National University of Singapore

EE2024 Assignment-2

Care Unit for The Elderly (CUTE)

Team members:

Sneha Santha Prabakar (A0147919N)

Rahumath Marini (A0138471A)

Assessing Graduating Assistant:

Truong Vinh Lan

SECTION 1: ASSIGNMENT DESCRIPTION

1.1. INTRODUCTION

Aging population is one of the major issues ~~facing~~ Singapore is facing. It is estimated that by the year 2030, 1 in 4 Singaporeans would be aged 65 years and above. Many of the elderly would be staying alone at home and it is imperative that they live in a safe and secure environment.

This project hence, aims to use the technology of embedded systems to provide a safe and secure environment for the elderly in their homes.

1.2. OBJECTIVE

To create a system, known as Care Unit for The Elderly (shortly, CUTE). CUTE is a portable and cost-effective device that elderly people from all economic backgrounds can use. It requires very less technical expertise and hence, easy for the elderly to use. Any issues that the elderly face in their homes, would be sent wirelessly to an agency known as Centralized Elderly Monitoring System (CEMS) so that they can proactively take measures to solve the issue.

CUTE would be built on a LPC1769 board, and would make use of external peripherals such as the temperature sensor, light sensor and the accelerometer to monitor the elderly's movement and the environment in which they live in. Any unsafe conditions such as fire or the elderly's movement in darkness, would be immediately reported to the CEMS so that they can send a caretaker to tackle the situation. If the elderly require any other help, for example to climb up a stairway, they can notify the CEMS using CUTE and receive help. CUTE also simplifies the way the elderly can take control of their environment and change it to suit their own needs. It allows the elderly to adjust the ambient lighting of their home according to their comfort. All these monitoring operations can be performed only when the elderly are alone at home (i.e. in the MONITOR mode), as they are personally taken care of in the presence of a caretaker in the STABLE mode. CUTE makes use of polling at regular time intervals and interrupts to collect data and transmit it to the CEMS in the MONITOR mode.

2.1. EXTERNAL PERIPHERALS

External peripherals were used to collect and output the processed data in the elderly's environment. Some of these external peripherals, such as the OLED display, joystick and the rotary switch form the user-interface components of the CUTE system. The other external peripherals are used to collect data from the surroundings of the elderly.

The following external peripherals were used to design CUTE:

- Temperature Sensor
- Light Sensor
- Accelerometer
- PCA9532 Port Expander (16 LEDs array)
- RGB LEDs
- Xbee RF-module
- 7 Segment Display
- OLED Display
- SW4 push button
- SW3 push button
- Rotary switch
- Joystick

2.2. ON-CHIP PERIPHERALS

On-chip peripherals are required for the communication between the processor and the external peripherals. The on-chip peripherals used while designing CUTE are:

- I2C (Accelerometer, Light sensor, OLED, PCA9532 Port Expander
- SSP (7 Segment Display, OLED)
- GPIO (Joystick, SW3 push button, SW4 push button, RGB LED, Rotary switch, Temperature sensor)
- UART (XBee RF-module)

2.3. CORE PERIPHERALS

The core peripherals form an integral part of the ARM processor. The most important core peripherals that we used while designing CUTE are:

- NVIC/Interrupts
- Systick timer

2.4. MAIN DESIGN OF THE SYSTEM

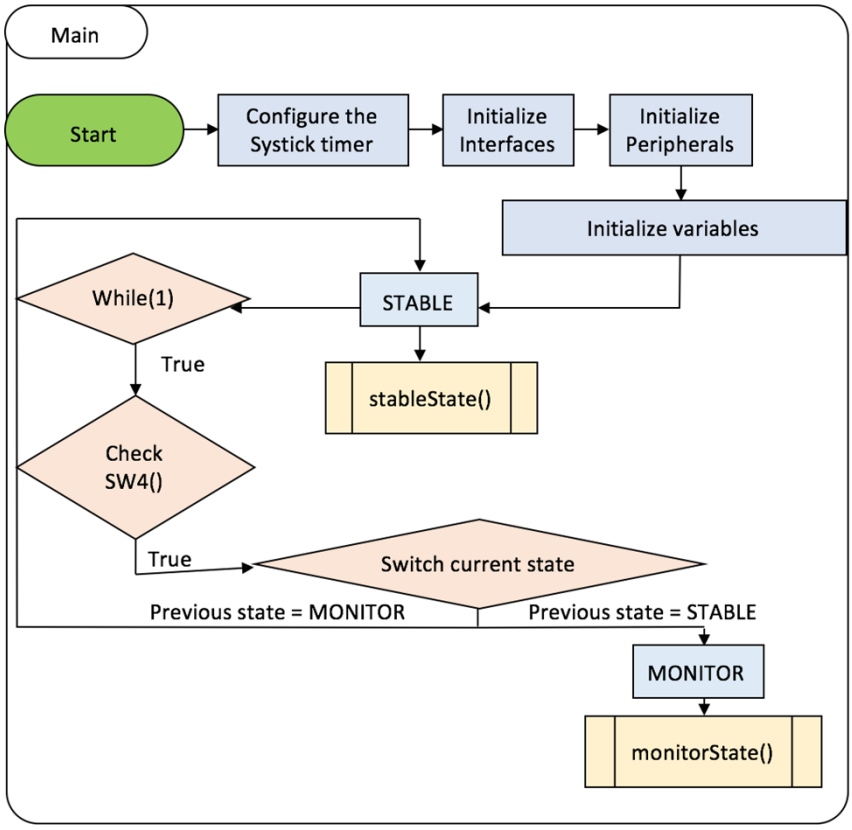


Figure 1: Main Design of CUTE

As illustrated in Figure 1 above, when the LPC1769 board is first connected to a power source, the first step carried out is to configure the Systick timer. The Systick timer is configured to produce 1 interrupt per second. This is the first step carried out as it synchronizes time for the entire system. Next, all the interfaces (such as I2C, GPIO) and interrupts are initialized. Following this, the external peripherals such as 7 Segment LED, light sensor are initialized. After this, some of the variables used in the code are initialized. Along with these, the timers for all peripherals are also initialized to be equal to the current time (given by msTicks).

The CUTE is initialized to be in STABLE mode every time the system turns on, by default.

After initializing all the interfaces, peripherals, interrupts and variables, inside the infinity while(1) loop, the system goes into a polling mode to carry out all the functions mentioned inside the while(1) loop. As such, it first checks whether the pushbutton SW4 is pressed or not, by going into the check_SW4() function. The check_SW4() function first checks if the pushbutton SW4 has been pressed, if YES then it goes on to check the current status of the system. If the current status of the system is STABLE, then it switches the system to the MONITOR state and indicates this transition by printing "Entering Monitor Mode" on the OELD for 1 second[1] and sending this same message to the UART terminal. On the other hand, if the current state of the system is MONITOR, then it switches the current state to STABLE mode and indicates this transition by printing "Entering Stable Mode" on the OLED screen for 1 second[1], before clearing all the display on the OLED (this clearing function is explained more in detail in the stableState() function). An extract of the check_SW4() function is shown below in Figure 2.

After executing the function check_SW4(), the system then detects the current (new) state of the system and guides it to its respective state function. This means that, if the current state is STABLE, then the system executes the stateState() function which contains all the operations that need to be executed in the STABLE mode. Likewise, if the current state is MONITOR, then the system executes the monitorState() funciton which contains all the operations that need to be executed in the MONITOR mode (this function is explained further). The polling continues and the system goes back to check_SW4() as it is inside the infinity loop, when power continues to be on.

[1] This is an enhancement feature; not specified in the assignment requirements. However, it does not affect the basic requirements. When the LPC1769 board is initally powered on, the OLED does not display "Entering Stable Mode" as the assignment basic requirement specifies that nothing should be displayed when the LPC1769 board is first powered on. Even otherwise, there is nothing displayed on the OLED in the Stable state. The messages "Entering Stable Mode" and "Entering Monitor Mode" are displayed only for 1 second during the transition period so that the user is aware of the transition. This feautre aims to improve the user experience. It does not conflict with the basic requirements; it is a seperately implemented enhancement feature.

```
void check_SW4(void){
    uint32_t currTime = getTicks();
    if((currTime - lastDebounceTime) > debounceDelay){
    lastDebounceTime = currTime;
    if(!((GPIO_ReadValue(1) >> 31) & 0x01)){

        if(CURRENT_STATE== STABLE){
            oled_putString (21, 15, "Entering", OLED_COLOR_WHITE, OLED_COLOR_BLACK);

            oled_putString (24, 25, "Monitor", OLED_COLOR_WHITE, OLED_COLOR_BLACK);

            oled_putString (28, 35, "Mode", OLED_COLOR_WHITE, OLED_COLOR_BLACK);

            Timer0_Wait(1000);
            sprintf(entering_msg, "Entering MONITOR Mode");
            strcat(entering_msg, "\r\n");
            UART_SendString(LPC_UART3, entering_msg);
            CURRENT_STATE= MONITOR;
            oled_clearScreen(OLED_COLOR_BLACK);

        }
        else{
            oled_clearScreen(OLED_COLOR_BLACK);
            oled_putString (0, 35, "Entering Stable", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
            oled_putString (40, 45, "Mode", OLED_COLOR_WHITE, OLED_COLOR_BLACK);
            Timer0_Wait(1000);
            CURRENT_STATE = STABLE;
        }
    }

    }
}
```

Figure 2: Code extract of the function check_SW4()

SECTION 3: DETAILED IMPLEMENTATION

3.1. STABLE STATE

All the operations that need to be performed in the STABLE state are dictated by the function statbleState(). An overview of the stableState() function is illustrated in Figure 3 shown below. The code itself, is shown in Figure 4 below.

