UNIVERSITY OF LJUBLJANA | FRI Faculty of Computer and Information Science

Temperature alarm

Project report for the course Input-Output Systems

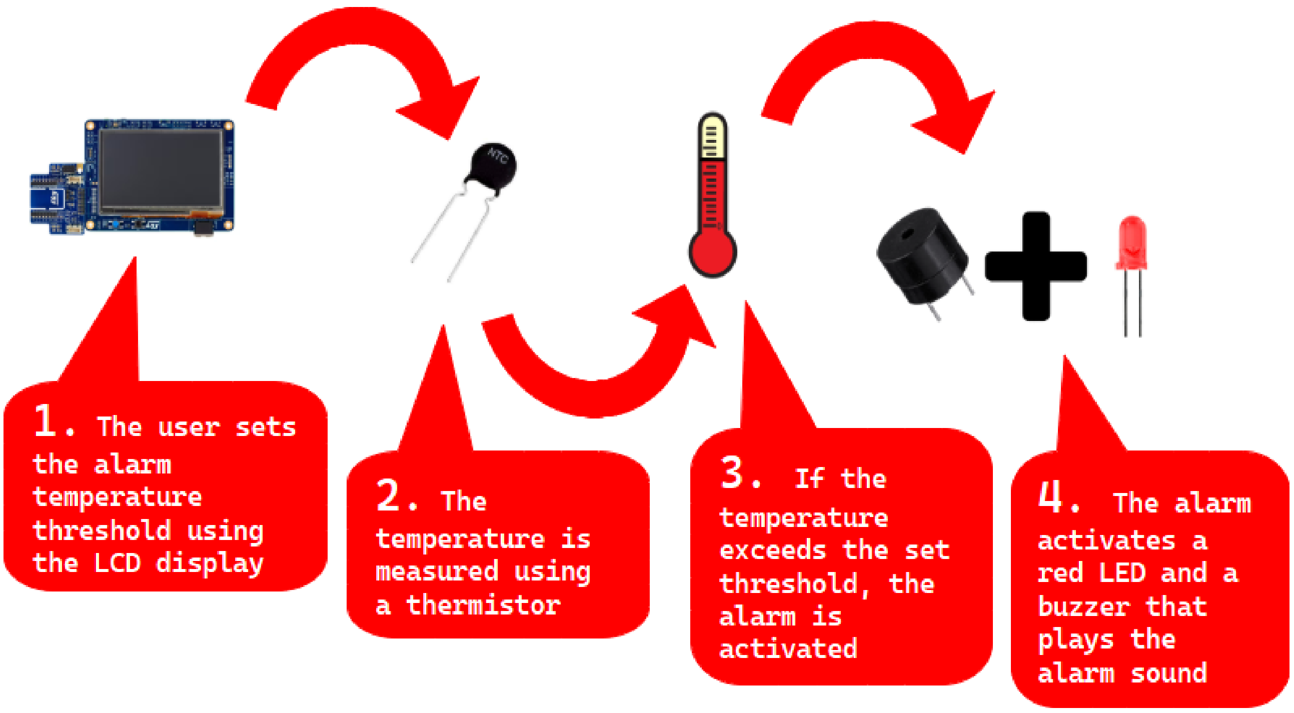
Author: Nik Uljarević, 63220345

Mentor: Sen. Lect. Dr. Robert Rozman

Table of Contents

Introduction

In this report, I will provide a detailed explanation of my project for the STM32H750B-DK microcontroller. The project involves using the STM32 board, a thermistor, a red LED and a buzzer to create a temperature alarm system. Here's a brief summary of how it works: The STM32 board sends a constant voltage to the thermistor. The board then converts the voltage passing through the thermistor to a digital signal using an ADC. This measured voltage is used to calculate the thermistor's resistance, which is then converted to Celsius using a resistance-to-temperature look-up table. If the temperature exceeds the alarm threshold set by the user via the touchscreen, a PWM signal alternating between two notes is sent to the buzzer to produce the alarm sound. Additionally, a high voltage signal is sent to the red LED to turn it on. The LCD screen displays both the current temperature and the user-set alarm temperature.


