Automotive Alert System with OS

Suraj Ajjampur and Rishikesh Sundaragiri

Final Project Report

ECEN 5613 Embedded System Design

May 5th, 2023

1. INTRODUCTION…… 2
1.1System Overview …… 2

2. TECHNICAL DESCRIPTION …… 3
2.1 Sensor Design……... 3
2.2 Firmware Design…….. 7
2.3 Software Design ……...10
2.4 Operating Systems ……....20
2.5 Testing Process……..22

3. RESULTS AND ERROR ANALYSIS …… 25
3.1 Results …… 25
3.2 Error Analysis …….. 25

4. CONCLUSION …… 26

5. FUTURE DEVELOPMENT IDEAS …….. 26

6. ACKNOLEDGEMENTS …… 27

7. REFERENCES …… 27

8. DIVISION OF LABOR……28

9. APPENDICES …….. 29
8.1 APPENDIX – Bill of Materials……..... 29
8.2 APPENDIX - Schematic …… 29
8.3 APPENDIX – Bare-Metal Source Code ……..29
8.4 APPENDIX – Free RTOS Source Code ……..29
8.5 APPENDIX – Data Sheets and Application Notes ……..29

1 INTRODUCTION

As a duo, my project partner and I share a passion for cars and the engineering behind them. We were particularly inspired by the synergy between sports and engineering in Formula 1 racing. Recently, while driving my roommate's car from Boulder to Longmont in the US, I struggled with staying in the correct lane as I was new to driving on the right side of the road. Fortunately, the car had a few features that made the transition much smoother, including proximity sensors on the blind spots of the back end of the car. These sensors triggered an alert on an LCD screen and emitted a beeping sound when another vehicle was in close proximity. Recognizing the importance of the timing of these alerts for the safety of drivers, passengers, and pedestrians, we decided to build a similar system and conduct a timing analysis using a Real Time operating system.

1.1 System Overview

With this mutual passion for Automobiles and their technical inner-workings, we chose to design an Automotive Alert System for our final project. This design includes novel elements that cover hardware, firmware and software. It is inspired by the Advanced Blind-Spot Monitoring System but it uses custom sensors built from scratch and the STM32F411 – Discovery Board.

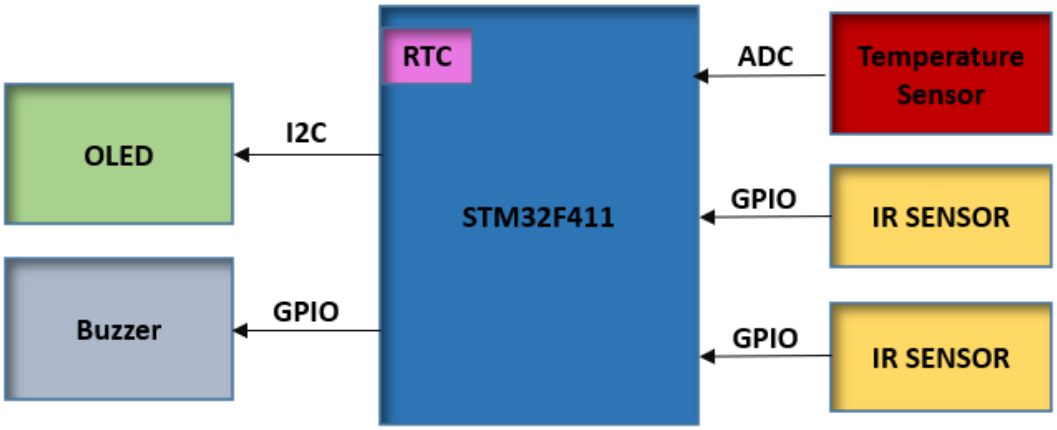

Fig 1.1 Project Block Diagram

2 TECHNICAL DESCRIPTION

This section details the design and implementation of the Automobile Alert System. The section outlined as follows: Sensor Design includes all custom designed sensors, Firmware Design details the RTC, ADC and GPIO functionality and I2C driver Code. Software Design describes the OLED development tools, and the Operating Systems covers the code that describes the RTOS implementation.

2.1 Sensor Design

2.1.1 Temperature Sensor

Our project includes a temperature sensor that is intended to detect the ambient temperature in automobiles. In conventional vehicles, the temperature is typically displayed on the dashboard. Similarly, our project uses an OLED screen that acts as a dashboard, displaying the current ambient temperature detected by the temperature sensor. Fig 2-1 shows the schematic design of the temperature sensor, which is powered by a 3V power supply from the STM32 board. The reason for using 3V instead of 5V is that the output of the temperature sensor should not exceed 3V, as this could potentially damage the GPIO pins of the STM32 board, to which the output of the sensor is connected.

To convert the resistance, change of the temperature sensor into a measurable voltage change, a 10K fixed resistor is connected in series with the NTC resistor. At a normal temperature, the resistance of the 10K temperature sensor is known, and the voltage across the NTC resistance is equal to the supply voltage of the circuit. However, as the temperature changes, the resistance of the NTC resistor changes as well, causing a corresponding change in the voltage across the NTC resistor. The 10K resistor in series with the sensor acts as a voltage divider network, reducing the voltage across the sensor to a measurable level.

The voltage across the NTC resistor is proportional to the resistance of the NTC, which changes with temperature. By measuring the voltage across the sensor using an analog-to-digital converter (ADC), the temperature can be calculated based on a known temperature-resistance table shown in Fig 2-2 which is obtained from the datasheet. The use of a 10K resistor in series with the sensor enables accurate temperature measurements to be made by creating a measurable voltage change across the sensor.

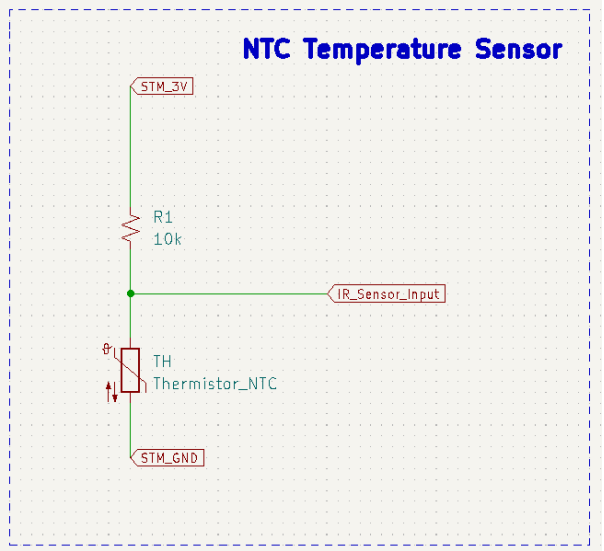

Fig 2-1: Temperature Sensor Schematic

Normal specification Resistance & Temperature Table of MF52-type (Unit : KΩ)