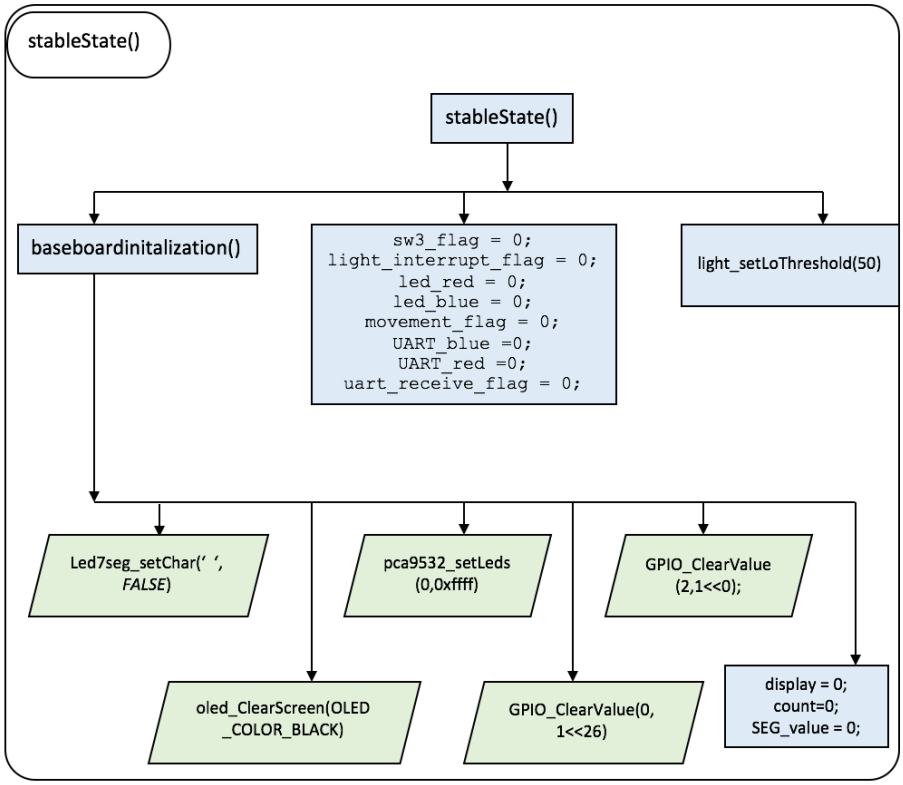


Figure 3: Overview of operations performed in the stableState() function

```
//Operations to perform when the state = STABLE
void stableState(void){
    baseboardinitializaton();
    sw3_flag = 0;
    light_interrupt_flag = 0;
    led_red = 0;
    led_blue = 0;
    movement_flag = 0;
    UART_blue =0;
    UART_red =0;
    light_setLoThreshold(50);
}
```

Figure 4: Code extract of the stableState() function

SECTION 3: DETAILED IMPLEMENTATION

As it can be seen in Figure 4 above, the first operation carried out in the stableState() function is to call **the baseboardinitialization()** method. The main purpose of calling the baseboardinitialization() method is to turn off all the peripherals on the LPC1769 baseboard during the STABLE state. An extract of the code for the baseboardinitialization() method is shown in Figure 5 below.

```
/*function to clear and off everthing before start of program*/
void baseboardinitializaton (void){
        led7seg_setChar(' ', FALSE);
        oled_clearScreen(OLED_COLOR_BLACK);
        pca9532_setLeds(0,0xffff);
        GPIO_ClearValue(0,1<<26);
        GPIO_ClearValue(2,1<<0);
        count=0;
        SEG_value = 0;
}
```

Figure 5: Extract of the baseboardinitialization() code

In this baseboardinitialization() function, library functions of the respective peripherals are used to turn them off. The steps are carried out in the following order:

1. Turn off the 7 Segment LED display, by writing no value to it.
2. Make the OLED display go blank, by clearing the screen with black colour.
3. Turn off all the LEDs in the 16 LED array (i.e. PCA9532 port expander). To do this, the library function pca9532_setLEDs(ledOnMask, ledOffMask) from pca9632.c is used. The ledOnMask is set to 0 as none of the LEDs are supposed to light up. The ledOffMask is set to 0xffff as all the LEDs are supposed to switch off. The hexadecimal number 0xffff when converted to binary, it is equal to 1111 1111 1111 1111, with each bit corresponding to one LED on the 16 LED array. As all the bits are masked to 0, all the LEDs in the 16 LED array turn off.
4. Turn off the BLUE led in the RGB leds. The GPIO_ClearValue function clears value of bits that are initilized with output direction on the GPIO port. Since the bitValue of the BLUE led = 1<<26, the function is written as GPIO_ClearValue(0,1<<26).
5. Turns of the RED led in the RGB leds. Uses the same logic as the BLUE led (described in point 4 above), just that the port of the RED led is 2 and

bitValue of RED led is 1<<0. Hence, the function to turn off the BLUE led is written as GPIO_ClearValue(2,1<<0).

6. The variable *count* is used to contain the number of times the 7-segment LED displays a value (the value can be a number or alphabet). Over here, this variable is initialized as 0 so that the 7 segment LED display starts from 0, the next time the system enters the MONITOR mode. (Further explanation of how the 7 segment display works is present).
7. The vairable *SEG_value* is used to to contain the numerical value displayed on the 7 segment display. Over here, it is set to 0 so that the 7 segment LED display starts from 0, the next time the system enters the MONITOR mode.

After calling the baseboardinitialization() method, the next step is to set all flags to 0 in the stableState() function. A summary of the meaning of each of these flags is shown in Figure 6 below.

Name of the flag	Meaning
sw3_flag	Used to check whether the system is in MONITOR mode. If yes, then sw3_flag=1. If the sw3_flag=1 and also the SW3 interrupt is raised (when SW3 is raised), the CURRENT_DISPLAY is switched from default to help display, or vice-versa depending on the display present when the SW3 is pressed.
light_interrupt_flag	The light_interrupt_flag is set to 1 only when the light interrupt is raised (when the light level is lesser than the minimum light threshold). Used to indicate whether the room is dark.
led_red	Used to indicate if the sampled temperature value is above the maximum threshold for the temperature. Hence, it is used to indicate if fire has occurred.
led_blue	Used to indicate whether movement was detected in darkness.
movement_flag	Used to check whether the difference between the current acceleration and the previously sampled acceleration is greater than 10. If yes, then it denotes that the person is moving and the movement_flag is set to 1.
UART_blue	Same function as led_blue flag. Used to check whether movement in darkness was detected. If yes, then the UART_blue is set to 1. This flag is used instead of the led_blue for UART communication functions.
UART_red	Same function as led_red flag. Used to check whether fire was detected. If yes, then the UART_red is set to 1. This flag is used instead of the led_red for UART communication functions.

Figure 6: Meaning of each flag initialized to 0 in the stableState() function

After that, the light_setLoThreshold(50) is called in order to set the minimum threshhold of the light interrupt to 50. Hence, the importation operations performed in the stableState() function (STABLE mode) are:

1. Calling the *baseboardinitialization()* to turn off all the peripherals
2. Set all flags to 0
3. Reset the minimum threshold of the light interrupt to be 50.

3.2. MONITOR STATE

All the operations to be performed in the MONITOR state are dictated in the function monitorState(). An overview of the monitorState() function is illustrated in the Figure 6 below. The code extract of the monitorState() function is diplayed in Figure 7 below.

