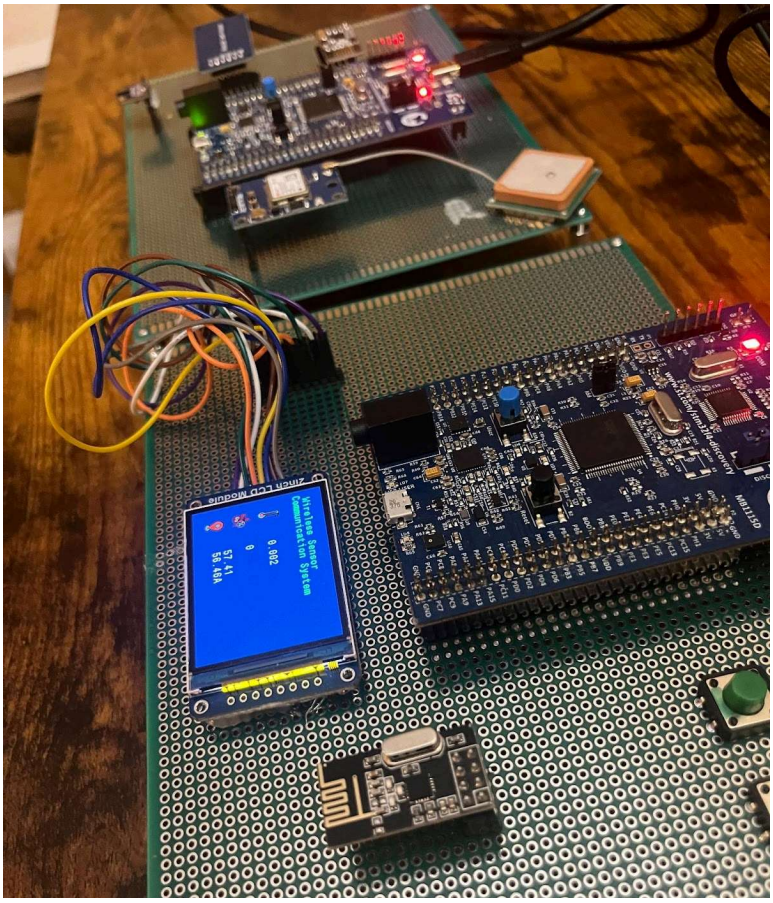
ESD FINAL PROJECT REPORT

Wireless Sensor Communication System



Abhirath Koushik and Nalin Saxena

12/15/2024

ECEN 5613

EMBEDDED SYSTEMS DESIGN

FALL 2024

TABLE OF CONTENTS

1. INTRODUCTION

The motivation behind this project is to learn how two microcontrollers/embedded systems communicate with each other wirelessly. Wireless communication between systems is a very important aspect of modern smart devices.

To simulate a real world application we have added 2 sensors namely-spo2 sensor (Max30102) and a gps module to generate real time data. As a fail-safe mechanism in case the transmission fails , the data is continuously logged to an external memory device (an sd card in this case). This project helped us understand the various key aspects of wireless communication and sensor interfacing. We were also able to understand the importance of good software design and how challenging it is to put individual working pieces together to form a cohesive and robust system .

This report outlines the hardware and software design, various challenges faced while developing the system and also discusses tradeoffs and testing details.

