SS3 (P821)
  ADC0_EMUX_R &= ~(0xF<<12);    // 10) seq3 is software trigger, We will change trigger for the periodic adc collect function (P833)
                                //     12-15 bits set to 0 for initializing
```

Figure 2: ADC_Open function

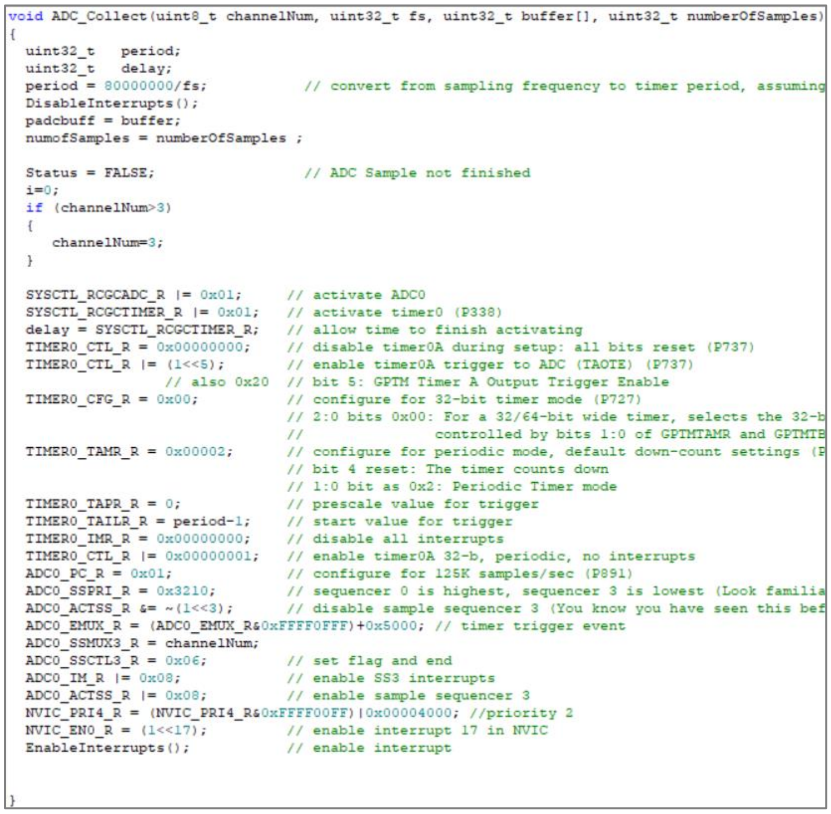
There are two modes of sampling:

ADC_In() takes one-shot ADC sample (shown in figure 3)
ADC_collect() collects a given number of samples from a given channel at a given frequency (shown in figure 4)



```
uint32_t ADC_In(uint8_t channelNum)
 {
 uint32_t result;
 uint32_t delay;
 if (channelNum>3)  // ONLY channel 0 to 3 accepted
 {
    channelNum=3;
 }

 ADC0_SSMUX3_R &= ~0x000F;       // 11) clear SS3 field: 3-0 bit for MUX0
 ADC0_SSMUX3_R += channelNum;    //     set channel channel MUX input

 ADC0_PSSI_R = 0x0008;           // 1) initiate SS3 (P845)
                                 //    bit 3: sampling on SS3, if enabled in ADCACTSS
 while((ADC0_RIS_R&0x08)==0){};  // 2) wait for conversion done
   // if you have an A0-A3 revision number, you need to add an 8 usec wait here
 result = ADC0_SSFIFO3_R&0xFFF;  // 3) read result (Why do we want to AND it with 0xFFF? (P8
                                 //    Conversion Result Data is 11:0 bit
 ADC0_ISC_R = 0x0008;            // 4) acknowledge completion (P828)
                                 //    bit 3: SS3 Interrupt Status and Clear
 return result;
}
```

Figure 3: ADC_In function

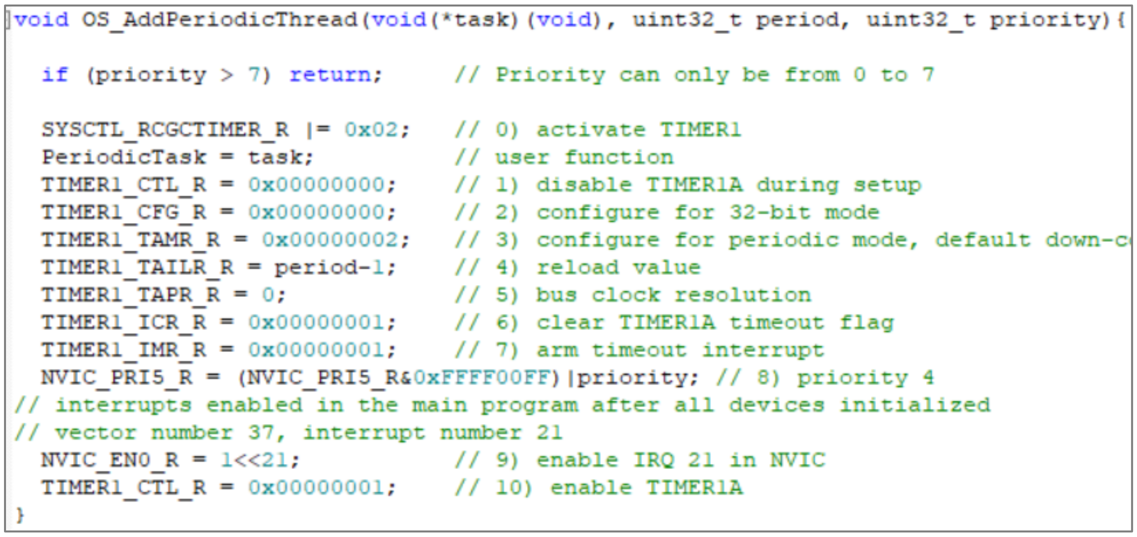