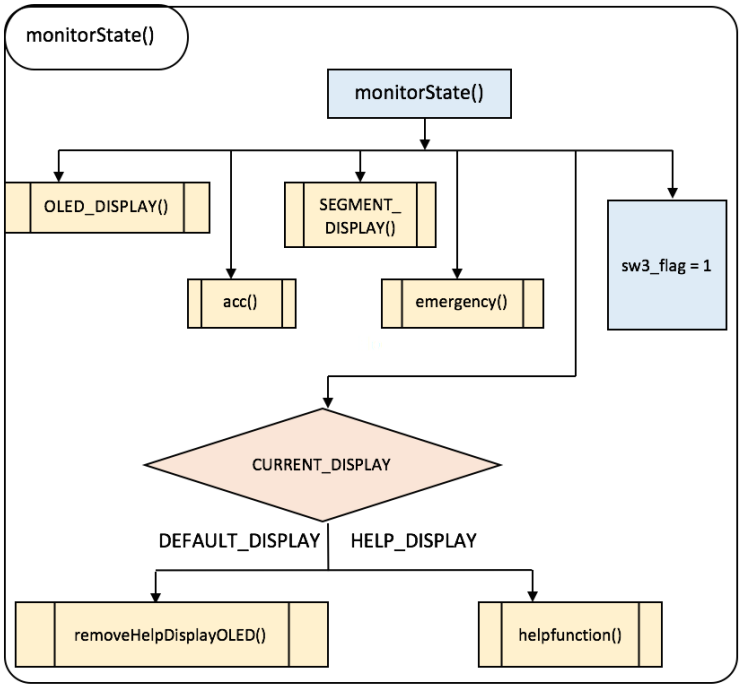


Figure 7: Overview of the monitorState() function

```
//Operations to perform when the state = MONITOR
void monitorState(void){

    OLED_DISPLAY();
    acc();
    SEGMENT_DISPLAY();
    sw3_flag = 1;
    emergency();

    switch(CURRENT_DISPLAY){
    case DEFAULT_DISPLAY:
        removeHelpDisplayOLED();
        break;
    case HELP_DISPLAY:
        helpfunction();
        break;
    default:
        break;
    }


    //light_enable();
    //uartTest();
}
```

Figure 8: Code extract of the monitorState() function

As it can be seen in Figure 6 below, the monitorState() function contains calls to many other functions such as the OLED_DISPLAY(), acc(), SEGMENT_DISPLAY(), emergency(), removeHelpDisplayOLED() and helpfunction(). Each of these functions are assigned for a specific operation that needs to be performed during the MONITOR mode. The operation of each function will be discussed in detail from page 11 onwards.
For now, a summary of the operation of each of these functions are:

- OLED_DISPLAY()
    - Displays “MONITOR” the whole time when the system is in the MONITOR mode
    - Designs the OLED display outlook
    - Collects and writes the temperature value (to 2 decimal places) to the string tempStr (which would be printed on the OLED) and also to the string tempStrUART (which would be printed on the UART terminal).
    - Collects and writes the light value to the string lightStr (which would be printed on the OLED) and also the string lightStrUART (which would be printed on the UART terminal).

  - Sets the flag of led_red to be 1, when the temperature reading is greater than the maximum temperature threshold.
- acc()
  - Calculates acceleration (x, y, z axis values and also the magnitude of acceleration)
  - Checks if movement is detected, if yes, then sets the value of movement_flag to be 1
  - Collects and writes the x-axis value to the string xString; y-axis value to the string yString, z-axis value to the string zString (these values would be printed on the UART).
  - Writes the acceleration value to the string "acceleration" (which would be printed on the OLED display).
- SEGMENT_DISPLAY()
  - Displays values from 0 – F continuously every 1 second, as long as the system is in the MONITOR state.
  - Sends the sampled values of the light, temperature and acceleration readings to the OLED to be displayed on the OLED screen every time the value displayed on the 7 segment is 5, A or F.
  - Calls the uartTest() function to send sampled values to the UART terminal every time the value displayed on the 7 segment is F.
- emergency()
  - Triggers the RBG leds to blink BLUE, RED or PURPLE in respective situations (i.e. movement is detected in darkness, or fire was detected, or both).
  - Sends the message of such emergency situations to the UART terminal.

The sw3_flag is set to 1 to indicate that the system is in MONITOR mode. If the CURRENT_DISPLAY is DEFAULT_DISPLAY, then the function removeHelpDisplayOLED() is called and the "Help" window[2] is not displayed on

[2] The "Help" operation is an enhancement feature that will be explained further in the "ENHANCEMENT FEATURES" section later.

the OLED screen. Likewise, if the CURRENT_DISPLAY is HELP_DISPLAY, then the helpfunction() is called and the “Help” window is displayed on the OLED screen.

3.2.1. OLED_DISPLAY()

An overview of the OLED_DISPLAY() function is shown in Figure 9 below:

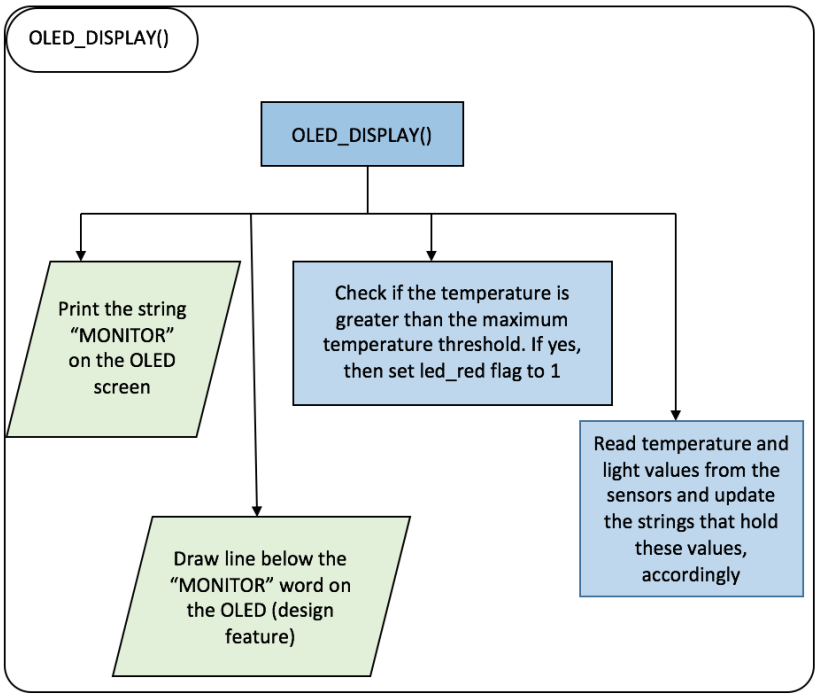


Figure 9: Overview of the OLED_DISPLAY() function

The OLED screen is initialized with a basic template that includes the word “MONITOR” at the first line of the OLED screen, and a white line below this word, to separate it from the sensor readings the are printed below it. Though, the sensor readings are not printed on the OLED through this function (the SEGMENT_DISPLAY() sends the strings to be displayed to the OLED), the strings which contain the values of the temperature and light sensor are updated with the latest sensor readings in this function. This function also checks if the current temperature (sampled at run-time) is greater than the maximum temperature threshold to warn about fire in the surrounding. If indeed the room temperature is greater than the maximum temperature threshold, the led_red flag is set to 1. Setting the led_red flag to 1, will inturn trigger a set of

operations to display warning lights on the OLED and send message to CEMS about the fire detected; this operation will be explained in detail later.

3.2.2. acc()

The acc() performs the acc_read() operation through polling method. The value of the different axis is than passed to 3 variables, namely x, y and z. The x, y and z values are then used to calculate the magnitude. The magnitude is calculated using the formula: sqrt((x*x)+(y*y)+(z*z)). We have made used of magnitude that was previously read with the current magnitude to determine if the elderly have moved. Below is the diagram that explains how the acc() function works.