R25 / B / Rt / T(℃)	10 KΩ	50 KΩ	100 KΩ	50 KΩ	50 KΩ	100 KΩ	100 KΩ	150 KΩ
	3950	3950	4000	4050	4150	4150	4300	4500
-30	181.70	908.30	1790.00					
-25	133.30	666.50	1321.00					
-20	98.88	494.50	984.70					
-15	74.10	370.50	740.80					
-10	56.06	280.30	562.30					
-5	42.80	214.00	430.50					
0	98.96	164.80	332.30	168.80	172.00	344.10	352.40	576.70
5	25.58	127.90	257.50	131.30	132.20	264.30	270.00	433.20
10	20.00	99.98	201.10	101.00	102.40	204.80	208.30	328.40
15	15.76	78.79	158.20	79.28	80.03	160.10	161.90	250.90
20	12.51	62.55	125.40	62.78	63.00	125.00	136.70	193.30
25	10.00	50.00	100.00	50.00	50.00	100.00	100.00	150.00
30	8.048	40.24	80.29	39.98	39.76	79.51	78.35	117.30
35	6.518	32.59	64.87	32.16	31.89	63.77	62.37	92.28
40	5.312	26.56	57.72	26.10	25.73	51.45	49.94	73.11
45	4.354	21.77	43.10	21.35	20.88	41.76	40.22	58.28
50	3.588	17.94	35.42	17.72	17.04	34.08	32.56	46.74
55	2.974	14.87	29.26	14.36	13.99	27.97	26.40	37.71
60	2.476	12.38	24.30	11.92	11.53	23.06	21.53	30.58
65	2.072	10.36	20.27	9.938	9.541	19.08	17.69	24.94
70	1.743	8.717	16.99	8.317	7.929	15.86	14.62	20.45
75	1.473	7.364	14.31	6.991	6.621	13.24	12.20	16.85
80	1.250	6.248	12.10	5.906	5.552	11.10	10.05	13.94
85	1.065	5.324	10.27	5.012	4.674	9.348	8.376	11.60
90	0.911	4.555	8.758	4.271	3.950	7.900	7.004	9.680
95	0.7824	3.912	7.495	3.654	3.349	6.698	5.894	8.118
100	0.6744	3.372	6.438	3.316	2.849	5.698	4.978	6.836
105	0.5836	2.918	5.550	2.701	2.438	4.875	4.215	5.780
110	0.5066	2.533	4.801	2.336	2.093	4.186	3.580	4.904

Fig 2-2: Temperature – Resistance table

2.1.2 IR Sensor

The key components are the IR led and the photodiode. IR LED emits light in the IR spectrum, which isn't visible to the naked eye. The photodiode also known as a photo resistor, has a very high resistance, in the absence of light and the resistance of the diode drops in the presence of light. Resistance is inversely proportional to intensity of light.

It is a semiconductor which has a PN junction and is operated in Reverse Bias.

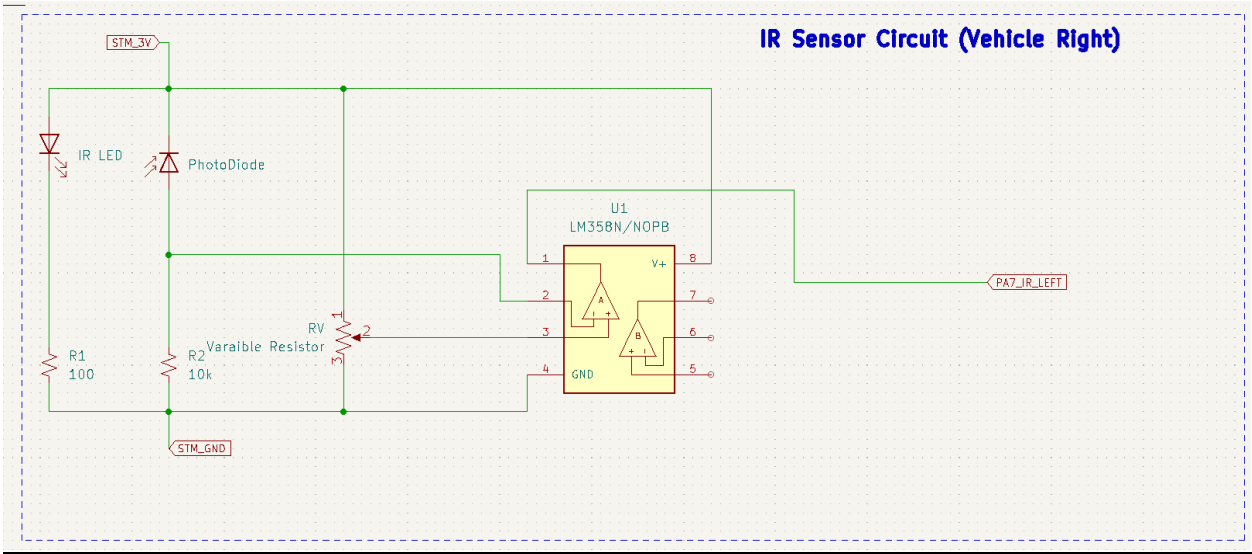

Fig 2.1.2 IR sensor circuit

When it is powered on, the IR LED emits light and when it is reflected from an object and falls on the photodiode, the resistance of the photodiode drops. Therefore, the voltage also changes. This is connected to the Inverting terminal of the op amp.

The potentiometer is kept as a reference, when combined with the 10k ohm resistance is given to the non-inverting terminal of the op amp.
If the voltage at V+ is lower than V- then the voltage at the output will be 0.
If the voltage at V+ is greater than V- then the voltage at the output will be 1.
Hence the Output of the circuit will be digital.

In our circuit, we have used a 5mm diameter and a wavelength of 940nm Infrared Emitter LEDs which are characterized by strong structure, corrosion resistance, high reliability and long life.

The photodiode has a forward voltage rating of 1.2 -1.3V. Even though it has a transmitting and receiving distance of 7-8m, the receiving angle of 40 degrees reduces the range drastically. When an object comes close to the receiver transmitter circuit within a certain range then there is a resistance drop which results in a voltage drop which is reflected on the -Vref of the op-amp.

A potentiometer is connected to the +Vref of the op amp and is adjusted to a resistance of 7k Ω. When the -Vref drops below 2.83 volts when an object is detected the op amp gives an output of 1.76 volts which is provided as an input to the STM32 GPIO pins. Based upon this positive signal on the GPIO pin the relevant action is taken.

2.2 Firmware Design

2.2.1 RTC

The STM32 has an inbuilt real-time clock module which provides various features, including timekeeping, alarm functionality, backup power management, calendar functionality, and timestamping. The RTC module includes a backup battery input pin that ensures accurate timekeeping during power failures. In our project, we use the RTC module to obtain the date, month, and year using the calendar functionality and to display the current time in 24-hour format on an OLED screen. The RTC module is like the dashboard in cars that displays the current calendar and time.

The configuration of the RTC module in bare metal was accomplished by utilizing the Reference manual[6] and application note[7] of STM32 to program the RTC. To begin, an initialization function was defined to set up the RTC module. This involved enabling the clock for the power interface, disabling the write protect to the RTC, and activating the low-speed internal (LSI) RC oscillator. The function will wait until the LSI RC oscillator is ready and then proceed to enable the clock for the RTC module and select LSI as the clock source. The prescaler is also set to bring down the LSI frequency to 1Hz, and the TR and DR registers were initialized for time and date. Lastly, the bypass mode for the RTC module was enabled, and the Init bit was cleared to exit initialization mode. After initialization, the TR register can be read to retrieve time, and the DR register can be read to retrieve the date. However, the format of the date and time may not be suitable, so two additional functions were implemented to shift bits and convert the date into the format MM-DD-YY and the time into the format HH-MM.

2.2.2 ADC

To obtain the exact temperature using the temperature sensor discussed earlier, we utilized the ADC module which is built-in to the STM32 and programmed it using bare metal and interrupts. The temperature sensor outputs analog values which are proportional to the ambient temperature. To convert these raw values into a real-time temperature measurement, an algorithm is applied to them. This involves reading the raw values through the ADC and then performing the necessary calculations.