2. SYSTEM OVERVIEW

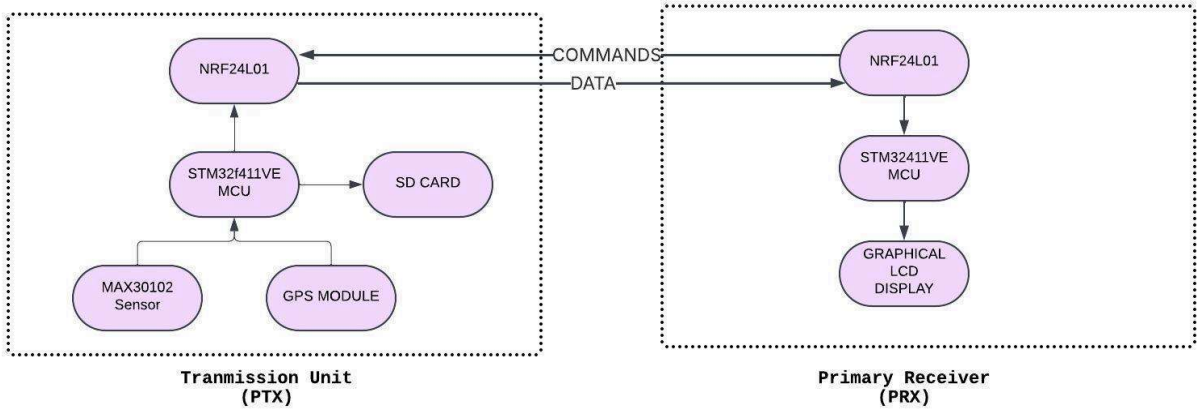


Figure 1: Overview Block Diagram

The system has two main units, the Primary Transmission Unit or **PTX** and the primary receiver or PRX. The **PTX** sends the sensor data to the **PRX** unit and in return the **PRX** responds with a custom acknowledgement message which encodes an instruction which can be used to disable/enable the sensors individually.

3. TECHNICAL DESCRIPTION

The technical description will involve deeper description of the board design, hardware and software design.

3.1. BOARD DESIGN

We have divided the Hardware onto 2 Prototype Boards, for the Transmitter and Receiver Units. All the connections have been wire wrapped on the bottom-side of the prototype boards.

In the images below, we can see the components of the Transmitter Unit,

- STM32f411VE Board
- NRF24L01 Transmitter Module
- NEO 6M GPS Module
- MAX30102 HeartRate and SpO2 Sensor
- SD Card Module
- Status LEDs

The Components on the Receiving Unit,

- STM32f411VE Board
- NRF24L01 Receiver Module
- Graphical LCD with ST7789V controller
- Status LEDs
- Command Push Buttons