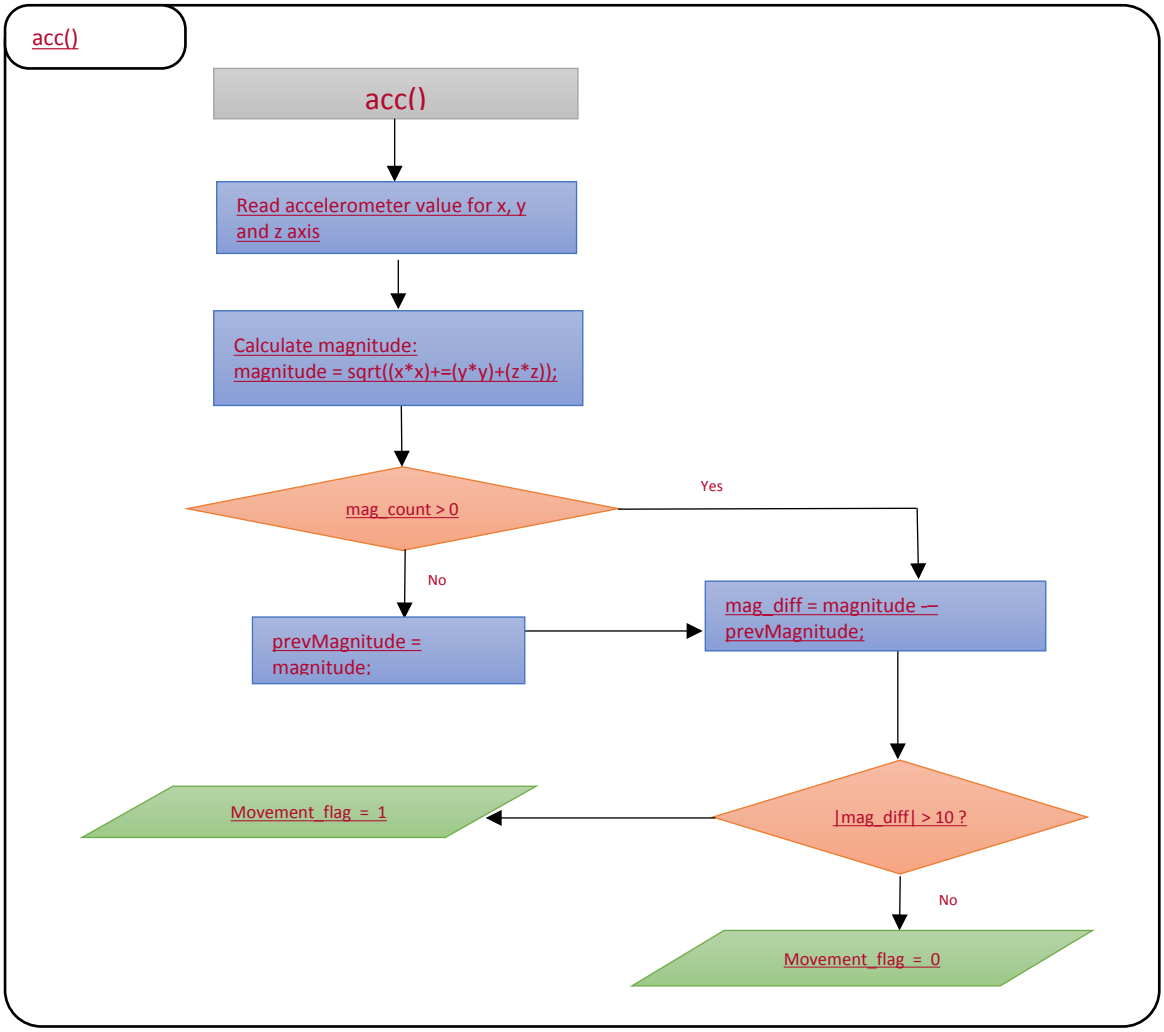


Figure 10: Overview of the acc() function

SECTION 3: DETAILED IMPLEMENTATION

If the difference for the previously read magnitude with the current is more than 10, it is considered that the elderly have moved and the movement flag will be set to 1 otherwise 0. The movement flag will later be used to determine if the elderly is walking in darkness when there's an light interrupt.

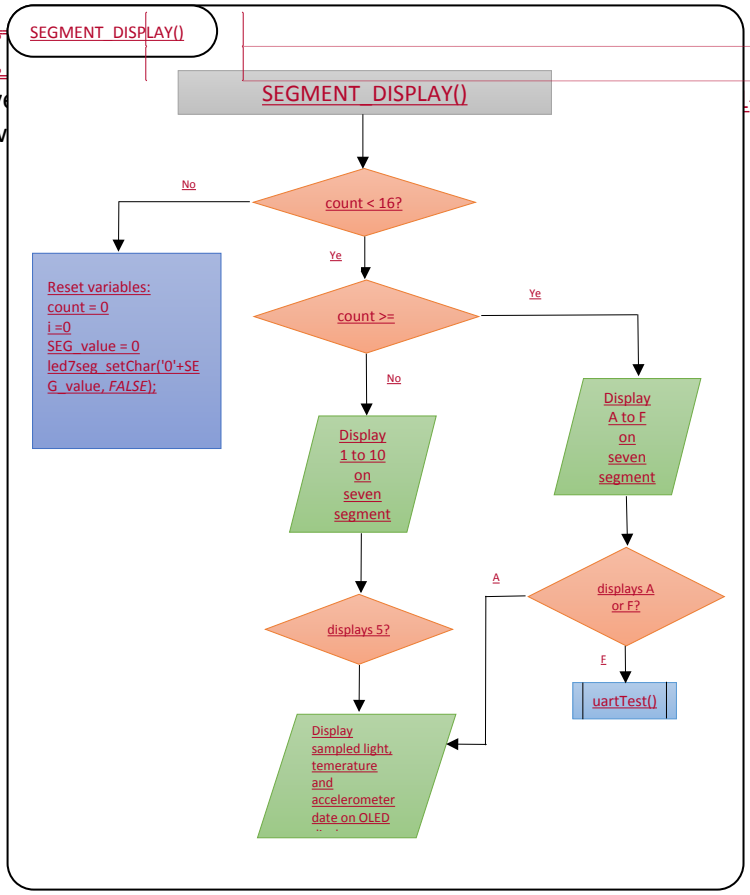
~~//MARINI, PLS COMPLETE THIS~~

~~3.2.3.~~ SEGMENT_DISPLAY()

3.2.3.

An ov~~e~~ ~~0~~

below



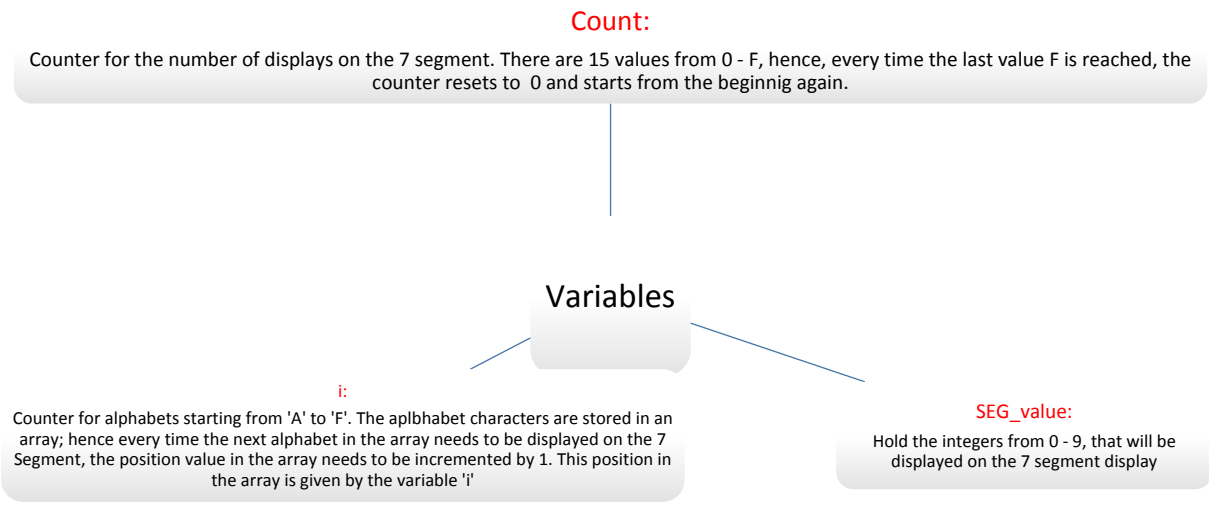
Commented [SSP1]: Marini, your part

Commented [SSP2]: Pls add a flowchart for the 7 segment display.

Commented [SSP3]: Pls add a flowchart for the 7 segment display.

Figure 11: Overview of the SEGMENT_DISPLAY() function

SECTION 3: DETAILED IMPLEMENTATION



The SEGMENT_DISPLAY() makes use of a timer called SEG_timer, which keeps track of the last time the SEGMENT_DISPLAY() function executed. At the very first line, the system first checks if it has been 1 second (= 1000 millisecond) since the last time the SEG_timer was called to execute the SEGMENT_DISPLAY() function. If 1 second has already elapsed, then the system checks if the count is less than 16. This is an important condition because there are only 16 values that need to be displayed on the 7 Segment display (10 numbers, 6 alphabets). Every time a value is printed on the 7 Segment display, the counter increments by 1. When the 16th value (i.e. count = 15) is printed on the 7 Segment display, the count variable resets to 0, and the 1st value has count=0 so it passes the condition count<16 and enters the loop again. Next, the SEGMENT_DISPLAY() function needs to differentiate between the aphabets display and the numerical display, so that it can pass the appropriate values as the parameter in the led7seg_setChar() function. To do this, the next condition in the if-loop, differentiates the numerical values and the alphabetical values – the numerical values count upto 9 (therefore, have $0 \leq count \leq 9$), and the alphabet values count upto 5 (therefore, continuing from the numerical "count" values, which are upto 9, the alphabet

values have $10 \leq count \leq 15$). Hence, if the count<10, numerical values are printed using the function: led7seg_setChar('0'+SEG_value, FALSE), where SEG_value is the integer value that needs to be displayed. After each display, the SEG_value is incremented by 1 so that the next number is printed on the 7 segment in the subsequent call. As according to the basic requirement of the assignment, the sampled values of temperature, light and acceleration need to be displayed on the OLED everytime the display on the 7 segment is 5, there is an additional if-condition inside the count<10 loop to check if the SEG_value = 5, if yes, then oled_putString() function is used to print temperature, light and acceleration values on the OLED display.