To enable the ADC module with interrupts, we first configured the corresponding registers in a sequence of steps. The ADC was initialized by enabling the clock source for GPIOA port and configuring pin PA0 as an analog input. Then, the clock source for the ADC1 module was enabled with a resolution set to 10 bits, resulting in a maximum ADC output value of 1023. Although the ADC was initially disabled, it was enabled later to initiate the conversion process. Additionally, the ADC was configured to generate an interrupt after

the conversion was completed. Finally, we set the priority level for the ADC interrupt and initiated the conversion process. With this configuration, the microcontroller can now convert analog signals from sensors into digital values, facilitating further processing.

To process the temperature value, it needs to be run through an algorithm defined in the function calculate_temperature(). This function performs several calculations to obtain the temperature value from the ADC reading. First, it converts the ADC reading to an analog voltage using multiplication and division factors. Then, it calculates the NTC resistance based on the analog voltage and known circuit parameters. Next, it uses a lookup table to map the NTC resistance value to a temperature value and sets the corresponding temperature value to a variable. The function uses a single lookup table for temperature-to-resistance conversion, which assumes that the NTC thermistor has a specific resistance versus temperature characteristic and that the lookup table is appropriately calibrated to match the thermistor. It is important to note that the temperature values are defined as constants and that the temperature value is set based on the index of the lookup table where the NTC resistance value falls within the specified ranges in the if-else statements.

2.2.3 I2C

The communication between the STM32 board and the SSD1306 OLED display is achieved using the I2C communication protocol. This protocol is a serial communication standard that employs two wires to connect multiple integrated circuits on a single bus. In the I2C communication protocol, one device acts as the master, while the other devices act as slaves. The master device controls the communication by initiating data transfer, and the slave devices respond to the master's commands.

To establish the connection, the SDA pin of the OLED is connected to the PB7 pin of the STM32 board, and the SCL pin of the OLED is connected to the PB6 pin of the STM32 board. The SDA and SCL pins of the OLED is shown in Fig 2.2.3(a).

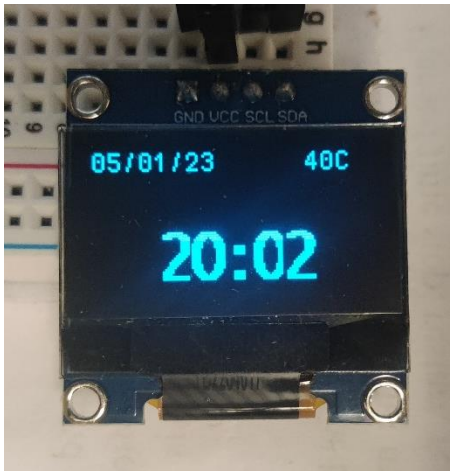

Fig 2.2.3(a) OLED I2C Connection

To configure the I2C communication between the STM32 board and the SSD1306 OLED display, a clock frequency of 400 kHz is set for fast mode, the duty cycle is set to 50%, the addressing mode is set to 7-bit addressing, and the clock stretching mode is disabled.

The SSD1306_Init() function initializes the communication with the SSD1306 OLED LCD display by first calling the ssd1306_I2C_Init() function to initialize the I2C interface. The function then verifies that the display is connected to the I2C bus by calling the HAL_I2C_IsDeviceReady() function, which returns 0 if the display is not ready.

Once the display connection is verified, the function sends a series of commands to the display using the SSD1306_WRITECOMMAND() function. These commands configure various display settings, such as the display resolution, contrast, and scan direction.

After configuring the display, the SSD1306_Init() function clears the screen by calling the SSD1306_Fill() function and updates the display by calling the SSD1306_UpdateScreen() function. Finally, the function sets the default values for the CurrentX and CurrentY variables, which indicate the current position of the cursor on the display, and sets the Initialized variable to 1 to indicate that the display has been successfully initialized. The diagram below illustrates the logic signals of SCL and SDA during the SSD1306_Init() function.

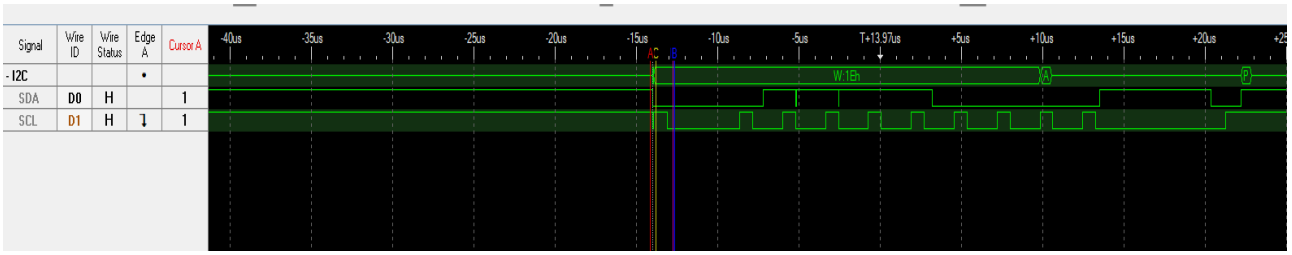

Fig 2.2.3(b) I2C Timing Diagram

2.2.4 GPIO

Configuring the GPIO pins is necessary for various purposes. To read the status of the IR sensor output and activate the buzzer when a vehicle is detected in front of the IR sensor, we require GPIO pin configuration. We opted for the bare metal approach to configure the GPIO pins.

To use a GPIO, the initial step is to perform configuration of the GPIO registers by setting specific bits to corresponding modes or settings. This process is known as initialization. In our case, the initialization function initializes the necessary hardware peripherals to enable the functioning of an infrared (IR) sensor module. This module is intended to detect and interpret IR signals emitted by remote controllers, as well as to activate a buzzer when an IR signal is detected. The function starts by enabling the clock for GPIO port A, which will be used to connect the IR sensor and buzzer. It then configures three pins of port A, namely PA7, PA2 and PA5, to input or output mode according to their specific functions. For PA7 and PA2, which are the IR sensor pins, the function sets them to input mode and enables

pull-down resistors to ensure that the pins remain at a known logic level when no signal is present. This configuration prevents potential issues caused by floating pins and minimizes power consumption. For PA5, which is the buzzer pin, the function sets it to output mode and enables a pull-up resistor. The output type register is set to push-pull mode, which means that the pin can both sink and source current. Finally, the function completes its initialization by setting the PA5 output low, so the buzzer is initially off.

The second objective involves verifying the IR sensor output status. This entails observing the output pin of the IR sensor that is connected to the GPIO pin. When the output pin is in a high state, it implies that a vehicle is nearby. To accomplish this, a function was developed which is accountable for identifying the presence of a vehicle on either the left, right or both sides. The function reads the input from two IR sensors connected to pins PA7 and PA2 of GPIOA respectively. If the input from PA7 is high and PA2 is low, it indicates that a vehicle is detected on the left side. The function sets the flag vehicle_leftside to true and turns on the buzzer connected to PA5 to alert the user. It then checks if the clear_count_left flag is set and clears the OLED screen if it is. The function then draws an image on the OLED screen to indicate the vehicle's position and updates the screen. The function sets the counter variable to 1 to indicate that the alert has been triggered. If the input from PA7 is low and PA2 is high, it indicates that a vehicle is detected on the right side. The function sets the flag vehicle_rightside to true and turns on the buzzer connected to PA5 to alert the user. It then checks if the clear_count_right flag is set and clears the OLED screen if it is. The function then draws an image on the OLED screen to indicate the vehicle's position and updates the screen. The function sets the counter variable to 1 to indicate that the alert has been triggered. If the input from both PA7 and PA2 are high, it indicates that a vehicle is detected on both sides. The function sets the flag vechile_bothside to true, turns on the buzzer connected to PA5 to alert the user, and draws an image on the OLED screen to indicate the vehicle's position. The function sets the clear_count_both flag to 1 to indicate that the screen needs to be cleared. If none of the above conditions are met, the function sets all flags to false and turns off the buzzer connected to PA5. If any of the clear_count flags are set, the OLED screen is cleared, and the alert mode will disappear.