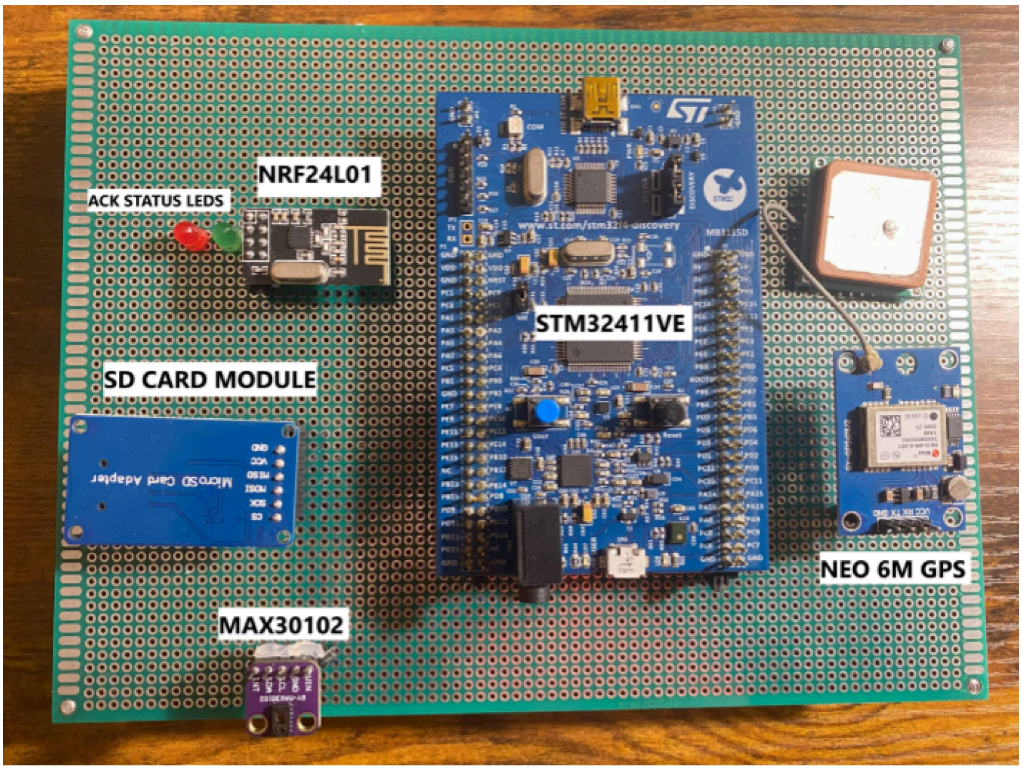


Figure 2: Transmitter Board

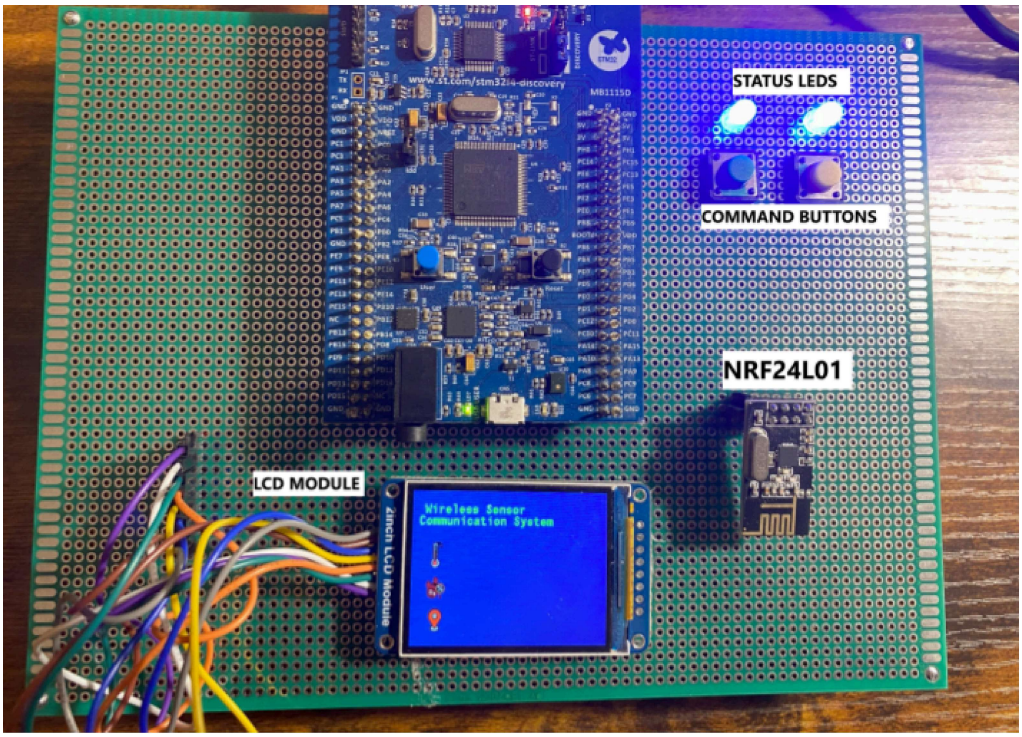


Figure 3: Receiver Board

3.2. NEW HARDWARE ELEMENT - NRF24L01

The NRF24L01 is a very low cost transceiver module developed by Nordic Semiconductor . We have used SPI (serial peripheral interface) to interface this to both the nodes in our project. The pinout is described by the Image below. (Figure 4: source)

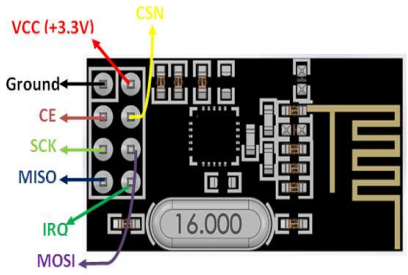


Figure 4: NRF MODULE

To verify if the module was healthy and functional we tested it with an arduino based library (test library). Basic spi drivers were then written for the stm32 board however due to lack of a typical WHO_AM_I_REGISTER , another option to check if the code was functioning correctly was to read the reset values of registers. After verifying the values against the datasheet for a few registers we conclude that the basic code was functioning as expected.