If the count>=0, then alphabet values are printed using the function led7seg_setChar(alpha[i], FALSE), where alpha[i] is a character array containing alphabets from A – F. This array, is initalized as a global variable: char alpha[6] = {'A', 'B', 'C', 'D', 'E', 'F'} In order to keep track of which value in the array in displayed on the 7 Segment, another counter named “i” is introduced. Like the counter “count”, “i” is also initalized with 0 in the main function (i.e. everytime the LPC1769 board is first powered on) and everytime the 7 segment has finished printing ‘F’. As according to the basic requirement of the assigment, the sampled values of temperature, light and acceleration need to be displayed on the OLED everytime the 7 segment display shows ‘A’ and ‘F’, there is another if-condition inside the count>=10 loop, to check if the current alphabet being displayed on the 7 segment is ‘A’ or ‘F’. If it is any one of these two alphabets, then oled_putString() function is used to print the sampled temperature, light and acceleration values on the OLED display. Another basic requirement of the assignment specifies that sampled sensor values and emergency messages need to be sent to the UART terminal everytime the 7 segment display shows ‘F’. Hence, an additional if-condition is included inside the loop to check if the current alphabet being displayed on the 7 segment is ‘F’; if yes, then the function uartTest() which handles the UART communication, is called upon.

An extract of the code for the SEGMENT_DISPLAY() function is displayed in Figure 12~~0~~ below.

```
// 7 Segment Display
void SEGMENT_DISPLAY(void)
{

if(checkFlag(&SEG_timer, 1000)){
if(count < 16){
    if(count>=10){
        led7seg_setChar(alpha[i], FALSE);
        if((alpha[i]=='A') || (alpha[i]=='F')){
            oled_putString(18, 20, (uint8_t *) tempStr, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
            oled_putString(18, 30, (uint8_t *) lightStr, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
            oled_putString(18,10, (uint8_t*) accelerometer,OLED_COLOR_WHITE,OLED_COLOR_BLACK);
            if(alpha[i]=='F'){
                uartTest();
            }
        }
        count++;
        i++;
    }
    else{
        count++;
        led7seg_setChar('0'+SEG_value, FALSE);
        if(SEG_value==5){
            oled_putString(18, 20, (uint8_t *) tempStr, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
            oled_putString(18, 30, (uint8_t *) lightStr, OLED_COLOR_WHITE, OLED_COLOR_BLACK);
            oled_putString(18,10,(uint8_t*) accelerometer,OLED_COLOR_WHITE,OLED_COLOR_BLACK);
        }
        SEG_value++;
    }
}
else{
    count = 0;
    i=0;
    SEG_value = 0;
    led7seg_setChar('0'+SEG_value, FALSE);
}

}
return;
}
```

Figure ~~10~~12: Code extract of the SEGMENT_ display() function

3.2.4. emergency() and LIGHT INTERRUPT

Though the light level of the surroundings is regularly sampled by polling, and displayed on the OLED and the UART terminal, the system also makes use of light interrupt to detect when the light level falls below the minimum threshold for light (= 50 Lux). Using interrupt for this purpose is more efficient than using polling at regular time intervals.

Commented [SSP4]: Marini pls complete this section IN your explanation, don't forget to include why you used flags --- I2c is half-duplex protocol, so to ensure that the I2C functions don't clash you use a flag, as you want the function to be called only when the interrupt is raised…etc etc…

SECTION 3: DETAILED IMPLEMENTATION

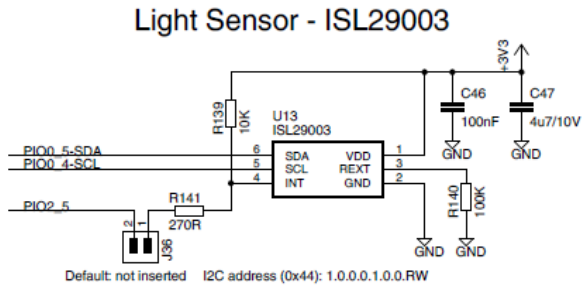


Figure 12: Light sensor schematic

Firstly, we set up the interrupt for the light sensor. According to figure 13 above, INT is connected to PIO2_5 on the baseboard. Then, we had to find out the port connected to the LPC1769. In the user manual, it states that P2.5 is used. Below is the setting for the interrupt for the light sensor:

Figure 13: Code extract of light interrupt

We cleared the interrupt before enabling it, to ensure the current program does not get interfered by the interrupt that was previously raised.

~~3.2.5. uartTest() and calculateCountUART()~~

```
//Light interrupt
light_enable();
LPC_GPIOINT->IO2IntClr |= 1<<5;
LPC_GPIOINT->IO2IntEnF |= 1<<5;
```

Figure 14: Code extract of light interrupt (2)

Commented [SSP5]: Explain blinkRED(), binkBLUE() and blinkPURPLE() functions also in this

SECTION 3: DETAILED IMPLEMENTATION

The low threshold value for the interrupt is set 50 so that there will be an interrupt raised if the light level falls below 50 lux.

```
void EINT3_IRQHandler(void)
{
    light_setRange(LIGHT_RANGE_1000);
    light_setLoThreshold(50); // below this number will interrupt
    light_setHiThreshold(972); // above this number will interrupt
    light_setIrqInCycles(LIGHT_CYCLE_1);
    light_clearIrqStatus();
        LPC_GPIOINT->IO2IntClr |= 1<<5;
        light_clearIrqStatus();
    }
}
```

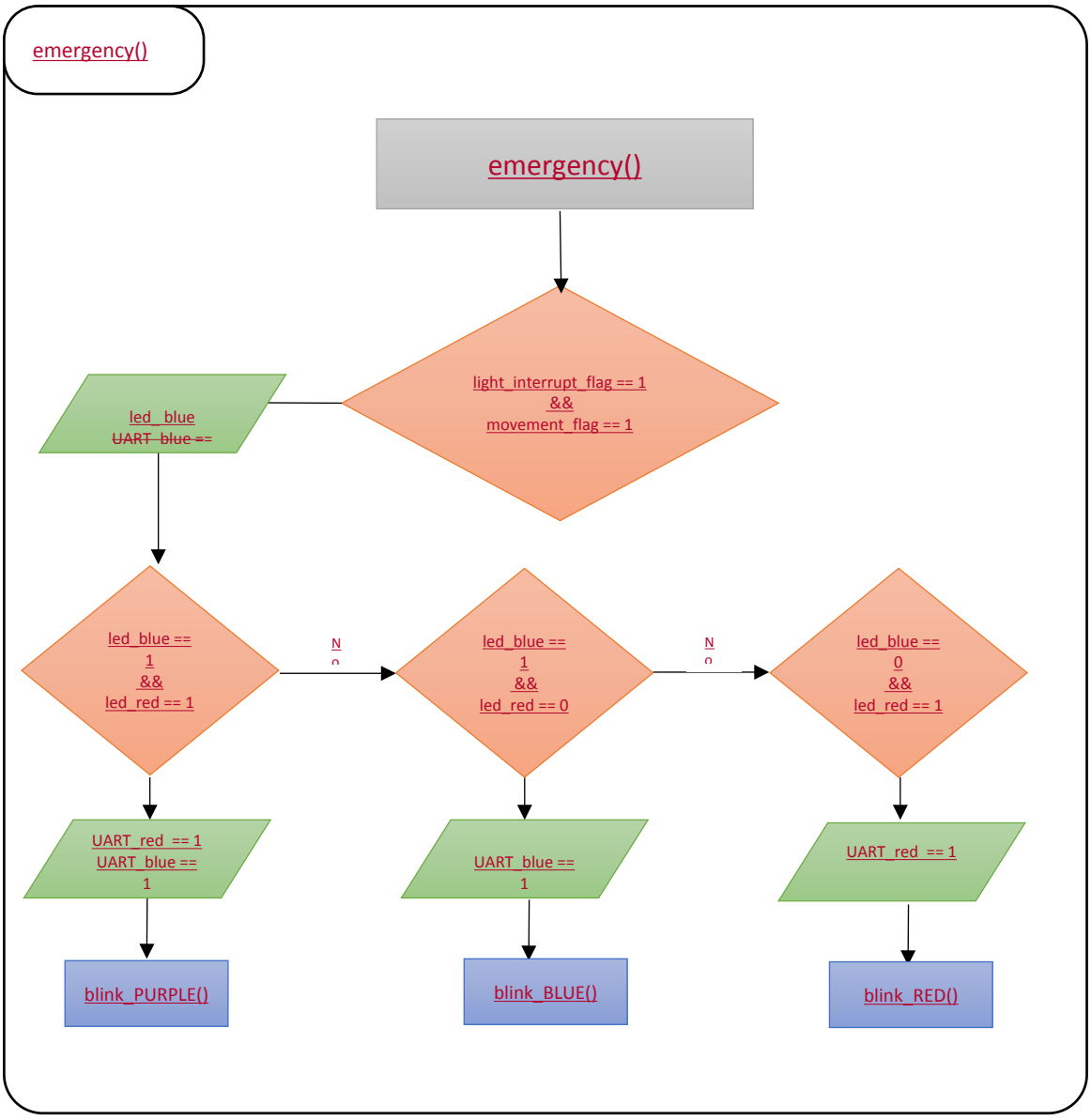
Figure 15: Code extract of light interrupt (2)

When the light interrupt is raised, the light interrupt handler set the light_interrupt_flag to 1. The reason why we did not call the emergency() function here and instead have used the and interrupt flag is because the I2C protocol uses half duplex so to avoid clashing the program by calling the entire function we have set a flag to 1. When flag set to 1 is detected in the while loop, the program calls the function for the light interrupt.