2.3 Software Design

To fully utilize the functionality of the SSD1306 OLED display, it is necessary to translate the information to be displayed onto the display. The process of translation is accomplished using functions sourced from the reference [4] by Controllers Tech.

Before using these functions, the SSD1306_Init() function must be called to initialize the SSD1306 OLED display, as explained in the previous section. To navigate to a specific position on the OLED display, the SSD1306_GotoXY() function is used. This function takes two variables, x and y, which represent the coordinates of the new position, and sets the cursor to that position on the bitmap.

```
void SSD1306_GotoXY(uint16_t
x, uint16_t y) {
       /* Set write pointers */
       SSD1306.CurrentX = x;
       SSD1306.CurrentY = y;
   }
```

2.3.1 Display Text

To display the temperature on the SSD1306 OLED screen, the SSD1306_Putc() function is used. This function is responsible for displaying a single character on the SSD1306 OLED screen. The function takes three arguments. The first argument, ch, is the ASCII code of the character to be written. The second argument, font, is a pointer to a structure that defines the font to be used for the character. The third argument, color, specifies the color of the character. The function first checks if there is enough available space on the display to write the character. If there is not enough space available, the function returns 0, indicating an error.

After checking the available space, the SSD1306_Putc() function goes through the font data to obtain the row data of the character to be written. For each row of the character, the function goes through the columns of the character and draws a pixel on the display by calling the SSD1306_DrawPixel() function. The SSD1306_DrawPixel() function takes the current position of the cursor, SSD1306.CurrentX and SSD1306.CurrentY, adds the current row and column index, and sets or clears the corresponding pixel on the display based on the value of the font data and the specified color. After writing the character on the display, the function increases the SSD1306.CurrentX pointer by the width of the font to prepare for writing the next character.

Finally, the function returns the character that was written on the display, indicating a successful write operation.

```
char SSD1306_Putc(char ch, FontDef_t* Font, SSD1306_COLOR_t color) {
	uint32_t i, b, j;

	/* Check available space in LCD */
	if (
		SSD1306_WIDTH <= (SSD1306.CurrentX + Font->FontWidth) ||
		SSD1306_HEIGHT <= (SSD1306.CurrentY + Font->FontHeight)
	) {
		/* Error */
		return 0;
	}

	/* Go through font */
	for (i = 0; i < Font->FontHeight; i++) {
		/* Get row data */
		b = Font->data[(ch - 32) * Font->FontHeight + i];
		/* Go through columns */
		for (j = 0; j < Font->FontWidth; j++) {
			/* Draw pixel */
			if ((b << j) & 0x8000) {
				SSD1306_DrawPixel(SSD1306.CurrentX + j,
(SSD1306.CurrentY + i), (SSD1306_COLOR_t) color);
			} else {
				SSD1306_DrawPixel(SSD1306.CurrentX + j,
(SSD1306.CurrentY + i), (SSD1306_COLOR_t)!color);
			}
		}
	}

	/* Increase pointer */
	SSD1306.CurrentX += Font->FontWidth;

	/* Return character written */
	return ch;
}
```

In this project, we need fonts to display characters on the screen. These fonts are listed in fonts.c and fonts.h files. Each font contains the hex representation of each ASCII character and is stored in a single array for that particular font type. The font is represented by a structure type with the format shown below.

```
typedef struct {
	uint8_t FontWidth;
/*!< Font width in pixels */
	uint8_t FontHeight;
/*!< Font height in pixels */
	const uint16_t *data;
/*!< Pointer to data font data
array */
    } FontDef_t;
```

To input strings on the SSD1306 OLED screen, such as date and temperature, the functions SSD1306_Puts(temp_string, &Font_7x10) are used. The SSD1306_Puts function takes three arguments: str, which is a pointer to the string to be written, Font, which is a pointer to a structure that defines the font to be used for the characters, and color, which specifies the color of the characters.

The SSD1306_Puts function iterates over the characters of the string one by one and writes each character on the display by calling the SSD1306_Putc function. The SSD1306_Putc function takes the current character, the font, and the color as arguments and returns the character that was written on the display. If the SSD1306_Putc function returns a value different from the current character, it indicates an error, and the SSD1306_Puts function returns the current character.

After writing each character on the display, the function increases the pointer to the string to prepare for writing the next character. Finally, the function returns zero to indicate that the operation was successful.

```
char SSD1306_Puts(char* str, FontDef_t* Font, SSD1306_COLOR_t color) {
	/* Write characters */
	while (*str) {
		/* Write character by character */
		if (SSD1306_Putc(*str, Font, color) != *str) {
			/* Return error */
			return *str;
		}

		/* Increase string pointer */
		str++;
	}

	/* Everything OK, zero should be returned */
	return *str;
}
```

To avoid overwriting existing pixels on the screen, it is important to call the SSD1306_Clear function before writing any data to the LCD. The purpose of the SSD1306_Clear() function is to clear the SSD1306 OLED LCD display by filling the display with black pixels. This is achieved by calling the SSD1306_Fill() function with the

color set to black (0), which fills the entire display with black pixels. Then, the function calls the SSD1306_UpdateScreen() function to update the display with the new black pixels.

The SSD1306_Fill() function is responsible for filling the SSD1306_Buffer array with a specified color. It takes one argument, which is the color to be used to fill the buffer. The function uses the memset() function to set all the bytes in the SSD1306_Buffer array to either 0x00 (black color) or 0xFF (white color), depending on the specified color.

```
void SSD1306_Clear (void)
{
      SSD1306_Fill (0);
    SSD1306_UpdateScreen();
}
```

After writing pixels on the OLED, it is necessary to update the display to show the written content. To do this, the SSD1306_UpdateScreen function is called.

This function updates the SSD1306 OLED LCD display by writing the content of the SSD1306_Buffer array to the display memory. The function starts by iterating over the eight pages of the display by sending the corresponding page address command (0xB0 + page) to the display. For each page, the function sets the column address to begin at 0 by sending the column address command (0x00 and 0x10) to the display.

After iterating over the display pages and setting the column address, the SSD1306_UpdateScreen() function uses the ssd1306_I2C_WriteMulti() function to write data to the display memory. This function requires four arguments: address, which is the I2C address of the SSD1306 display; reg, which is the register address to write to; data, which is the data to be written to the register; and count, which is the number of bytes to be written. The ssd1306_I2C_WriteMulti() function constructs a buffer containing the register address followed by the data and sends the buffer to the display using the HAL_I2C_Master_Transmit() function.

In this situation, the ssd1306_I2C_WriteMulti() function is invoked with three arguments: the I2C address of the SSD1306 display, which is the first argument, and the register address, which is set to 0x40. The last argument is a pointer to the first byte of the row data

```
void SSD1306_UpdateScreen(void) {
	uint8_t m;

	for (m = 0; m < 8; m++) {
		SSD1306_WRITECOMMAND(0xB0 + m);
		SSD1306_WRITECOMMAND(0x00);
		SSD1306_WRITECOMMAND(0x10);

		/* Write multi data */
		ssd1306_I2C_WriteMulti(SSD1306_I2C_ADDR, 0x40,
&SSD1306_Buffer[SSD1306_WIDTH * m], SSD1306_WIDTH);
	}
}
```

in the SSD1306_Buffer array for the current page, while the count argument is set to the width of the display, which is equal to the number of bytes per row of data.