The nrf module supports SPI communication up to 10Mhz , however while testing we had to set clock speed down 1.5 Mhz to ensure reliable communication with the module.

The module supports various modes of configuration such as static payload width, dynamic payload width and other advanced features such as auto acknowledgement with custom payload and more.

11	RX_PW_P0				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P0	5:0	0	R/W	Number of bytes in RX payload in data pipe 0 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes

Figure 5: static payload setup

We chose an iterative approach while working on this, and first implemented a simple one way communication between the nodes with static payload. This has to be pre configured on both the receiver and transmission side with the above command (page-57).

It was far more challenging to implement bidirectional communication between the two nodes. The data sheet outlines a protocol called Enhanced ShockBurst™ . “ An Enhanced ShockBurst™ packet transaction is always initiated by a packet transmission from the PTX, the transaction is complete when the PTX has received an acknowledgment packet (ACK packet) from the PRX. The PRX can attach user data to the ACK packet enabling a bi-directional data link. ” (following excerpt from section 7.2)

Essentially after a packet transmission the Primary Transmitter node goes into receiving mode and expects an acknowledgement message from the primary receiver. A simplified overview of the process is explained by the block diagram below.

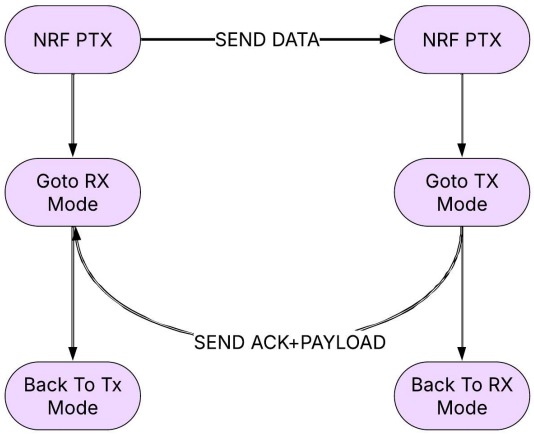


Figure 6: Simplified Bi-directional communication

The datasheet has an elaborate diagram regarding the expected PTX and PRX behaviour as per the Enhanced ShockBurst™ protocol. For the sake of simplicity below is a simplified version on the **PTX** side of the same which also includes a few key configuration steps which are essential.