1. The user sets the alarm temperature threshold using the LCD display
2. The temperature is measured using a thermistor
3. If the temperature exceeds the set threshold, the alarm is activated
4. The alarm activates a red LED and a buzzer that plays the alarm sound

Parts list

- 1x STM32H750B-DK
- 1x STMod+ connector
- 1x Breadboard
- 1x Red LED
- 1x Buzzer
- 1x NTC 103 Thermistor B value: 3950K@25
- 1x 10 Ohm resistor
- 1x 220 Ohm resistor
- 1x 10k Ohm resistor
- 8x Connector cable

This project uses the STM32H750B-DK and the STMod+ connector for easy integration with the breadboard. A red LED and a buzzer will serve as the alarm indicators. The thermistor will be used to measure temperature. A 10 Ohm and a 220 Ohm resistor are used to reduce the voltage output to the buzzer and LED light. A 10k Ohm resistor is used in the voltage divider circuit in conjunction with the thermistor. Connector cables are used to connect the pins from the STM32 microcontroller to the breadboard

Project setup

In this section, I will explain how to initialize and configure each of the pins required for this project. Since this project is based on the STM32H750-DK_BSP_Touch_Demo, I will not cover changes already made in that project, such as uncommenting lines in the stm32h7xx_hal_conf.h file or adding the BSP library, which is foundational to this project.

Since the STM32H750-DK_BSP_Touch_Demo project was not created using STM32CubeIDE or STM32CubeMX, it does not include an .ioc file. This means that all pins must be configured and initialized manually in the code. To simplify this process, I have adopted a different approach for initializing the pins

1. Create a new project using STM32CubeIDE (or STM32CubeMX)

   1.1 With STM32CubeIDE opened go to File → New → STM32 Project

   1.2 Under board selector choose the STM32H750B-DK board and click Next

   1.3 Name the project (this project will only be used for code generation, so the name is not important) and click Finish

   1.4 When asked “Initialize all peripherals with their default mode” choose No

1.5 Go to Pinout & Configuration -> Pinout → Clear Pinouts, this will set all the pins to Reset_State

1.6 Go to Project Manager → Code Generator → Tick the "Generate peripheral initialization as a pair of '.c/.h' files per peripheral

2. Set the pins

2.1 Setup the timer that will generate the PWM signal for the buzzer

2.1.1 Pinout & Configuration → TIM2 → Channel4 → PWM Generation CH4

2.1.2 Set the PA3 pin to TIM2_CH4

2.1.3 Under Parameter Settings, set the Prescaler to 199. This will effectively divide the APB1 Timer Clock speed used by TIM2, which is set to 200 MHz, by 200, resulting in an effective speed of 1 MHz