2.3.2 Creating bitmaps

This section explains the process of displaying an image on the SSD1306 OLED LCD module using Fig 2.3.2(a).

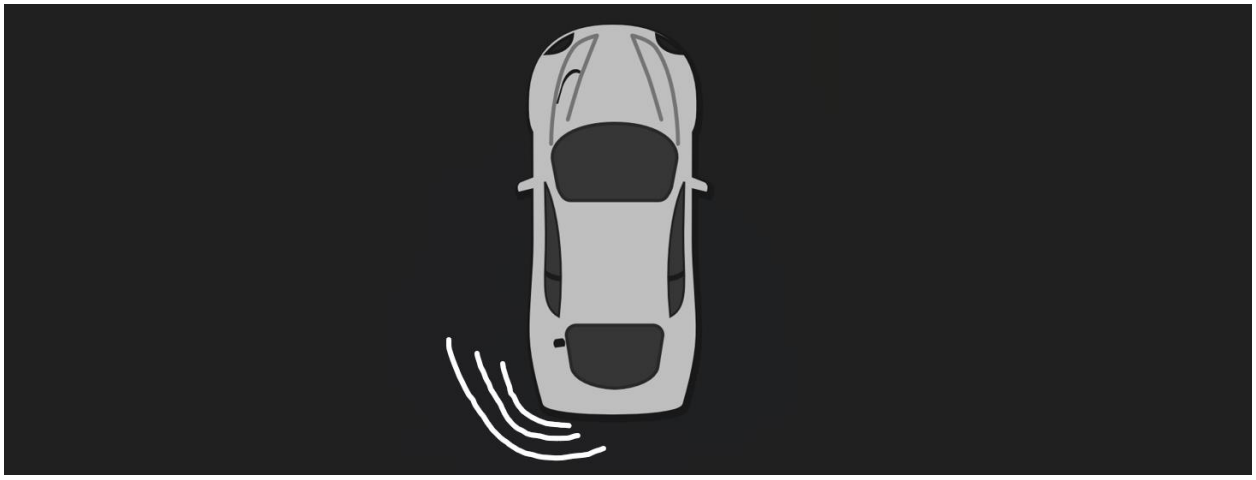

Fig 2.3.2(a) Left Side Car PNG.

To modify the image, we used GIMP, which is a free and open-source tool used for tasks such as photo retouching, image composition, and image authoring.

The first step is to resize the image to 128x64 by clicking on Image->Scale Image and making the necessary changes in the window displayed as shown in Fig 2.3.2(b)

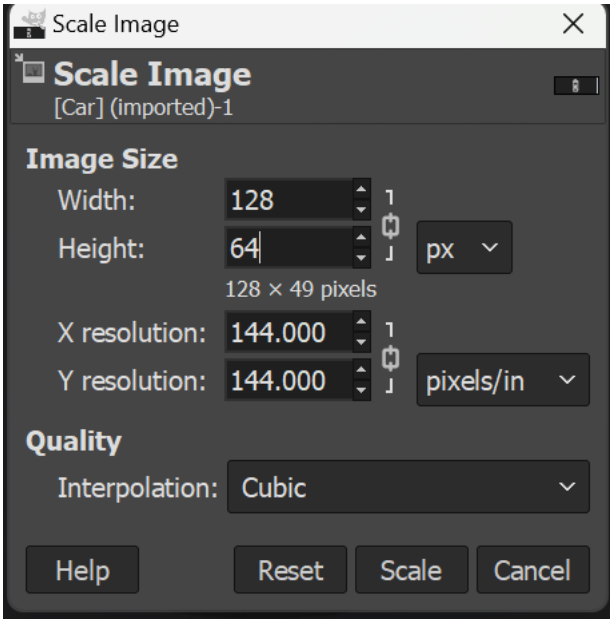

Fig 2.3.1(b) GIMP Scale Image Window

Next, we change the color index to 1 bit black and white only, as our OLED supports only 1 bit color index. It can be either 0 or 1. This is achieved by clicking on Image->Mode->Indexed and configuring the settings in the Indexed color conversion window, as shown in the Fig 2.3.1(c)

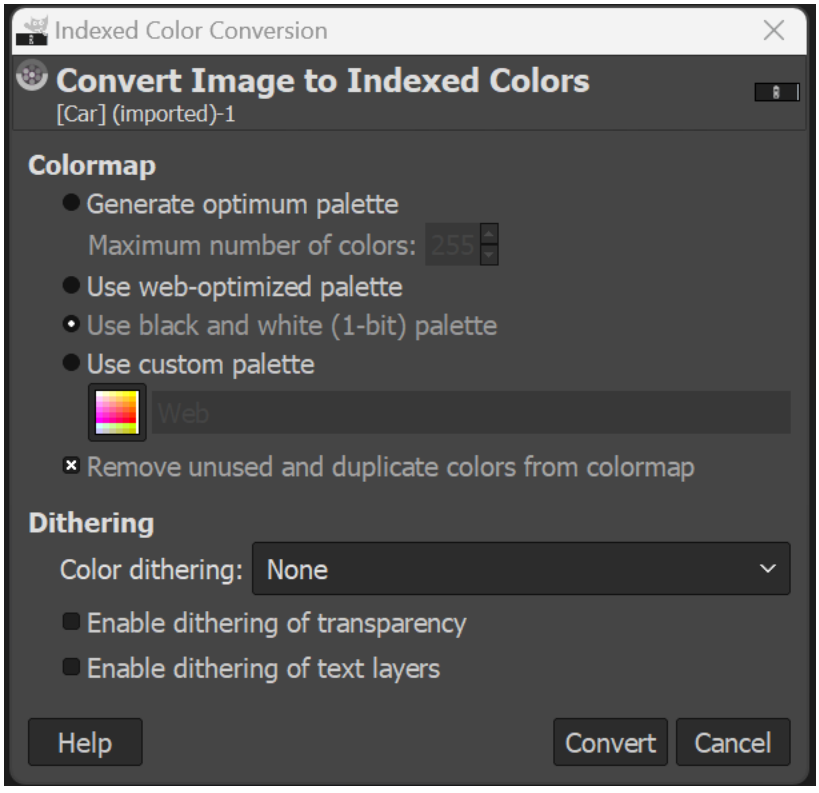

Fig 2.3.1(c) GIMP Indexed Color Conversion

Next, the image is exported in the .bmp format and opened in LCDAssistant to convert the bitmap file to a hex file. It is important to ensure that the byte orientation is horizontal and the Endianness is big for the conversion process. The required settings are shown in the figure provided.