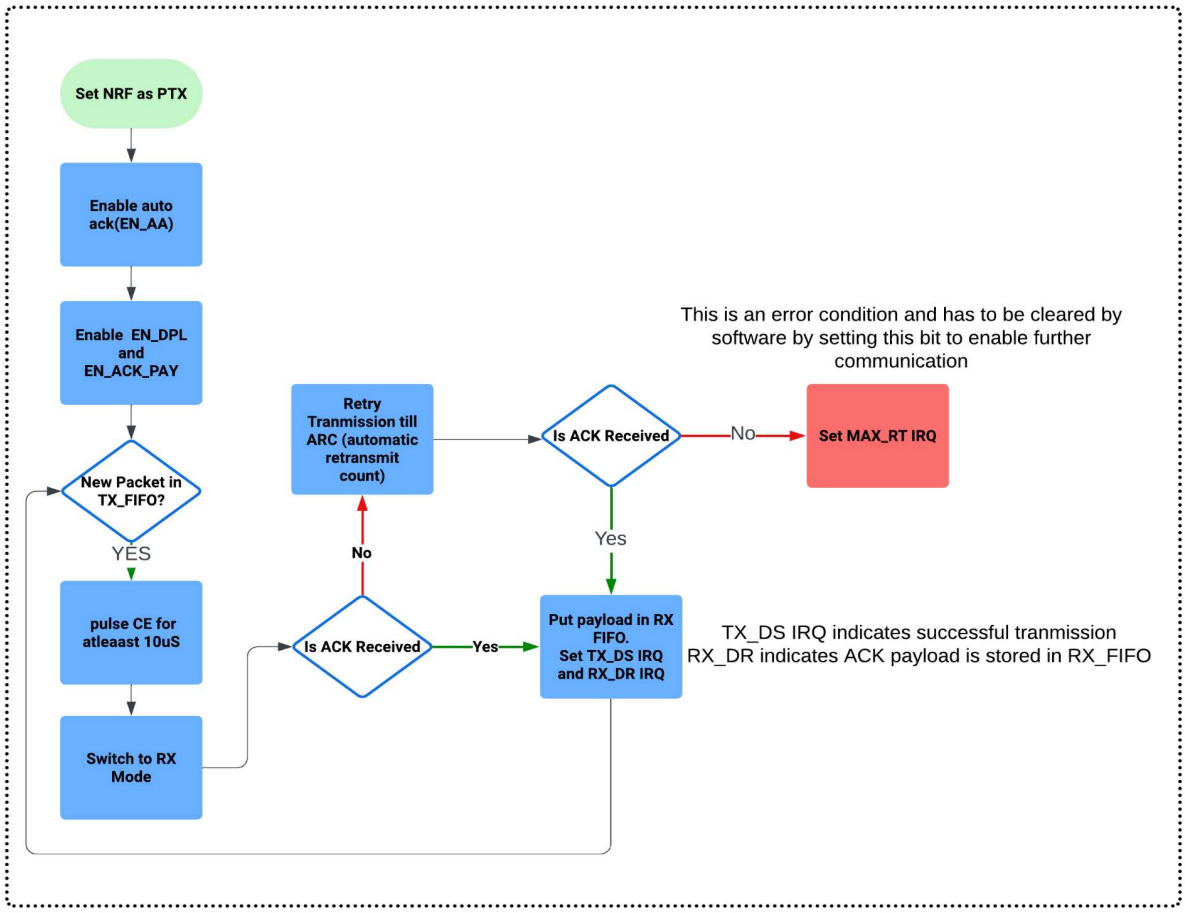


Figure 7: Simplified ESB PTX operation

First it's essential to configure the module as a transmitter; this can be done from the **CONFIG** register. Then Automatic Acknowledgment has to be enabled with the **EN_AA** register. Under the **FEATURE** register we need to enable **EN_DPL** and **EN_ACK_PAY.** We then check if new data is in NRF's TX_FIFO the actual transmission happens when the **CE**(chip enable) pin is pulsed for at least 10uS, in code this is achieved by a simple delay function using systick timer. Post transmission the module goes to receiver mode and expects an ACK. If it is received then 2 bits **TX_DS** and **RX_DR** are set in the **STATUS** register(0x07) as shown below. **TX_DS** indicates that an ack is successfully received and **RX_DR** indicates new data in the RX_FIFO of the nrf. The ack message can be read using the **R_RX_PAYLOAD** command.

In case ack message is not received even after a pre-configured **ARC-** Auto Retransmit Count (this is set under **SETUP_RETR 0x04** register) then an error bit MAX_RT is set indicating max re-transmission have been reached, we must clear this bit via software to continue communication.