2.2 Setup the ADC that will be used to read the thermistor's resistance

2.2.1 Pinout & Configuration → ADC1 → IN18 → IN18 Single-ended

2.2.2 Set the PA4 pin to ADC1_INP18

2.3 Setup the UART that will transmit information to the PC

2.3.1 Pinout & Configuration → USART3 → Mode → Asynchronous

2.3.2 Set the PB10 pin to USART3_TX

2.3.3 Set the PB11 pin to USART3_RX

2.3.4 Under Parameter Settings, set Data Direction to Transmit Only, as we will be using only the transmit function

2.4 Set up the output pin for controlling the high and low signals to the LED alarm light

2.4.1 Set the PH10 pin to GPIO_Output

2.5 Save the project

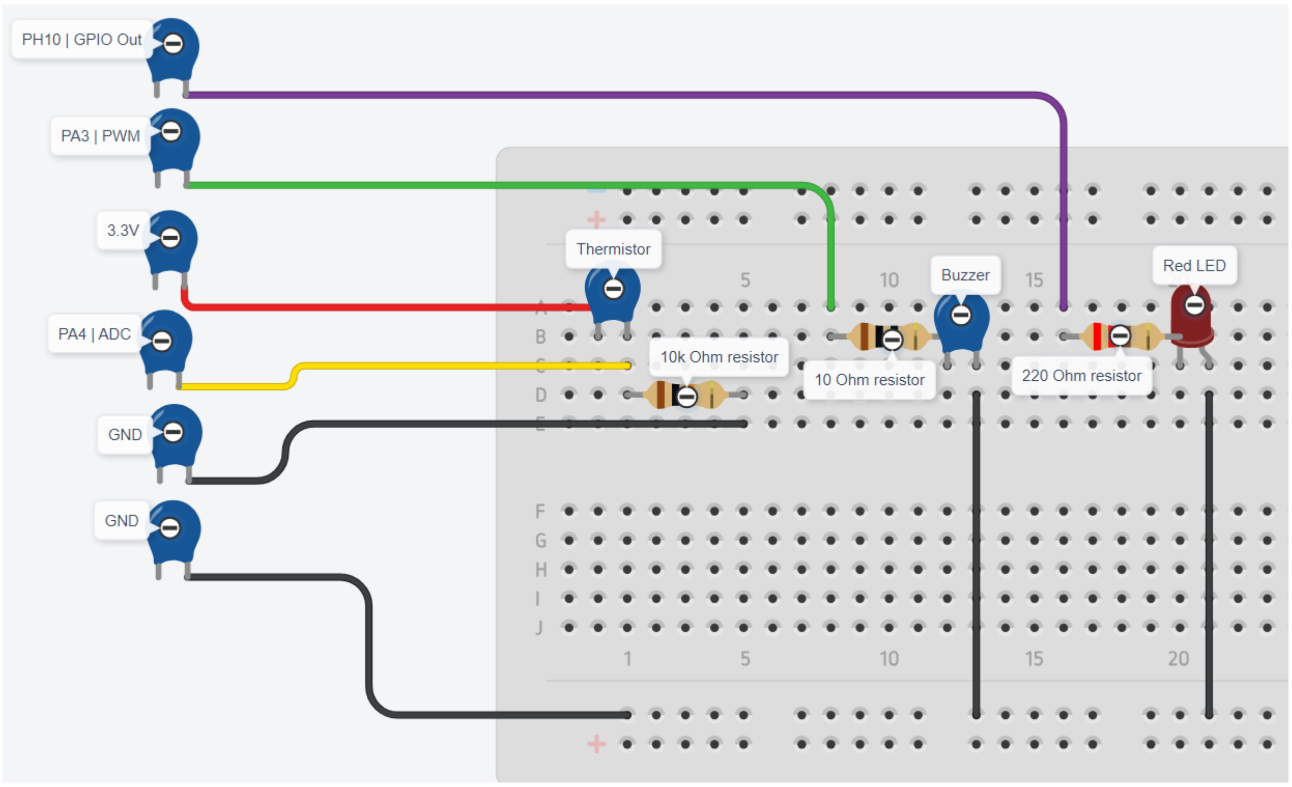
3. Copy and paste the .c and .h files generated by the .ioc to the STM32H750-DK_BSP_Touch_Demo project

3.1 Copy and paste adc.c, gpio.c, tim.c and usart.c to STM32H750-DK_BSP_Touch_Demo project folder → Core → Src

3.2 Copy and paste adc.h, gpio.h, tim.h and usart.h to STM32H750-DK_BSP_Touch_Demo project folder → Core → Inc

We will use the functions defined in these files to initialize the pins and peripherals, rather than writing the code manually.

Circuit explanation



The above picture displays the circuit and its connections to the STM32 board. I used capacitors as substitutes for the board pins, thermistor, and buzzer, but I have clearly labeled everything. I had to use capacitors because Tinkercad, the website I used to create this circuit, does not have these specific components.