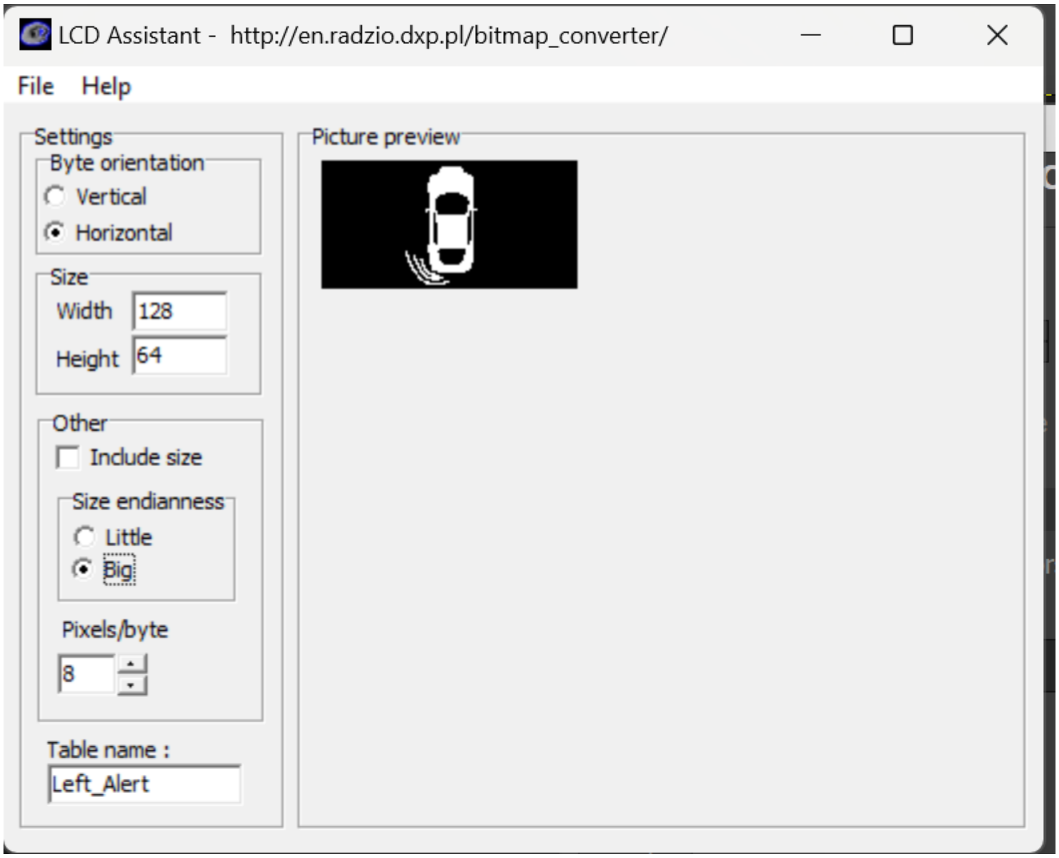

Fig 2.3(d) LCD Assistant Window

Once the hex file is generated it contains a const array of unsigned char data type

```
const unsigned char Left_Alert [] = {
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF0, 0x03, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE0, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE0, 0x01, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xC0, 0x00, 0x7F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE, 0x00, 0x00, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
```

```
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x03, 0xF8, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x1F, 0xFE, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x3F, 0xFF, 0x8F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x7F, 0xFF, 0x8F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x7F, 0xFF, 0xCF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x7F, 0xFF, 0xCF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x7F, 0xFF, 0x8F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x7F, 0xFF, 0x8F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF0, 0x7F, 0xFF, 0x83, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF4, 0x3F, 0xFF, 0xAF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xBF, 0xFF, 0x2F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x80, 0x00, 0x2F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x80, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x80, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x80, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x6F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xC0, 0x00, 0x4F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x40, 0x00, 0x4F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x00, 0x00, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF8, 0x1F, 0xFF, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xDF, 0xF8, 0x3F, 0xFF, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xDF, 0xFC, 0x1F, 0xFF, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xCD, 0xFC, 0x1F, 0xFE, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xFC, 0x1F, 0xFE, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0x9C, 0x1F, 0xFE, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE6, 0xDC, 0x0F, 0xFC, 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF6, 0x5C, 0x07, 0xF8, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF3, 0x4E, 0x00, 0x00, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFB, 0x2E, 0x00, 0x00, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF9, 0xA6, 0x00, 0x00, 0x3F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0x93, 0xC0, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE, 0xC8, 0x7F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x66, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x30, 0x07, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x9E, 0x1F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xE3, 0xF8, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xF0, 0x03, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
```

The SSD1306_DrawBitmap() function is used to draw a bitmap on the SSD1306 OLED LCD display. It takes several parameters, including the starting x and y coordinates, the width and height of the bitmap, and the color of the image. The function calls the SSD1306_DrawPixel() function iteratively to draw each pixel of the bitmap.

The SSD1306_DrawPixel() function is responsible for drawing a single pixel on the SSD1306 OLED LCD display. It takes three arguments: x and y, which are the coordinates of the pixel to be drawn, and color, which specifies the color of the pixel. The function first checks if the specified coordinates are within the bounds of the display. If they are out of bounds, the function returns indicating an error.

Next, the function checks if the display is inverted by checking the value of the SSD1306.Inverted variable. If the display is inverted, the color parameter is inverted by using the logical NOT operator. Finally, the function sets the color of the pixel by modifying the corresponding bit in the SSD1306_Buffer array.

The SSD1306_Buffer array is a one-dimensional array of size SSD1306_WIDTH x SSD1306_HEIGHT / 8, where each byte represents a vertical slice of 8 pixels in the display. To set the color of the pixel, the function calculates the index of the byte that contains the pixel using the formula x + (y / 8) * SSD1306_WIDTH. Then, the function sets or clears the corresponding bit in the byte using the bit-shifting and bitwise OR/AND operations.

```
void SSD1306_DrawBitmap(int16_t x, int16_t y, const unsigned char* bitmap,
int16_t w, int16_t h, uint16_t color)
{

    int16_t byteWidth = (w + 7) / 8; // Bitmap scanline pad = whole byte
    uint8_t byte = 0;

    for(int16_t j=0; j<h; j++, y++)
    {
        for(int16_t i=0; i<w; i++)
        {
            if(i & 7)
            {
               byte <<= 1;
            }
            else
            {
               byte = (*(const unsigned char *)(&bitmap[j * byteWidth + i /
8]));
            }
            if(byte & 0x80) SSD1306_DrawPixel(x+i, y, color);
        }
    }
}
```

Overall, the SSD1306_DrawBitmap() function is used to draw a bitmap on the SSD1306 OLED LCD display by calling the SSD1306_DrawPixel() function iteratively. The SSD1306_DrawPixel() function is responsible for drawing a single pixel on the display by setting the corresponding bit in the SSD1306_Buffer array. It checks if the specified coordinates are within the bounds of the display, checks if the display is inverted, and sets or clears the corresponding bit in the SSD1306_Buffer array based on the specified color.

2.4 Operating Systems

Our project is utilizing the FreeRTOS operating system. The decision to use FreeRTOS was based on its open-source nature and its compatibility with most ARM-based processors, making it easy to schedule and write tasks. The STM32, which we are using, has a built-in RTOS feature that can be enabled through the STM32CubeIDE. This feature allows us to configure tasks, set priorities, select preemption, and perform scheduling. Although the STM32CubeIDE does not write the code for us, it provides tasks and their handlers, based on our choices of the number of tasks and their priorities at the beginning of the project. After creating the project with these tasks, we can use the ioc GUI to edit, add, or remove tasks. Once the tasks are defined, it is up to us to decide how to schedule them in the most effective manner.

In RTOS, the scheduling of tasks is essential to ensure that the system is running efficiently and effectively. By assigning priorities to tasks, the RTOS can allocate CPU resources to the most critical tasks first, ensuring that they are executed in a timely and accurate manner. The OSdelay parameter is used to determine how often a task is executed, and it should be set according to the task's criticality and the system's requirements. If the OSdelay value is set too low, the system may become overloaded, leading to missed deadlines and system failures. Conversely, if the OSdelay value is set too high, the system may not respond quickly enough to critical events, leading to poor system performance. Therefore, it is essential to schedule tasks with the right OSdelay value to ensure that the system is running smoothly and efficiently.