```c
void ADC_Collect(uint8_t channelNum, uint32_t fs, uint32_t buffer[], uint32_t numberOfSamples)
{
  uint32_t   period;
  uint32_t   delay;
  period = 80000000/fs;            // convert from sampling frequency to timer period, assuming
  DisableInterrupts();
  padcbuff = buffer;
  numofSamples = numberOfSamples ;

  Status = FALSE;                  // ADC Sample not finished
  i=0;
  if (channelNum>3)
  {
     channelNum=3;
  }

  SYSCTL_RCGCADC_R |= 0x01;      // activate ADC0
  SYSCTL_RCGCTIMER_R |= 0x01;    // activate timer0 (P338)
  delay = SYSCTL_RCGCTIMER_R;    // allow time to finish activating
  TIMER0_CTL_R = 0x00000000;     // disable timer0A during setup: all bits reset (P737)
  TIMER0_CTL_R |= (1<<5);        // enable timer0A trigger to ADC (TAOTE) (P737)
                 // also 0x20  // bit 5: GPTM Timer A Output Trigger Enable
  TIMER0_CFG_R = 0x00;           // configure for 32-bit timer mode (P727)
                                 // 2:0 bits 0x00: For a 32/64-bit wide timer, selects the 32-b
                                 //                 controlled by bits 1:0 of GPTMTAMR and GPTMTB
  TIMER0_TAMR_R = 0x00002;       // configure for periodic mode, default down-count settings (P
                                 // bit 4 reset: The timer counts down
                                 // 1:0 bit as 0x2: Periodic Timer mode
  TIMER0_TAPR_R = 0;             // prescale value for trigger
  TIMER0_TAILR_R = period-1;     // start value for trigger
  TIMER0_IMR_R = 0x00000000;     // disable all interrupts
  TIMER0_CTL_R |= 0x00000001;    // enable timer0A 32-b, periodic, no interrupts
  ADC0_PC_R = 0x01;              // configure for 125K samples/sec (P891)
  ADC0_SSPRI_R = 0x3210;         // sequencer 0 is highest, sequencer 3 is lowest (Look familia
  ADC0_ACTSS_R &= ~(1<<3);       // disable sample sequencer 3 (You know you have seen this bef
  ADC0_EMUX_R = (ADC0_EMUX_R&0xFFFF0FFF)+0x5000; // timer trigger event
  ADC0_SSMUX3_R = channelNum;
  ADC0_SSCTL3_R = 0x06;          // set flag and end
  ADC0_IM_R |= 0x08;             // enable SS3 interrupts
  ADC0_ACTSS_R |= 0x08;          // enable sample sequencer 3
  NVIC_PRI4_R = (NVIC_PRI4_R&0xFFFF00FF)|0x00004000; //priority 2
  NVIC_EN0_R = (1<<17);          // enable interrupt 17 in NVIC
  EnableInterrupts();            // enable interrupt


}
```

Figure 4: ADC_Collect function

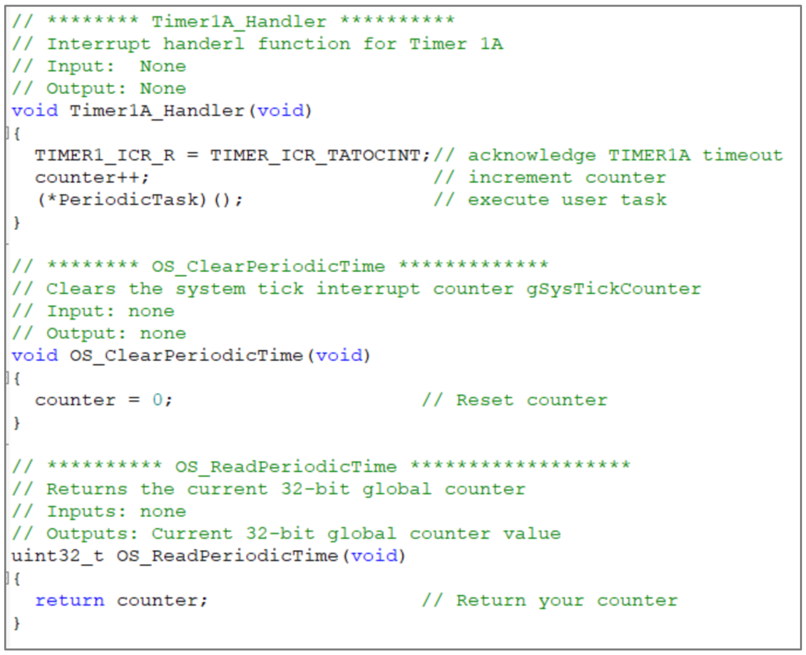
c. Low level timer driver

The OS_AddPeriodicThread() activate timer 1, initialize system tick interrupt, and limits the priority between 0 and 7. The user task is passed to the global pointer PeriodicTask, and gets called in the Timer1A_Handler() which is executed at every instance when timer 1 times out, hence periodically. The system tick interrupt counter is incremented in the Timer1A_Handler() to keep track of the number of times an interrupt takes place. This counter is cleared by *OS_ClearPeriodicTime()* and its value can be read by *OS_ReadPeriodicTime().*