As you can see, the thermistor and the 10k Ohm resistor function as a voltage divider. This voltage divider converts changes in resistance into proportional changes in voltage, which can then be easily measured by an ADC. We then calculate the resistance using the following formula:

$$R_{NTC}=R_{fix}\times\left(\frac{V_i}{V_o}-1\right)$$

R_{ntc} represents the resistance of the thermistor, R_{fix} represent the resistance of the 10k resistor and V_i and V_o represent the input and output voltage. The part we measure with the ADC is, of course, V_i.

If the measured temperature exceeds the alarm limit set by the user via the touchscreen, the STM32 board sends a PWM signal to the buzzer to produce the alarm sound and a high voltage signal through the GPIO output pin to turn on the red LED.

The thermistor is connected to a separate ground to prevent the PWM signal from affecting the temperature measurement. Alternatively, a diode could have been used instead of a separate ground.

Code explanation

In this section, I will walk through the code step by step and explain the key parts that may require further clarification.

Main.c

```c
MX_GPIO_Init();
MX_ADC1_Init();
MX_USART3_UART_Init();
MX_TIM2_Init();

BSP_LCD_Init(0, LCD_ORIENTATION_LANDSCAPE);
UTIL_LCD_SetFuncDriver(&LCD_Driver);

// Initializes the static features of the LCD GUI
Touchscreen_template_init();
// Draws the default alarm temperature (35)
Draw_threshold(alarm_temp);
```

The first four functions displayed above were created using the .ioc file, as described in the Project Setup section of this report. They configure and initialize all the necessary peripherals and pins. BSP_LCD_Init and UTIL_LCD_SetFuncDriver are used to initialize the LCD screen. Touchscreen_template_init initializes the touchscreen and draws the static parts of the GUI on the LCD display. Draw_threshold displays the alarm temperature threshold, which is currently set to its default value of 35, between the two buttons.

```c
 while (1)
 {
  // Is true every 20ms
  if (flag_20ms) {
    new_state = Button_check();
    if (current_state == new_state) {
          hold_counter++;
    } else {
          hold_counter = 0;
          hold_speed = 10;
    }

    if (new_state == BUTTON_PRESSED_UP && alarm_temp < 130 && (!hold_counter || hold_speed == hold_counter)) {
          hold_counter = 0;
          if (hold_speed > 4) {
                hold_speed--;
          }
          Draw_threshold(++alarm_temp);
    } else if (new_state == BUTTON_PRESSED_DOWN && alarm_temp > -30 && (!hold_counter || hold_speed == hold_counter)) {
          hold_counter = 0;
          if (hold_speed > 4) {
                hold_speed--;
          }
          Draw_threshold(--alarm_temp);
    }

    if (new_state != current_state) {
          Draw_buttons(new_state, current_state);
          current_state = new_state;
    }
    flag_20ms = 0;
  }
```

The first part of the while loop is executed when flag_20ms is non-zero, which occurs every 20 milliseconds. This section of the code handles the buttons on the touchscreen. It first checks if a button is being pressed. If a button is pressed and either no button or a different button was previously pressed, it adjusts the alarm temperature threshold by increasing or decreasing it, depending on which button is pressed. If the user continues to hold the same button, the speed at which the threshold increases or decreases gradually accelerates, providing a smoother user experience. Additionally, if the button state changes from the previous 20 ms cycle, the buttons that need to be updated are redrawn by calling the Draw_buttons function. This is necessary because the color of the pressed button changes.

```c
        // Is true every second
        if (flag_1s) {
            // Gets value from ADC
            HAL_ADC_Start(&hadc1);
            HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
            analogValue = HAL_ADC_GetValue(&hadc1);

            // Calculates ADC input voltage
            VOut = (double)analogValue / 65535.0f * 3.3;

            VOut_int = (int)VOut;
            VOut_frac = (int)((VOut - VOut_int) * 10000);

            // Calculates thermistor resistance
            res = 100000 * ((65535.0 / (double)analogValue) - 1);

            res_int = (int)res;
            res_frac = (int)((res - res_int) * 100);

            // Converts resistance to Celsius using LUT
            temps = Binary_search(res);

            // Interpolates temperature for more accurate result
            temp = Linear_interpolation(temps, res);

            temp_int = (int)temp;
            temp_frac = (int)((temp - temp_int) * 100);

            // Draws current temperature on LCD
            Draw_temperature(temp_int, temp_frac);
            // Handles alarm activation, deactivation and the changing of the note being played
            Alarm((int)alarm_temp, temp);

            snprintf(sendBuffer, BUFFSIZE, "[%d] analog: %d | VOut: %d.%04d | res: %d.%02d | temp: %d.%02d\n\r",
                        counter++, analogValue, VOut_int, VOut_frac, res_int, res_frac, temp_int, temp_frac);
            HAL_UART_Transmit(&huart3, (uint8_t *)sendBuffer, strlen(sendBuffer), 100);

            flag_1s = 0;
        }
    }
```