The diagram depicted in Fig 2.4(a) illustrates how the RTOS is scheduled for our project. There are three main tasks in our project, out of which the most critical task is the one that reads the IR sensor, and it has been assigned above-normal priority. It is vital to detect the presence of a vehicle approaching in the blind spot using the IR sensor, as it directly affects the safety of the passengers. Therefore, this task cannot be missed or delayed. The OSdelay parameter used for this task is 100 milliseconds, which means that the task will be executed every 100 milliseconds. As soon as the task finishes running, it relinquishes the CPU to other tasks for 100 milliseconds, after which it preempts the other tasks and takes control of the CPU to read the IR sensor again. It is crucial to schedule tasks with the appropriate OSdelay value; otherwise, the RTOS will not allocate the CPU resources effectively.

The second task of the project involves displaying the date, time, and temperature on the OLED screen during normal conditions, and an animation during alert conditions. This task has a normal priority, and its execution is scheduled with an osDelay of 1ms. The task is designed to run whenever the first task of IR detection with above normal priority completes its 100ms osDelay. If the above normal task finishes its execution before 100ms, then the second task will not be executed until the above normal task has finished executing. The above normal task takes precedence over the normal task and will preempt its execution as soon as its 100ms delay is over. This prioritization ensures that the time-critical task of detecting the IR sensor when vehicle approaches the blind spots takes precedence over displaying the OLED screen with the time, date , year, month and the temperature.

In our project, we have implemented an idle task as the last task in our RTOS. The purpose of the idle task is to keep the CPU in a low power state when it is not being used by the other tasks in the system. This is important in real-time systems, where the system must respond to events within a specific timeframe. The idle task is responsible for managing any hardware that needs to be serviced periodically, such as refreshing the display or updating the system clock. Additionally, it can perform background maintenance tasks, monitor system health, or run diagnostic tests.

The idle task typically does very little work and enters a low power mode to conserve energy while it waits for the next event. The idle task is scheduled at the lowest priority level, which means that it will only run when there are no other higher-priority tasks in the system that need the CPU's attention. The other tasks in the system can preempt the idle task at any time when they require CPU resources to perform their actions.

The main benefit of the idle task is that it helps to ensure that the system is always responsive and operating at optimal efficiency. Without the idle task, the CPU would be idle and consuming unnecessary power, which could reduce the system's overall performance.

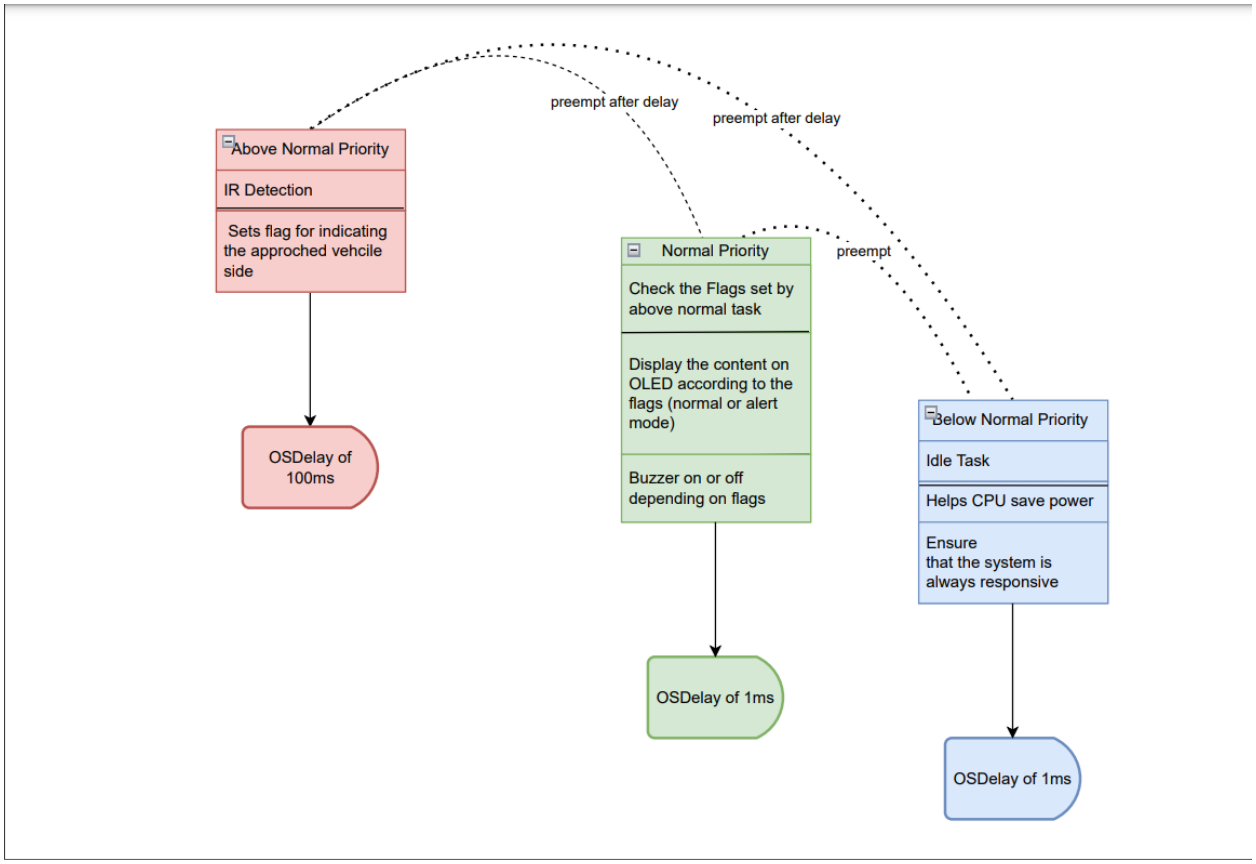

Fig 2.4(a) RTOS Scheduling diagram

2.5 Testing process

During the development process, we employed the Agile methodology with a focus on continuous integration and testing. We used a version control in the form of Github to keep track of our progress and have now made the repository public after the demonstration for easy access. As soon as the hardware parts arrived, we designed and tested the sensors using a Digital Multimeter tool. For software testing, we wrote code for each element and debugged it using breakpoints to verify that we obtained the desired result.

2.5.1 Hardware Testing

The measured voltages of each component in the circuit are presented in figures 2.5.1(a) to 2.5.1(g) after being measured with the Digital multimeter.

Fig 2.5.1(a) +Vref

Fig 2.5.1(b) -Vref without object

Fig 2.5.1(c) -Vref with object

The operation amplifier functions as a comparator, whereby the output is expected to be 1 when +Vref is greater than -Vref.

Fig 2.5.1(d) Op Amp output without object

Fig 2.5.1(e) Op Amp output with object

The screenshots of the temperature sensor measured NTC resistance are displayed in figures 2.5.1(f) and 2.5.1(g). It can be observed that the resistance and temperature are inversely proportional.

Fig 2.5.1(f) Resistance across NTC at 28C

Fig 2.5.1(g) Resistance across NTC at 45C

2.5.1 Software Testing

During software development, incremental testing was performed after integrating each module. The RTC module was tested by adding watch variables to verify if the seconds, minutes and hours were incrementing correctly. For the ADC module, datasheet values were referred to compare with the sensor values to verify if the ADC module was functioning correctly. GPIO pins were verified using a multimeter, for example, checking if the buzzer pin was giving an output voltage of 3V to drive the buzzer high.