```
void OS_AddPeriodicThread(void(*task)(void), uint32_t period, uint32_t priority){

  if (priority > 7) return;     // Priority can only be from 0 to 7

  SYSCTL_RCGCTIMER_R |= 0x02;   // 0) activate TIMER1
  PeriodicTask = task;          // user function
  TIMER1_CTL_R = 0x00000000;    // 1) disable TIMER1A during setup
  TIMER1_CFG_R = 0x00000000;    // 2) configure for 32-bit mode
  TIMER1_TAMR_R = 0x00000002;   // 3) configure for periodic mode, default down-c
  TIMER1_TAILR_R = period-1;    // 4) reload value
  TIMER1_TAPR_R = 0;            // 5) bus clock resolution
  TIMER1_ICR_R = 0x00000001;    // 6) clear TIMER1A timeout flag
  TIMER1_IMR_R = 0x00000001;    // 7) arm timeout interrupt
  NVIC_PRI5_R = (NVIC_PRI5_R&0xFFFF00FF)|priority; // 8) priority 4
// interrupts enabled in the main program after all devices initialized
// vector number 37, interrupt number 21
  NVIC_EN0_R = 1<<21;           // 9) enable IRQ 21 in NVIC
  TIMER1_CTL_R = 0x00000001;    // 10) enable TIMER1A
}
```

Figure 5: OS_AddPeriodicThread

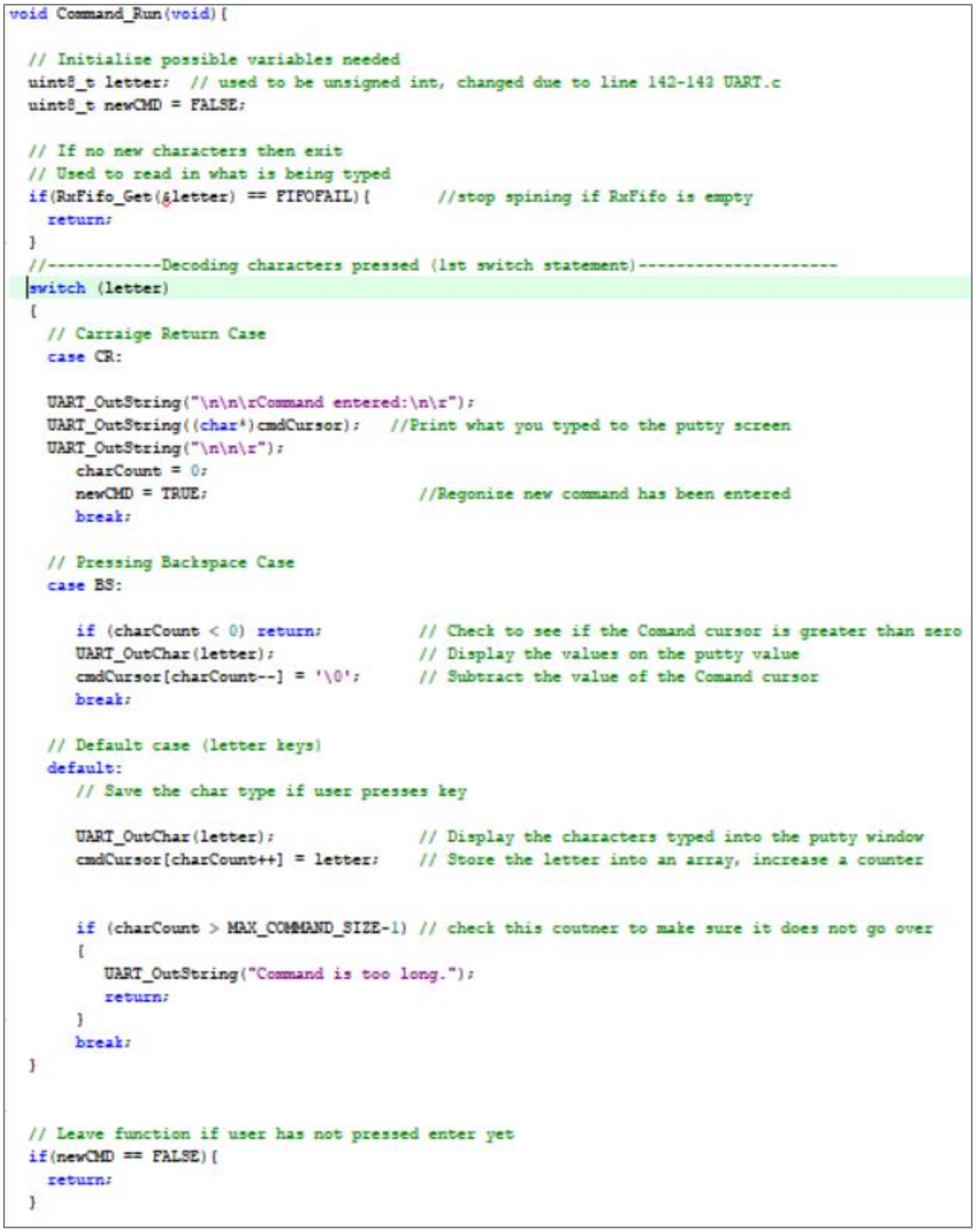