The second part of the while loop is executed every second. It handles the conversion of the ADC input value to Celsius, alarm activation and transmission of information to the connected PC via UART, which was mainly used for debugging but remains included for potential user benefit. It starts by reading the value from the ADC, then converts this value to voltage and calculates the thermistor’s resistance using the formula mentioned in the Circuit explanation section of this report. The resistance is then converted to Celsius using a binary search algorithm on a resistance-to-Celsius look-up table (LUT) for the thermistor, with a step size of 1 degree Celsius, by calling the Binary_search function. The Linear_interpolation function is then called to interpolate the temperature between two steps of the LUT. Next, the Draw_temperature function is called to display the calculated temperature on the LCD screen. The Alarm function is then invoked to activate or deactivate the alarm, depending on whether the temperature exceeds the set threshold. Finally, the debugging information is transmitted to the PC via UART.

```c
// Handles the 20ms and 1s flags
void HAL_IncTick(void)
{
    static uint32_t counter = 0;

    counter++;
    if (counter >= 1000) {
        flag_1s = 1;
        counter = 0;
    }
    if (!(counter % 20)) {
        flag_20ms = 1;
    }

  uwTick += (uint32_t)uwTickFreq;
}
```

The HAL_IncTick function is already defined in the HAL library and is used to increment the system tick count. By default, this is its sole purpose. However, since it is defined as a weak function, I have overridden it to also set the flag_1s and flag_20ms flags to 1 every second and every 20 milliseconds, respectively.

Alarm.c

```c
const Note note1 = {477, 238};
const Note note2 = {506, 253};
```

These two structs represent the two notes that the alarm buzzer will alternate between: note1, which is a C7 note, and note2, which is a B6 note. The first value of each note represents the frequency of the PWM signal (1,000,000 Hz) divided by the frequency of the actual note. The second value is simply half of the first value, used for the pulse width of the PWM signal to achieve a 50% duty cycle.

```c
void Alarm(int alarm_temp, float temp) {
  static uint8_t current_note = 0;

  switch(alarm_state){
    // If alarm is not active and temperature is above alarm temperature, active it
        case ALARM_NOT_ACTIVE:
            if (temp >= (float)alarm_temp) {
                HAL_GPIO_WritePin(GPIOH, GPIO_PIN_10, GPIO_PIN_SET);
                Set_alarm(note1);
                alarm_state = ALARM_IS_ACTIVE;
            }
        break;
        case ALARM_IS_ACTIVE:
            // If alarm is active and temperature is below alarm temperature, deactivate it
            if (temp < (float)alarm_temp) {
                current_note = 0;
                HAL_GPIO_WritePin(GPIOH, GPIO_PIN_10, GPIO_PIN_RESET);
                HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_4);
                alarm_state = ALARM_NOT_ACTIVE;
            // If alarm is active and temperature is still above alarm temperature, change note
            } else {
                if (current_note == 0) {
                    Set_alarm(note2);
                } else {
                    Set_alarm(note1);
                }
                current_note = 1 - current_note;
            }
        break;
    }
}
```