As the implementation of FreeRTOS started with limited background knowledge, the team explored the use of preemption and context switching for implementation. Tasks were defined and the osDelay() function was learned. To compare the response time of FreeRTOS and Bare-Metal implementation, timers were used to measure the time taken between the IR sensor object detection and the LCD Update and the LCD turning on. The Systick Timer was used for the Bare Metal implementation, and the General Purpose timer was used for the FreeRTOS measurement as the Systick Timer was used for FreeRTOS internal functionality.

3 RESULTS AND ERROR ANALYSIS

3.1 Results

A study was conducted to measure the time it takes for the IR sensor to detect an approaching vehicle and update the OLED display. The results of the study are presented in the Table 3.1. There is an improvement in the response time when we use the FreeRTOS as compared to the Bare-metal implementation.

Sample	Left Alert(ms)		Right Alert(ms)		Both Side Alert(ms)	
	Bare-Metal	RTOS	Bare-Metal	RTOS	Bare-Metal	RTOS
1	133	67	132	62	95	92
2	133	84	133	69	95	72
3	132	52	133	52	95	55
4	132	84	133	84	95	54
5	133	58	132	58	95	58
Average	133	69	133	65	95	66
Difference	64		68		29	

Table 3.1

Based on the Delaware DMV's research [8], the average reaction time of a driver is 0.75 seconds. This study emphasizes the importance of an alert system that runs on a robust operating system, as it can play a crucial role in preventing road accidents and fatalities.

3.2 Error Analysis

During testing, we faced issues with the IR sensor as it did not produce a logic high when an object was withing the range of the IR sensor. To fix this, we checked the resistance across the photodiode, which was changing. Then, we checked the voltage across the potentiometer to see if it reduced when an object was detected. The problem was with the voltage at +Vref, which came from the potentiometer. We changed the potentiometer resistance to 7k ohm and set the voltage to 2.84 V. This enabled the op amp to output produce a logic high when -Vref dropped below +Vref when an object was close enough to the emitter-receiver circuit.

While utilizing FreeRTOS, we encountered timing difficulties with the I2C protocol, which mandates that the start and stop bits occur at precise intervals with appropriate setup and hold times. However, we found that this was not occurring in our case. This was due to the higher-priority task preempting the second task responsible for the OLED I2C interface during the transmission of data frames, resulting in the screen freezing before the stop bit was received. We invested a significant amount of time in debugging and testing different scheduling techniques to identify the most effective approach. After numerous hours of trial and error, we eventually uncovered the root cause and implemented an appropriate delay for proper scheduling.

4 CONCLUSION

As a team with a keen interest in cars and their systems, we found this project to be a fantastic opportunity to delve into the hardware, firmware, software, and operating systems techniques that are involved in these systems. The project was centered around creating a real-time safety system, which had some limitations despite being successful.

During the project, we were able to compare the response time of a Bare Metal system versus RTOS system. While it may not have been the best Bare Metal or RTOS approach, we wanted to compare the difference between them and observe how the time changed. Our intention was not to prove anything, but to gain an understanding of how to define and schedule tasks in an RTOS environment. As a result of our implementation, the study showed that RTOS has a faster response time than Bare Metal, making it ideal for time-critical operations such as the alert system we developed. Overall, this project provided valuable insights into the workings of an RTOS system and how it can be leveraged to improve the performance of real-time systems.

5 FUTURE DEVELOPMENT IDEAS

In its current state, the automobile system comprises a sensory system and a dashboard that includes a calendar, clock, temperature display, and animations. However, there are some additional features that have not yet been implemented that would make the system more comprehensive. For instance, a sound system could be integrated into the existing setup, allowing users to connect a speaker and play various audio files. The current song being played could be displayed on the dashboard's OLED screen, and users could use a rotary switch to change songs.

The STM32 board provides the capability to add a battery to the RTC module. By adding a battery, we can ensure that the RTC always has the correct time, even when the system is powered off. This eliminates the need for us to set the time every time the system is booted up, as the RTC will retain the current time even when the system is powered off.

6 ACKNOLEDGEMENTS

We express our gratitude towards our fellow classmates for their unwavering encouragement and support throughout this project. Their curiosity about our project gave us the opportunity to explain its functionality, which in turn enhanced our understanding of its working.

We would also like to extend our thanks to Maanas for verifying our IR sensor circuit when it wasn't functioning properly and for providing us with confidence in the FreeRTOS implementation of our project. Additionally, we would like to thank Saanish and Jordi for their support in Bare-metal programming with STM32 board throughout the semester.

Our sincere appreciation goes to the authors of the various sources cited in the references section.

Finally, we would like to express our appreciation to our course instructor, Prof. Linden McClure, for designing a well-structured course that enabled us to develop our skillset in ARM board development and 8051.

7 REFERENCES

[1] https://www.kbb.com/car-advice/blind-spot-monitors/
[2] How to Make IR Sensor : 4 Steps (with Pictures) - Instructables
[3] OLED display using I2C with STM32 » ControllersTech
[4] Bitmap converter for mono and color LCD displays (dxp.pl)
[5] https://www.gotronic.fr/pj2-mf52type-1554.pdf
[6] https://www.st.com/en/microcontrollers-microprocessors/stm32f411ve.html
[7] https://www.st.com/resource/en/application_note/an3371-using-the-hardware-realtime-clock-rtc-in-stm32-f0-f2-f3-f4-and-l1-series-of-mcus-stmicroelectronics.pdf
[8] Senior Driving - Division of Motor Vehicles (de.gov)

8 DIVISION OF LABOR

Feature / Design	Contributor
Hardware Procurement	Rishikesh Sundaragiri & Suraj Ajjampur
Report, schematics and PPT's preparation	Rishikesh Sundaragiri & Suraj Ajjampur
IR Sensor design	Rishikesh Sundaragiri & Suraj Ajjampur
Temperature Sensor Design	Rishikesh Sundaragiri
RTC Bare Metal implementation	Rishikesh Sundaragiri
Calendar and time algorithm to current time, date, month, and year from RTC registers	Rishikesh Sundaragiri
ADC Bare metal for Temperature Sensor and Real time temperature calculation	Rishikesh Sundaragiri
GPIO bare metal for IR detection and buzzer alert	Rishikesh Sundaragiri
Free RTOS Implementation	Rishikesh Sundaragiri
OLED Animations using SSD OLED library.	Suraj Ajjampur
I2C using HAL for interfacing OLED to STM32	Suraj Ajjampur

9 APPENDIXES

Several appendices have been attached to this report in the order shown below

9.1 Appendix - Bill of Materials (BOM)

PARTS	QUANTITY	COST
OLED	1	7.29
OP-AMP (LM358)	2	1.27
NTC (10k)	4	1.94
Resistor (10kohms)	2	0.1
Resistor (100ohms)	2	0.1
Resistor (330ohms)	2	0.1
Variable Resistor(0-10k)	2	2
IR Led	2	0.62
Photo Diode	4	3.09
Jumpers	NA	6.98
Buzzer	2	1
Bread Boards	4	6.64
TOTAL		**38.42**

9.2 Appendix – Schematics

9.3 Appendix – Bare Metal Source Code

9.4 Appendix – FreeRTOS Source Code

9.5 Appendix – Data Sheets and Application Notes