```
// ******** Timer1A_Handler **********
// Interrupt handerl function for Timer 1A
// Input:  None
// Output: None
void Timer1A_Handler(void)
{
  TIMER1_ICR_R = TIMER_ICR_TATOCINT;// acknowledge TIMER1A timeout
  counter++;                        // increment counter
  (*PeriodicTask)();                // execute user task
}

// ******** OS_ClearPeriodicTime *************
// Clears the system tick interrupt counter gSysTickCounter
// Input: none
// Output: none
void OS_ClearPeriodicTime(void)
{
  counter = 0;                     // Reset counter
}

// ********** OS_ReadPeriodicTime *******************
// Returns the current 32-bit global counter
// Inputs: none
// Outputs: Current 32-bit global counter value
uint32_t OS_ReadPeriodicTime(void)
{
  return counter;                  // Return your counter
}
```

Figure 6: The function Timer1A_Handler, OS_ClearPeriodicTime, and OS_ReadPeriodicTime

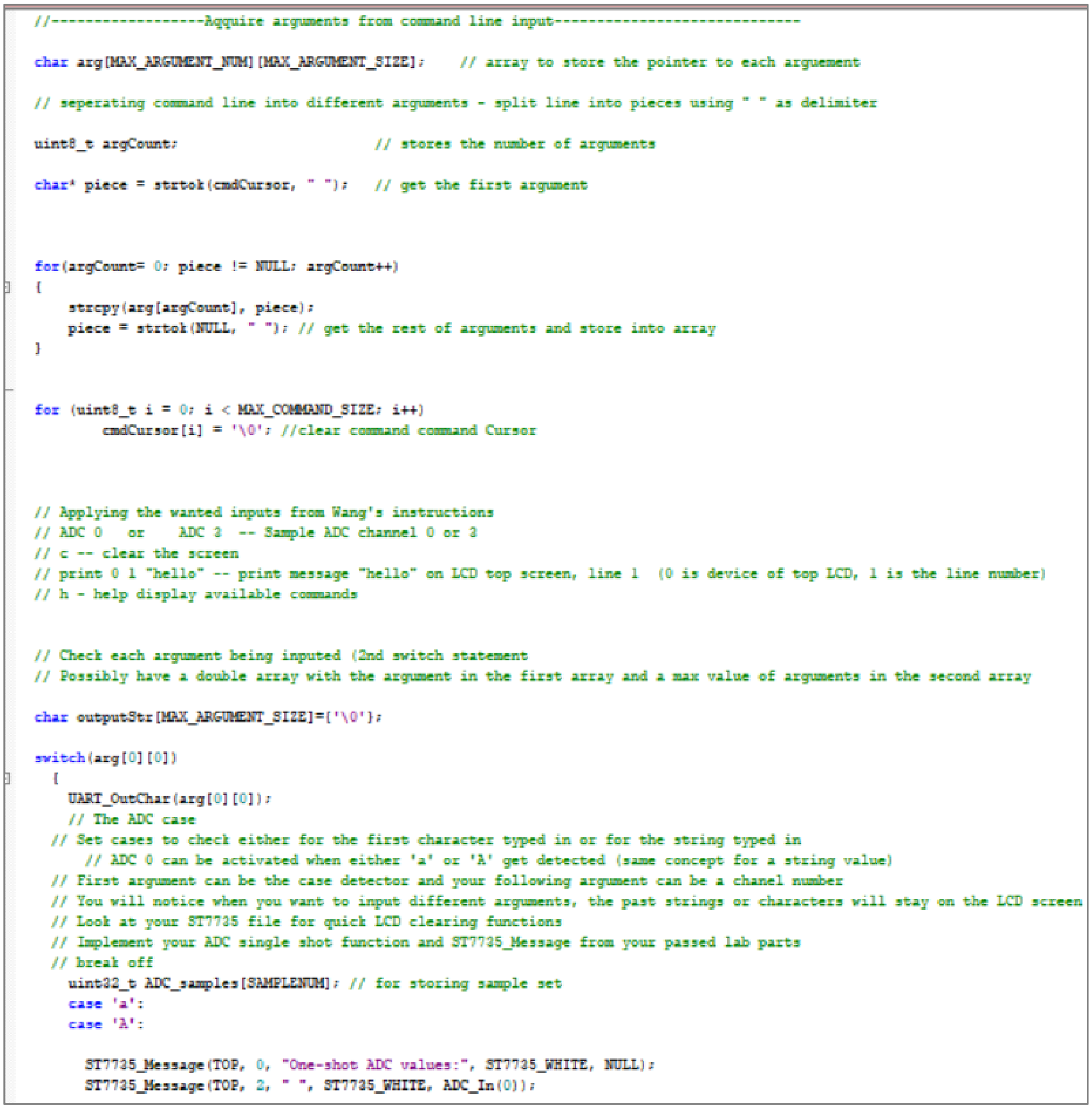
d. High level main program

A command line interface was created to interpret a set of commands to be run and display corresponding results on the LCD. This was achieved via the Command_Run() function defined in UART.c. A FIFO was implemented to accept keystrokes via PuTTY. The characters are stored, parsed and fed to a LUT implemented using switch cases to recognize commands and execute their respective actions. The ST7735_Message() function defined earlier is then used to display results on the LCD.



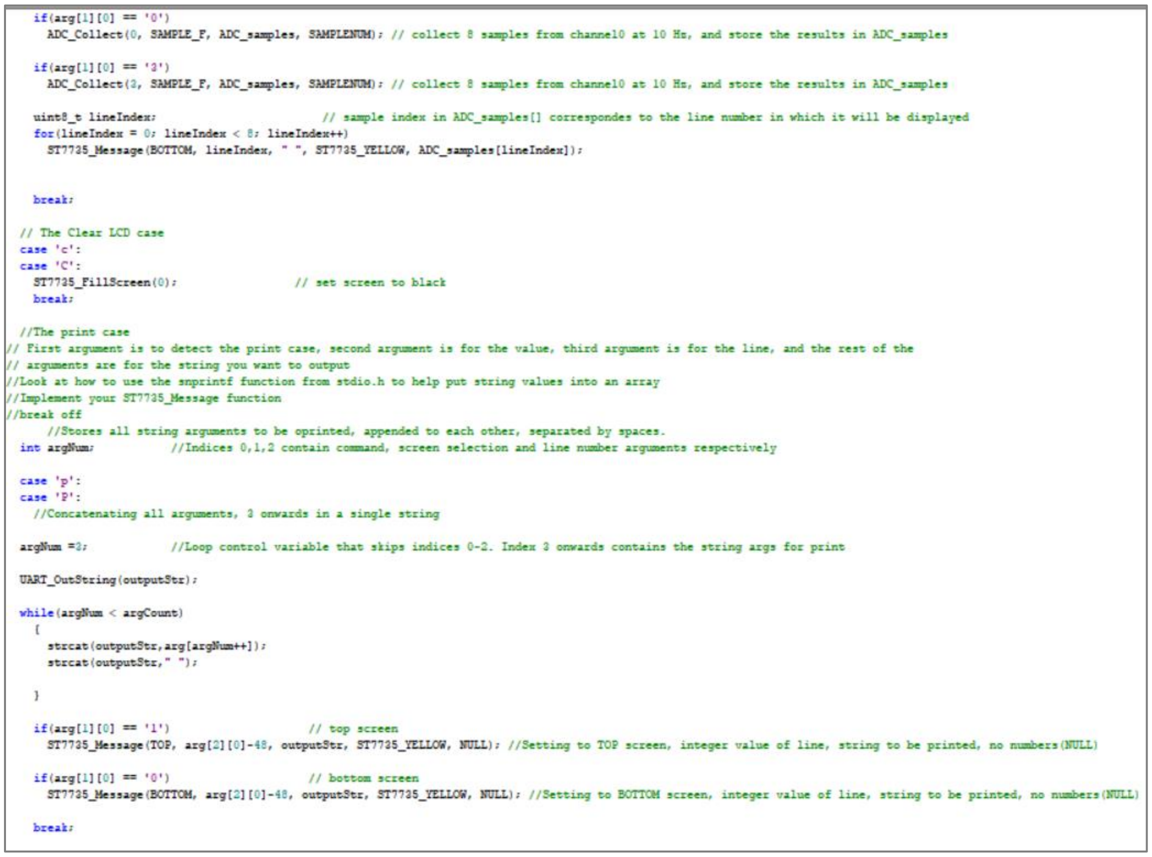
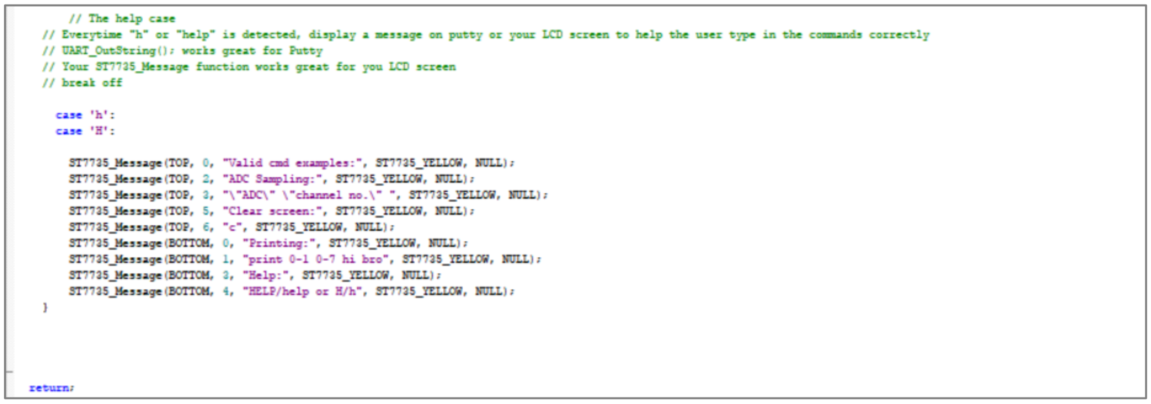
```
void Command_Run(void){