The emergency function is called in the while loop when the program is in MONITOR state. When the light interrupt handler sets the light_interrupt flag to one and when the movement_flag is set to one by acc() function, the emergency() function sets led_blue flag to one. At the same time, emergency() function check if the led_red is set to 1(led_red flag is set to 1 when the temp_sample value is above 450) . After that, the emergency() function calls the blink_PURPLE() function and sets both UART_red flag and UART_blue flag to one when both the led_blue flag and led_red flag has been set to one. When only led_blue flag is 1 then, emergency() function calls blink_BLUE() function and sets UART_blue to 1. When only led_red flag is 1 then, emergency() function calls blink_RED() function and sets UART_red to 1.

SECTION 3: DETAILED IMPLEMENTATION

The figure 16 below is an overview of how the emergency() function works:



The blink_PUPLE() function performs set red and blue led of the RGB led and then clear these two led every 0.33 seconds. This cause the RGB led to blink purple at every 0.33

seconds. The blink_RED() and blink_BLUE() performs the same except that it sets and clear the respective coloured led.

The usage of the UART_blue and the UART_red flag will be explained in the next section(3.2.5).

3.2.5. UART configuration and uartTest() ~~and calculateCountUART()~~

PINSEL0	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P0.0	GPIO Port 0.0	RD1	TXD3	SDA1	00
3:2	P0.1	GPIO Port 0.1	TD1	RXD3	SCL1	00

Figure 17: UART configuration

The initialization of UART uses TXD3 in our LPC1769. To begin we first have to configure the pins ports and function number respectively. Function number is 10 (2 in decimal) is used for TXD3 and P0.0 is concerned. For RXD3 function number is also 10(2 in decimal) and P0.1 is concerned:

```
void pinsel_uart3(void){
    PINSEL_CFG_Type PinCfg;
    PinCfg.Funcnum = 2;
    PinCfg.Pinnum = 0;
    PinCfg.Portnum = 0;
    PINSEL_ConfigPin(&PinCfg);
    PinCfg.Pinnum = 1;
    PINSEL_ConfigPin(&PinCfg);
}
```

Figure 18: UART configuration (2)

After initializing, we have to set the baud rate, data bits, parity bits and stop bits respectively. Shown below in figure 19 is how we set them:

SECTION 3: DETAILED IMPLEMENTATION

```
void init_uart(void){
    UART_CFG_Type uartCfg;
    uartCfg.Baud_rate = 115200;
    uartCfg.Databits = UART_DATABIT_8;
    uartCfg.Parity = UART_PARITY_NONE;
    uartCfg.Stopbits = UART_STOPBIT_1;
    //pin select for uart3;
    pinsel_uart3();

    //supply power & setup working parameters for uart3
    UART_Init(LPC_UART3, &uartCfg);
    //enable transmit for uart3
    UART_TxCmd(LPC_UART3, ENABLE);
}
```

Figure 19: UART configuration (3)

The uartTest() function makes use of the flag that the emergency() function sets which I explained in the previous section. When the UART_blue flag is set to one "Movement in darkness was detected" message will be sent to terra term. When the UART_red flag is set to one "Fire was detected" message will be sent to terra term. When both the flags are set to one, both respective UART messages will be sent to terra term. Moreover, the uartTest() function also sends the samples values of the light, temperature and accelerometer as an UART message to the terra term. This function is called in the SEGMENT_DISPLAY() function when the seven segment display displays F.

```
//UART test code
void uartTest(void){
        sprintf(emer_msg_blue, "Movement in darkness was Detected.");
        strcat(emer_msg_blue, "\r\n");
        sprintf(emer_msg_red, "Fire was Detected.");
        strcat(emer_msg_red, "\r\n");

        if((UART_blue == 1) && (UART_red==0)){
            UART_SendString(LPC_UART3, emer_msg_blue);
            //UART_SendString(LPC_UART3, help_msg);
            //strcpy(countUART, " ");
        }
        else if((UART_red == 1) && (UART_blue==0)){
            UART_SendString(LPC_UART3,emer_msg_red) ;
            //UART_SendString(LPC_UART3, help_msg);
            //strcpy(countUART, " ");
        }
        else if ((UART_red == 1) && (UART_blue==1)){
            UART_SendString(LPC_UART3,emer_msg_red) ;
            UART_SendString(LPC_UART3, emer_msg_blue);
            //UART_SendString(LPC_UART3, help_msg);
        }

        calculateCountUART();
        sprintf(msg, "%s_-_T%s_L%s_AX%s_AY%s_AZ%s",countUART, tempStrUART,
                lightStrUART, xString, yString, zString);
        strcat(msg, "\r\n");
        UART_SendString(LPC_UART3, msg);
        strcpy(countUART, " ");
}
```

Figure 20: Code extract of light interrupt (2)

SECTION 4: ENHANCEMENT FEATURES

4.1. Rotary Switch Interrupt

Rationale: Each elderly person may have different sensititvity towards light or may be comfortable with different light levels while doing different activities (for example, they may need only dim light while watching TV, bright light while reading, no light while sleeping, etc). Hence, the system has an **ambient light feature**. The eldely can adjust the light levels in their surroundings using the rotary switch. Rotating the rotary switch in the clockwise direction would gradually increase the level of brighness of the 16 array LEDs in the pca9532 port expander. Similarly, rotating the rotary switch in the anticlockwise direction would gradually decrease the level of brightness of the same 8 LEDs in the pca9532 port expander. They can stop the rotary switch at any point they want when the 8 LEDs light at their desired brightness level.

Implementation: When the rotary switch is turned, the rotary switch SW5 interrupt is triggered. When the interrupt is triggered, the interrupt handler function sets the rotary flag to 1, by calling the function setRotaryFlag(1), where the input parameter is the rotary flag. As the rotary flag is set to 1, the system enters the setRotaryFlag() function and executes the series of actions specified in this function. The flow of actions that follow when the setRotaryFlag(1) function is called can be seen in Figure 21~~12~~ below.