The Alarm function manages both the activation and deactivation of the alarm. It takes the calculated temperature and the set alarm temperature threshold as inputs. If the threshold is exceeded, the function activates the alarm if it was previously deactivated by calling the Set_alarm function with the default note, note1. If the alarm was already active, it changes the note played by the buzzer by calling the Set_alarm function with the new note. Conversely, if the temperature drops below the threshold and the alarm was previously active, the function deactivates the alarm.

```c
void Set_alarm(Note note) {
  HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_4); // stop generation of pwm
  TIM_OC_InitTypeDef sConfigOC;
  htim2.Init.Period = note.period; // set the period duration
  HAL_TIM_PWM_Init(&htim2); // reinititialise with new period value
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = note.pulse; // set the pulse duration
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_4);
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4); // start pwm generation
}
```

Set_alarm configures the PWM to play the specified note and then starts the PWM.

Lut.c

Lut.c contains the look-up table for converting resistance to Celsius, stored as an array of LUTEntry structs along with the size of the table.

Temp_coversion.c

```c
TempPair Binary_search(double res) {
  int left = 0;
  int right = LUTSize - 1;

  while (left <= right) {
        int mid = left + (right - left) / 2;

        if (LUT[mid].res >= res) {
              if (LUT[mid+1].res <= res) {
                    return Get_temps(mid);
              } else {
                    left = mid + 1;
              }
        } else {
              if (LUT[mid-1].res >= res) {
                    return Get_temps(mid-1);
              } else {
                    right = mid - 1;
              }
        }
  }

  TempPair error;
  error.res1 = -1;
  return error;
}
```

The Binary_search function implements a standard binary search algorithm. It finds the two neighboring resistances around the given resistance in the resistance-to-Celsius look-up table. It then calls the Get_temps function, which returns a struct containing the upper and lower neighboring resistances and their corresponding temperatures. These values are used by the Linear_interpolation function to calculate the final temperature value.

Touchscreen.c

```c
// Coordinates of button for increasing alarm temperature
Point button_up[3] = {
              {435, 101},
              {455, 141},
              {415, 141}
  };

// Coordinates of button for decreasing alarm temperature
Point button_down[3] = {
                    {435, 251},
                    {455, 211},
                    {415, 211}
        };
```

These two Point struct arrays store the x and y coordinates of the three points that define each of the two triangular buttons used to increase or decrease the alarm temperature threshold on the LCD screen.

```c
void Touchscreen_template_init(void) {
    ts_status = BSP_ERROR_NONE;
    uint32_t x_size, y_size;

    BSP_LCD_GetXSize(0, &x_size);
    BSP_LCD_GetYSize(0, &y_size);

    hTS.Width = x_size;
    hTS.Height = y_size;
    hTS.Orientation =TS_SWAP_XY ;
    hTS.Accuracy = 5;

    /* Touchscreen initialization */
    ts_status = BSP_TS_Init(0, &hTS);

    if(ts_status == BSP_ERROR_NONE)
    {
        uint32_t x_size, y_size;

        BSP_LCD_GetXSize(0, &x_size);
        BSP_LCD_GetYSize(0, &y_size);


        /* Clear the LCD */
        UTIL_LCD_Clear(UTIL_LCD_COLOR_WHITE);

        /* Set Temperature Alarm description */
        UTIL_LCD_FillRect(0, 0, x_size, 80, UTIL_LCD_COLOR_RED);
        UTIL_LCD_SetTextColor(UTIL_LCD_COLOR_WHITE);
        UTIL_LCD_SetBackColor(UTIL_LCD_COLOR_RED);
        UTIL_LCD_SetFont(&Font24);
        UTIL_LCD_DisplayStringAt(0, 0, (uint8_t *)"Temperature Alarm", CENTER_MODE);
        UTIL_LCD_SetFont(&Font20);
        UTIL_LCD_DisplayStringAt(0, 131, (uint8_t *)"Current Temperature", CENTER_MODE);
        UTIL_LCD_SetFont(&Font12);
        UTIL_LCD_DisplayStringAt(0, 30, (uint8_t *)"Please use the touchscreen to", CENTER_MODE);
        UTIL_LCD_DisplayStringAt(0, 45, (uint8_t *)"change at what temperature", CENTER_MODE);
        UTIL_LCD_DisplayStringAt(0, 60, (uint8_t *)"the alarm will trigger.", CENTER_MODE);

        // Draw button border
        UTIL_LCD_DrawRect(400, 90, 70, y_size - 100, UTIL_LCD_COLOR_RED);
        UTIL_LCD_DrawRect(401, 91, 68, y_size - 102, UTIL_LCD_COLOR_RED);

        // Draw buttons
        UTIL_LCD_FillPolygon(button_up, 3, UTIL_LCD_COLOR_BLACK);
        UTIL_LCD_FillPolygon(button_down, 3, UTIL_LCD_COLOR_BLACK);
    }
}
```