  // Initialize possible variables needed
  uint8_t letter;  // used to be unsigned int, changed due to line 142-143 UART.c
  uint8_t newCMD = FALSE;

  // If no new characters then exit
  // Used to read in what is being typed
  if(RxFifo_Get(&letter) == FIFOFAIL){       //stop spining if RxFifo is empty
    return;
  }
  //------------Decoding characters pressed (1st switch statement)---------------------
  switch (letter)
  {
    // Carraige Return Case
    case CR:

    UART_OutString("\n\n\rCommand entered:\n\r");
    UART_OutString((char*)cmdCursor);   //Print what you typed to the putty screen
    UART_OutString("\n\n\r");
       charCount = 0;
       newCMD = TRUE;                      //Regonize new command has been entered
       break;

    // Pressing Backspace Case
    case BS:

       if (charCount < 0) return;          // Check to see if the Comand cursor is greater than zero
       UART_OutChar(letter);               // Display the values on the putty value
       cmdCursor[charCount--] = '\0';      // Subtract the value of the Comand cursor
       break;

    // Default case (letter keys)
    default:
       // Save the char type if user presses key

       UART_OutChar(letter);               // Display the characters typed into the putty window
       cmdCursor[charCount++] = letter;    // Store the letter into an array, increase a counter


       if (charCount > MAX_COMMAND_SIZE-1) // check this coutner to make sure it does not go over
       {
          UART_OutString("Command is too long.");
          return;
       }
       break;
  }


  // Leave function if user has not pressed enter yet
  if(newCMD == FALSE){
    return;
  }
```

Figure 7. The Command_Run() function accepting chars from the FIFO and a switch case for incoming characters



```
//------------------Aqquire arguments from command line input-----------------------------

char arg[MAX_ARGUMENT_NUM][MAX_ARGUMENT_SIZE];    // array to store the pointer to each arguement

// seperating command line into different arguments - split line into pieces using " " as delimiter

uint8_t argCount;                        // stores the number of arguments

char* piece = strtok(cmdCursor, " ");    // get the first argument



for(argCount= 0; piece != NULL; argCount++)
{
    strcpy(arg[argCount], piece);
    piece = strtok(NULL, " "); // get the rest of arguments and store into array
}


for (uint8_t i = 0; i < MAX_COMMAND_SIZE; i++)
        cmdCursor[i] = '\0'; //clear command command Cursor



// Applying the wanted inputs from Wang's instructions
// ADC 0    or    ADC 3  -- Sample ADC channel 0 or 3
// c -- clear the screen
// print 0 1 "hello" -- print message "hello" on LCD top screen, line 1  (0 is device of top LCD, 1 is the line number)
// h - help display available commands


// Check each argument being inputed (2nd switch statement
// Possibly have a double array with the argument in the first array and a max value of arguments in the second array

char outputStr[MAX_ARGUMENT_SIZE]={'\0'};

switch(arg[0][0])
  {
    UART_OutChar(arg[0][0]);
    // The ADC case
  // Set cases to check either for the first character typed in or for the string typed in
      // ADC 0 can be activated when either 'a' or 'A' get detected (same concept for a string value)
  // First argument can be the case detector and your following argument can be a chanel number
  // You will notice when you want to input different arguments, the past strings or characters will stay on the LCD screen
  // Look at your ST7735 file for quick LCD clearing functions
  // Implement your ADC single shot function and ST7735_Message from your passed lab parts
  // break off
    uint32_t ADC_samples[SAMPLENUM]; // for storing sample set
    case 'a':
    case 'A':