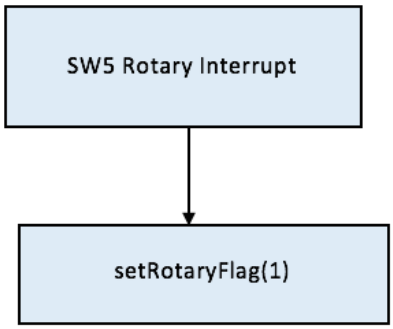


Figure ~~21~~2111: Action performed when the rotary interrupt is triggered

SECTION 4: ENHANCEMENT FEATURES

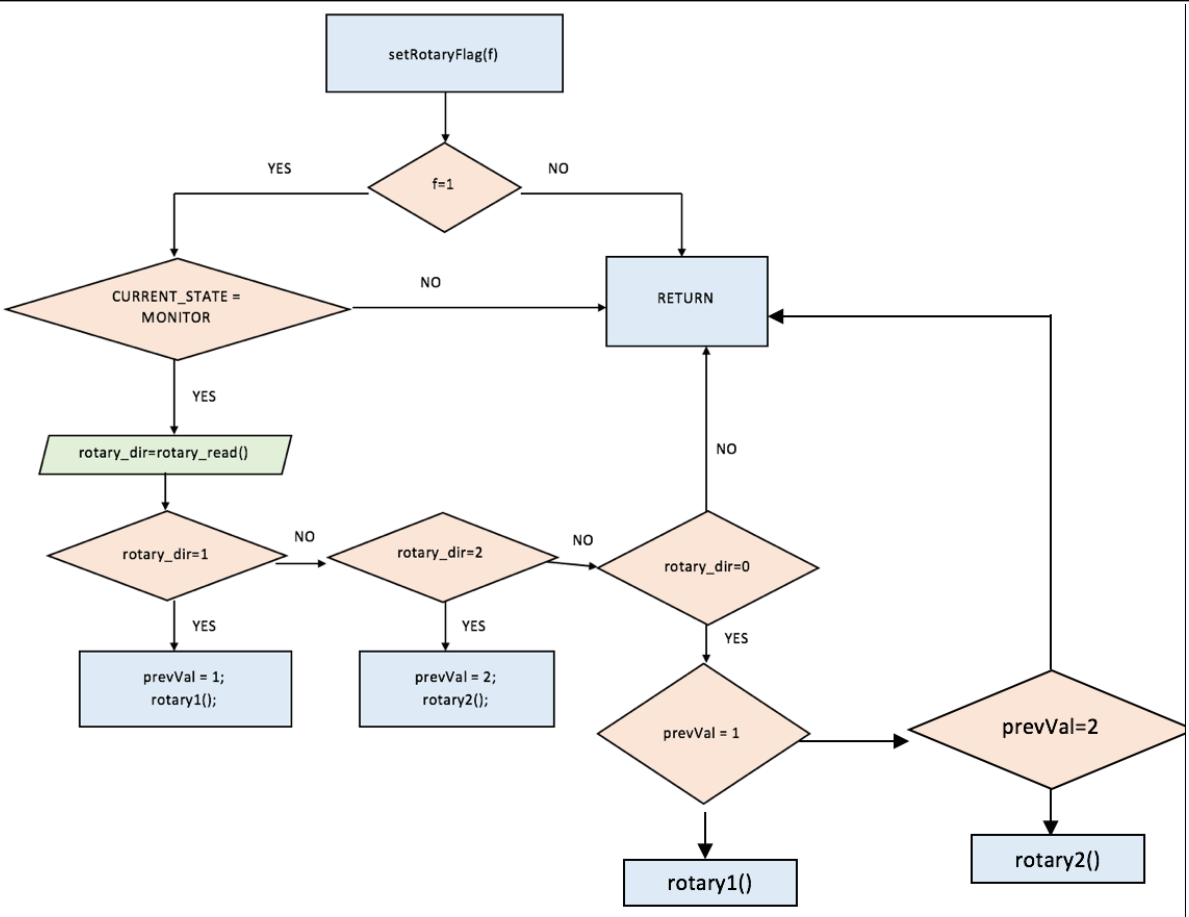


Figure 10: Overview of setRotatyFlag(f) function

As it can be seen in figure 2~~1~~2 above, the sequence of steps that follow when the setRotaryFlag(1) function is called in the interrupt handler are:

1. Checks if the system is in MONITOR state. If yes, then it collects the input from the rotary switch using the rotary_read() library function. The data type of rotary_dir is uint32_t, hence, the output of the rotary_read() function is stored as an integer in the rotary_dir variable.
2. The direction of the rotary (specified by the rotary_dir variable) can be either 1 (clockwise), 2 (anti-clockwise) or 0 (intermediate state).

SECTION 4: ENHANCEMENT FEATURES

3. If the direction is clockwise, i.e. rotary_dir = 1, then the variable prevVal is set to 1 and the function rotary1() is executed. The variable prevVal stores the value of the previous direction of the rotary, so that when rotary_dir = 0, this previous direction can be taken as the current direction and its operations can be executed[3].

[3] Addition of prevVal is an enhancement to the rotary switch reading, aimed to improve the accuracy and smooth execution of operations depended on the rotary switch reading.

4. Likewise, if the direction is anti-clockwise, i.e. rotary_dir = 2, then the variable prevVal is set to 2 and the function rotary2() is executed.
5. In events of systematic error, when the rotary switch reads 0 instead of 1 or 2 (which are theoretically the only 2 possible outputs of the rotary switch direction), then based on the previous value read (indicated by the prevVal value), the appropriate rotary function (i.e. rotary1() or rotary2()) is executed. For example, if the prevVal = 1 and the rotary_dir = 0, then rotary1() will be executed, and vice-versa. This ensures a smooth flow of inputs from the rotary switch.

It can be seen that the setRotaryFlag() function simply assigns the system to carry out either rotary1() or rotary2() based on the direction of the rotary switch read from the library function rotary_read(). The actual operations of each direction is performed in rotary1() and rotary2() fucntions, depending on the direction read. An overview of these 2 functions is shown in Figure 13 (for rotary1()) and Figure 214 (for rotary2()) below:

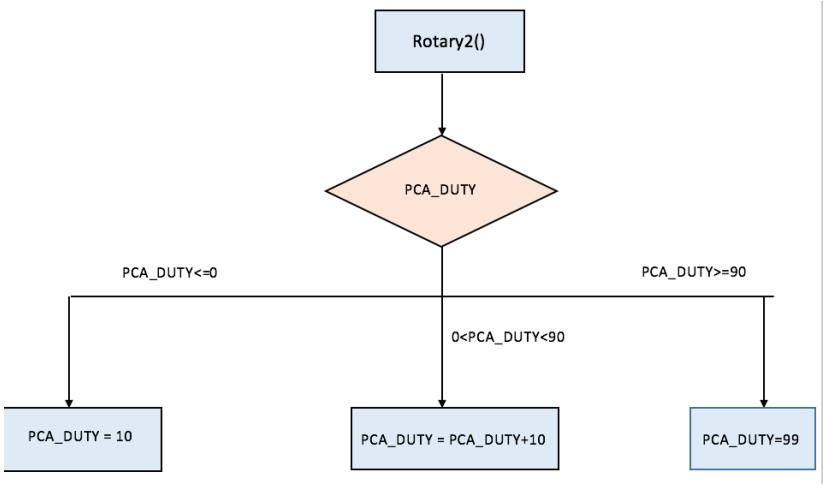


Figure 11: Overview of rotary1() function

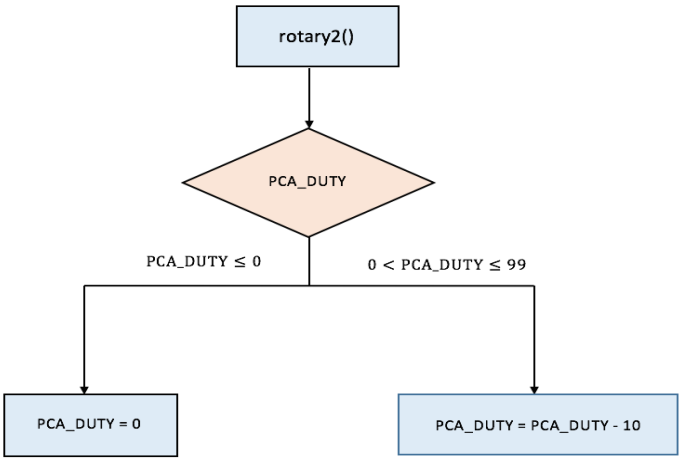


Figure 12: Overview of rotary2() function

As it can be seen in Figure 213 and Figure 214 above, the direction in which the rotary switch is turned by the user controls the PCA_DUTY of the 16 array LEDs. The PCA_DUTY is the variable which holds the value of duty cycle of the 16 array LEDs. Hence, as the brightness of the 16 array LEDs is controlled by their duty cycle, the duty cycle (represented by the variable PCA_DUTY) is altered according to the way in which the rotary switch is turned by the user. Larger the value of the duty cycle (i.e. larger PCA_DUTY), higher the brightness level of the 16 array of LEDs. Hence, when the rotary switch is turned clockwise, the system executes rotary1() function, where the PCA_DUTY is increased gradually (by 10 steps every time[4]) until the maximum brightness is reached (i.e. until PCA_DUTY = 99). ~~Similary~~Similarly, when the rotary switch is turned anticlockwise, the system executes rotary2() function, where the PCA_DUTY is reduced gradually (by 10 steps every time) until the minimum brightness level is reached (i.e. until PCA_DUTY = 0). Hence, the rotary switch performs the operation of varying the duty cycle of the 16 array LEDs. Further explanation of how varying the duty cycle affects the brightness level of the 16 array LEDs will be provided in the next section (i.e. section 4.2.).

4.2. Ambient Lighting – Using pca9532 Port Expander

Rationale: Same as that for Section 4.3.

[4] The increment factor for the PCA_DUTY was chosen to be 10 as the change in the level of brightness for change in PCA_DUTY less than 10 was too small to be noticed, and the change in the level of brightness for change in PCA_DUTY greater than 10 was quite drastic. Hence, 10 was chosen as the increment factor as it is neither too small nor too large, and allows the elderly to have a greater choice of comfortable lighting level.