The Touchscreen_template_init function initializes the touchscreen, draws all the static elements of the LCD GUI and displays the buttons in their default state. It first retrieves the screen's width and height by calling the BSP_LCD_GetXSize and BSP_LCD_GetYSize functions. It then configures the touchscreen’s width, height, orientation, and accuracy, and calls BSP_TS_Init to initialize the touchscreen. If initialization is successful, the function clears the LCD display and draws the instructions, GUI layout and buttons in their default (unpressed) state.

```c
Button_Pressed_Typedef Button_check(void) {
  ts_status = BSP_TS_GetState(0, &TS_State);

  if(TS_State.TouchDetected) {
        x = TS_State.TouchX;
        y = TS_State.TouchY;

        if (x <= 470 && x >= 400 && y <= 141 && y >= 91) {
              return BUTTON_PRESSED_UP;
        }
        if (x <= 470 && x >= 400 && y <= 261 && y >= 211) {
              return BUTTON_PRESSED_DOWN;
        }
  }

  return BUTTON_PRESSED_NONE;
}
```

The Button_check function is used to determine if one of the two buttons is being pressed. It first retrieves the touchscreen status by calling the BSP_TS_GetState function. If a touch is detected, it then retrieves the x and y coordinates of the touch and checks if they fall within the hardcoded boundaries of the buttons. For simplicity and ease of use, the boundaries are defined as rectangular areas rather than the triangular shapes of the buttons, making it easier for users to press the small threshold buttons.

```c
void Draw_buttons(Button_Pressed_Typedef current_state, Button_Pressed_Typedef
prev_state) {
  switch(current_state) {
        case BUTTON_PRESSED_NONE:
              if (prev_state == BUTTON_PRESSED_UP) {
                    UTIL_LCD_FillPolygon(button_up, 3, UTIL_LCD_COLOR_BLACK);
              } else {
                    UTIL_LCD_FillPolygon(button_down, 3, UTIL_LCD_COLOR_BLACK);
              }
        break;
        case BUTTON_PRESSED_UP:
              if (prev_state == BUTTON_PRESSED_DOWN) {
                    UTIL_LCD_FillPolygon(button_down, 3, UTIL_LCD_COLOR_BLACK);
              }
              UTIL_LCD_FillPolygon(button_up, 3, UTIL_LCD_COLOR_DARKRED);
        break;
        case BUTTON_PRESSED_DOWN:
              if (prev_state == BUTTON_PRESSED_UP) {
                    UTIL_LCD_FillPolygon(button_up, 3, UTIL_LCD_COLOR_BLACK);
              }
              UTIL_LCD_FillPolygon(button_down, 3, UTIL_LCD_COLOR_DARKRED);
        break;
  }
}
```

The Draw_buttons function updates the button colors based on their current state. If a button is not being pressed, it is drawn in black; if it is pressed, it is drawn in dark red. The function receives both the current and previous states to ensure that only the buttons requiring updates are redrawn, thereby avoiding unnecessary overdraw.

Links

STM32H750-DK_BSP_Touch_Demo Github repository

Temperature Alarm project Github repository