      ST7735_Message(TOP, 0, "One-shot ADC values:", ST7735_WHITE, NULL);
      ST7735_Message(TOP, 2, " ", ST7735_WHITE, ADC_In(0));
```

Figure 8. The command and its arguments are tokenized, parsed and stored in a 2D array

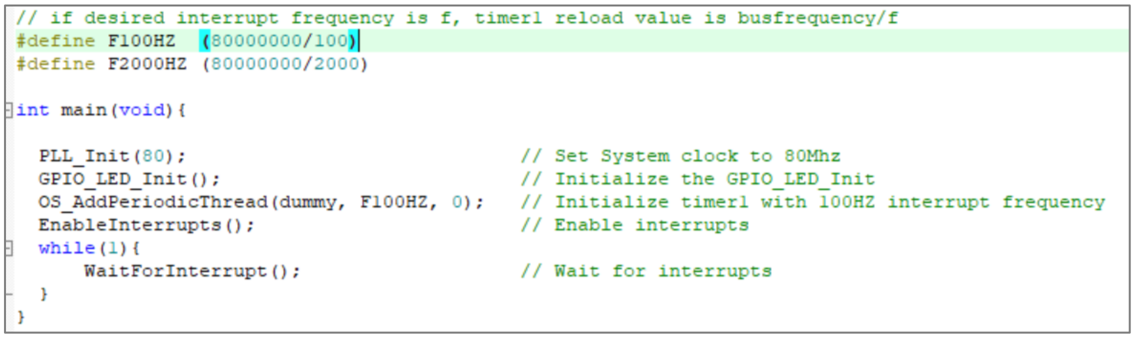
```c
    if(arg[1][0] == '0')
      ADC_Collect(0, SAMPLE_F, ADC_samples, SAMPLENUM); // collect 8 samples from channel0 at 10 Hz, and store the results in ADC_samples

    if(arg[1][0] == '3')
      ADC_Collect(3, SAMPLE_F, ADC_samples, SAMPLENUM); // collect 8 samples from channel0 at 10 Hz, and store the results in ADC_samples

    uint8_t lineIndex;                        // sample index in ADC_samples[] correspondes to the line number in which it will be displayed
    for(lineIndex = 0; lineIndex < 8; lineIndex++)
      ST7735_Message(BOTTOM, lineIndex, " ", ST7735_YELLOW, ADC_samples[lineIndex]);


    break;

  // The Clear LCD case
  case 'c':
  case 'C':
    ST7735_FillScreen(0);                 // set screen to black
    break;

  //The print case
// First argument is to detect the print case, second argument is for the value, third argument is for the line, and the rest of the
// arguments are for the string you want to output
//Look at how to use the snprintf function from stdio.h to help put string values into an array
//Implement your ST7735_Message function
//break off
      //Stores all string arguments to be oprinted, appended to each other, separated by spaces.
  int argNum;           //Indices 0,1,2 contain command, screen selection and line number arguments respectively

  case 'p':
  case 'P':
    //Concatenating all arguments, 3 onwards in a single string

  argNum =3;            //Loop control variable that skips indices 0-2. Index 3 onwards contains the string args for print

  UART_OutString(outputStr);

  while(argNum < argCount)
    {
      strcat(outputStr,arg[argNum++]);
      strcat(outputStr," ");

    }

    if(arg[1][0] == '1')                    // top screen
      ST7735_Message(TOP, arg[2][0]-48, outputStr, ST7735_YELLOW, NULL); //Setting to TOP screen, integer value of line, string to be printed, no numbers(NULL)

    if(arg[1][0] == '0')                    // bottom screen
      ST7735_Message(BOTTOM, arg[2][0]-48, outputStr, ST7735_YELLOW, NULL); //Setting to BOTTOM screen, integer value of line, string to be printed, no numbers(NULL)

    break;
```

```c
      // The help case
   // Everytime "h" or "help" is detected, display a message on putty or your LCD screen to help the user type in the commands correctly
   // UART_OutString(); works great for Putty
   // Your ST7735_Message function works great for you LCD screen
   // break off

     case 'h':
     case 'H':

      ST7735_Message(TOP, 0, "Valid cmd examples:", ST7735_YELLOW, NULL);
      ST7735_Message(TOP, 2, "ADC Sampling:", ST7735_YELLOW, NULL);
      ST7735_Message(TOP, 3, "\"ADC\" \"channel no.\" ", ST7735_YELLOW, NULL);
      ST7735_Message(TOP, 5, "Clear screen:", ST7735_YELLOW, NULL);
      ST7735_Message(TOP, 6, "c", ST7735_YELLOW, NULL);
      ST7735_Message(BOTTOM, 0, "Printing:", ST7735_YELLOW, NULL);
      ST7735_Message(BOTTOM, 1, "print 0-1 0-7 hi bro", ST7735_YELLOW, NULL);
      ST7735_Message(BOTTOM, 3, "Help:", ST7735_YELLOW, NULL);
      ST7735_Message(BOTTOM, 4, "HELP/help or H/h", ST7735_YELLOW, NULL);
   }



  return;
```

Figure 9. Switch case to look up commands and initiate their executions

3.0 Measured Time

The value been passed into function OS_AddPeriodicThread() as the load value for timer 1 is (80000000/100) as shown in Figure 10 below. This is to achieve the desired interrupt frequency 100 Hz. Therefore, the estimated time to run the periodic timer interrupt is 0.01s.