Implementation: The interrupt handler of the rotary switch and functions associated with it vary the PCA_DUTY (which holds the value for the duty cycle of the 16 LEDs), according to the direction in which the rotary switch is turned. Theoretically, the duty cycle is defined as the amount ~~amount~~ of time the LED is in ON state (i.e. power is supplied to the LED) over a duration of 1 time period. So, if the duty cycle is 50%, the LED is ON only for 50% of the time in 1 cycle, hence, its brightness is only 50% of its maximum brightness level. The library for pca9532 port expander has a function called pca9532_setBlink0Duty(duty) which sets the duty cycle of the 16 LEDs in the pca9532 port expander. CUTE makes use of this library function to vary the duty cycle of the 16 LEDs array according to the value of the PCA_DUTY, which is in turn, controlled by the rotation of the rotary switch. Figure 25~~16~~ below shows the code extract for setting the duty cycle of the 16 LEDs array.

```
void setLEDs(void){
    pca9532_setBlink0Duty(PCA_DUTY);
    pca9532_setBlink0Leds(0xFF00);
}
```

Figure 13: Setting the duty cycle of the 16 LEDs

Since we require only the first column (i.e. 8 LEDs) out of the 16 LEDs to light up, we turn off the remaining 8 LEDs in the 16 LEDs array using the pca9532_setBlink0Leds(0xFF00[5]) function.

4.3. SW3 Interrupt

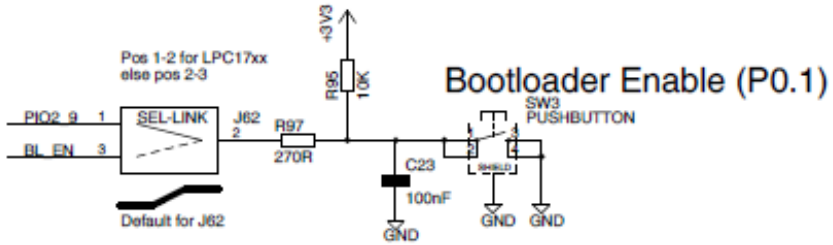


Figure26: SW3 configuration

[5] The binary equivalent of 0xFF00 is 1111 1111 0000 0000. Hence, we can see that the last 8 LEDs (corresponding to the last 8 bits of the binary value) are masked off. This is how the last 8 LEDs in the 16 LEDs array are turned off.

SECTION 4: ENHANCEMENT FEATURES

For the pushbutton switch 3 in Figure 3, in the baseboard we have set our jumper to BL_EN, which corresponds to P0.4 on the LPC 1769.

```
//SW3 Interrupt
LPC_GPIOINT->IO2IntClr |= 1 << 10; //SW3
LPC_GPIOINT->IO2IntEnF |= 1 << 10; //switch
```

Figure27: SW3 Interrupt

SW3 uses GPIO interrupt. So we cleared the interrupt, before raising another interrupt. This is done in the main program.

```
//SW3 interrupt
if ((LPC_GPIOINT->IO2IntStatF>>10)& 0x1)
{
    if(sw3_flag == 1){
        if(CURRENT_DISPLAY== DEFAULT_DISPLAY){
            CURRENT_DISPLAY= HELP_DISPLAY;

        }
        else
            CURRENT_DISPLAY= DEFAULT_DISPLAY;
    }
    sw3_flag == 0;

    // Clear GPIO Interrupt P2.10
    LPC_GPIOINT->IO2IntClr |= 1<<10;
}
```

Figure28: SW3 Interrupt handler

In the interrupt handler, the SW3 switches to help display state if it was in default display state. And switches to default display state if it was in help display state.

4.4. Joystick Interrupt

```
//joystick interrupt for left and right direction
if (((LPC_GPIOINT->IO2IntStatF>>4)&0x1) |((LPC_GPIOINT->IO0IntStatF>>16)&0x1))
{
    if(joystick_flag ==1){
        help_joystick();
    }
        joystick_flag = 0;
        LPC_GPIOINT->IO0IntClr |= 1<<16;
        LPC_GPIOINT->IO2IntClr |= 1<<4;

}
```

Figure29: Joystick Interrupt handler

SECTION 4: ENHANCEMENT FEATURES

When the Joystick (left and right) interrupt is triggered, the joystick handler calls the help_joystick() function. Below is the diagram for how the help_joystick() works. Joystick interrupt only works if the CURRENT_DISPLAY is in HELP_DISPLAY state.

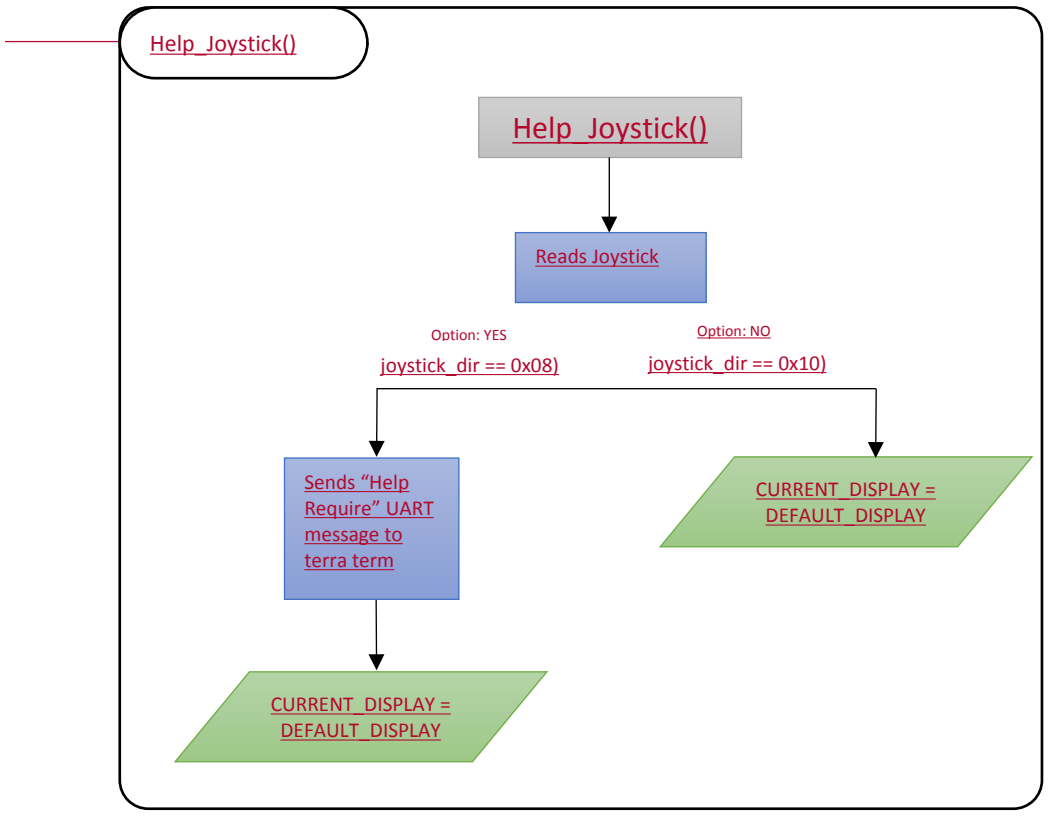
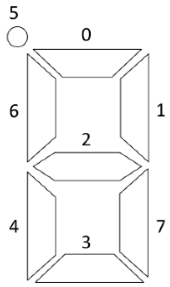


Figure30: Overview of Help_Jostick() Funtion

SECTION 4: ENHANCEMENT FEATURES

4.5. 7 Segment Display Inversion

This is the image of the 7 segment LED display on our baseboard. We have noticed that it has been soldered on in the reverse way and so conventional method of lighting the 7 segment LED display would result in the numbers being inverted. Therefore, we went to change the code in the '7seg.c' file to get the desired outcome.



In the led7eg.c, the array chars [] contains all the different numbers and alphabets. From the 4th element to the 13th element stores the digit 0 – 9. However previously the hexadecimal equivalent corresponds to the numbers being inverted. To find out which bit corresponds to which segment, we first have to use different hexadecimal to light up the 7 segment. For the first try we replaced the number 0 with binary 11111110(0xFE) and the segment 0 lights up shown above in the diagram. We do this for all the characters and finally changed all the numbers accordingly.

~~4.6. OLED display~~

SECTION 5: PROBLEMS ENCOUNTERED AND SOLUTIONS PROPOSED

Initially, the program freezes every time an interrupt is detected. We then realised that calling a function in the interrupt handler causes the program to clash. In order to tackle this problem we created flag variables for each interrupt (SW3, Joystick, Rotary and Light). Whenever an interrupt is raised, it will set the flag that is associated to it to one. The flag variable then calls the function that has to run when that particular interrupt has been raised. Through this method, the program no longer freezes when an interrupt is raised.

SECTION 6: IMPROVEMENT SUGGESTIONS

It would have been more efficient if we had implement the temperature interrupt. The timer interrupt could be used to raise an interrupt whenever the temperature reading raises above 45 degree Celsius. This method would have been more efficient than the polling method that we used in our program.

SECTION 7: CONCLUSION

Through this project we learnt to put the theories learnt in lectures and tutorials into practice as we kept referring to the lecture materials and this gave us an in depth meaning to the concepts we were learning. We started off by experimenting with the different helper functions and then used them asWe also learnt how to read the manual on our own and implement different protocols and features. When we further enhance the program we stick to the main purpose of the system when we came up with our own enhance feature and made sure these features are beneficial to the elderlies.