```
// if desired interrupt frequency is f, timer1 reload value is busfrequency/f
#define F100HZ  (80000000/100)
#define F2000HZ (80000000/2000)

int main(void){

  PLL_Init(80);                          // Set System clock to 80Mhz
  GPIO_LED_Init();                       // Initialize the GPIO_LED_Init
  OS_AddPeriodicThread(dummy, F100HZ, 0); // Initialize timer1 with 100HZ interrupt frequency
  EnableInterrupts();                    // Enable interrupts
  while(1){
      WaitForInterrupt();                // Wait for interrupts
  }
}
```

Figure 9. Interrupt frequency defined and used

The interrupt period is measured using the value of the global counter. Its value increments from 0 to 255 in around 41 seconds(as shown in figure10). Therefore, the measured timer is 41/255 = 0.16 sec. Since the user task defined in the dummy function is delayed for 5 ticks and each task involves 8 interrupts to complete. The number of times this task is being executed is actually 255/(5*8) = 6.4

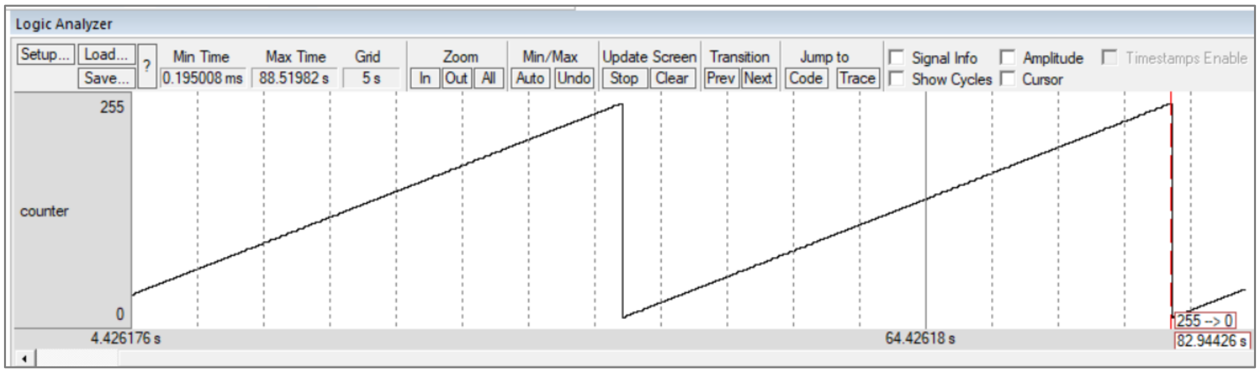


Figure10: Measuring interrupt frequency using logic analyzer

4.0 Analysis and Discussions

1) What are the range, resolution, and precision of the ADC?

The range of the ADC was 0-3.3V is the maximum and minimum ADC input. with a resolution of 0.805mV input voltage by 2^n where n is the number of bits and since we are using the 12-bit ADC and precision of 4096 number of distinguishable inputs.

2) The ADC samples the same voltage; however, it comes out with 8 different values, please explain why?

Even though ADC collects samples at the same voltage it returns 8 different values because an ADC converts a continuous-time and continuous-amplitude analog signal to a discrete-time and discrete-amplitude digital signal. The conversion involves quantization of the input, so it necessarily introduces a small amount of error or noise. Furthermore, instead of continuously performing the conversion, an ADC does the conversion periodically, sampling the input, limiting the allowable bandwidth of the input signal.

3) **List the ways the ADC conversion can be started. Explain why you choose the way you did.**

ADC conversion can be started in many ways, these being:

a. Direct-conversion or flash ADC which has a bank of comparators sampling the input signal in parallel, each firing for a specific voltage range.
b. Successive-approximation ADC which uses a comparator and a binary search to successively narrow a range that contains the input voltage.
c. Ramp-compare ADC which produces a saw-tooth signal that ramps up or down then quickly returns to zero.
d. Integrating ADC (also dual-slope or multi-slope ADC) which applies the unknown input voltage to the input of an integrator and allows the voltage to ramp for a fixed time period.
e. Time-interleaved ADC which uses M parallel ADCs where each ADC samples data every M:th cycle of the effective sample clock.

The method we used was Time-interleaved ADC since the sample rate is increased M times compared to what each individual ADC can manage.

4) **The measured time to run the periodic interrupt can be measured directly by setting a bit high at the start of the ISR and clearing that bit at the end of the ISR. It could also be measured through a counting timer. How did you measure it? Compare and contrast your method to these two.**

In this lab, a global counter is declared to keep track of the number of time when an interrupt is trigged. The average time to run an execution was found by using the counter timer method. In this method we observe the value of the global counter in the logic analyzer and the deference between 2 peaks as the measured time.

This method compared to measuring it by setting a bit high at the start of the ISR and clearing that bit at the end of the ISR it is easier to track since counter can be inputted as a signal in the logic analyzer.

5) **Divide the time to execute once instance of the ISR by the total instructions in the ISR it to get the average time to execute an instruction. Compare this to the 12.5 ns system clock period (80 MHz).**

Execution time: 0.16 sec
User task execution count: 6.4
Timer per task: 0.025 sec

Comparison: 80,000,000 Hz * 0.025 sec = 2,000,000 times

6) **What are the range, resolution, and precision of the SysTick timer? I.e., answer this question relative to the NVIC_ST_CURRENT_R register in the Cortex M4 core peripherals.**

   The range of the SysTick timer is from 0 to 16777215. The resolution is 12.5ns and the precision is 24 bit.