07	STATUS				Status Register (In parallel to the SPI command word applied on the **MOSI** pin, the STATUS register is shifted serially out on the **MISO** pin)
	Reserved	7	0	R/W	Only '0' allowed
	RX_DR	6	0	R/W	Data Ready RX FIFO interrupt. Asserted when new data arrives RX FIFO[c]. Write 1 to clear bit.
	TX_DS	5	0	R/W	Data Sent TX FIFO interrupt. Asserted when packet transmitted on TX. If AUTO_ACK is activated, this bit is set high only when ACK is received. Write 1 to clear bit.
	MAX_RT	4	0	R/W	Maximum number of TX retransmits interrupt Write 1 to clear bit. If MAX_RT is asserted it must be cleared to enable further communication.
	RX_P_NO	3:1	111	R	Data pipe number for the payload available for reading from RX_FIFO 000-101: Data Pipe Number 110: Not Used 111: RX FIFO Empty
	TX_FULL	0	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.

Figure 8: Status Register NRF 24L01

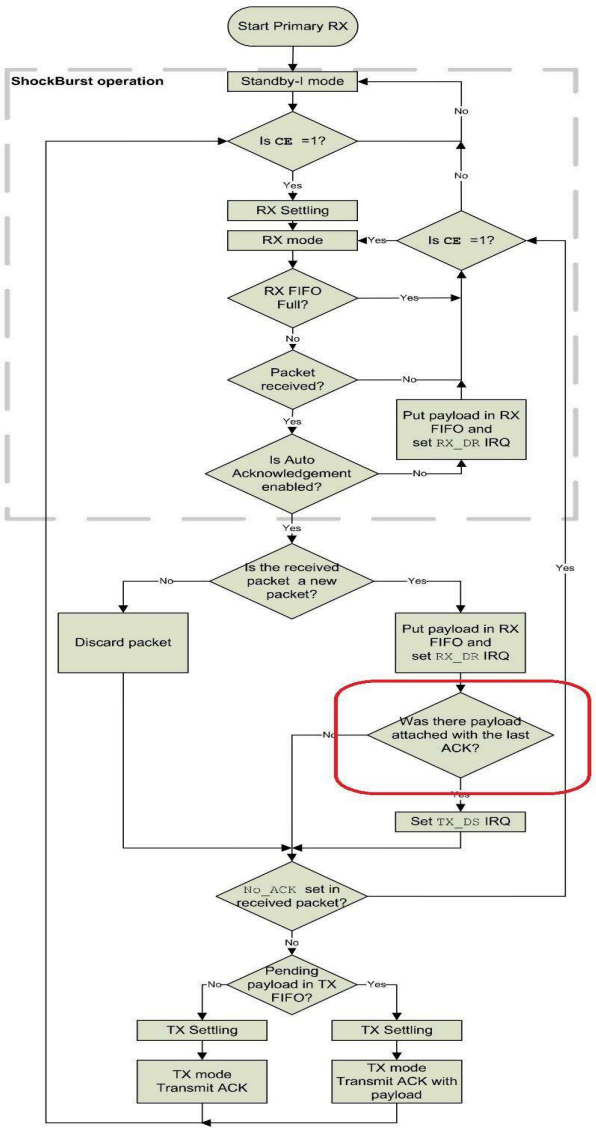


Figure 9: PRX ESB operation

The expected behaviour and configuration of the PRX node is given on the left (Source -7.6.2). This was far more complex to understand and implement in code , it was highly unclear where the receiver should load the acknowledgment after the reception is done, the ambiguous part is marked in **red.** After a few iterations of experimenting we were unable to load a custom ack message.

We decided to contact Nordic via the Nordic Devzone . (support-ticket link). They replied with an application note-**nAN24-12** and HAL based code examples to implement the same.

Below is the relevant code excerpt from the application note-

```
void device_prx_mode_pl(void) {
    CE_HIGH(); // Set Chip Enable (CE)
pin high to enable receiver

    while (true) {
      // Setup and put the ACK payload
on the FIFO
      pload_pl[0] = 0;

      if (B1_PRESSED()) {
        pload_pl[0] = 1;
      }

      hal_nrf_write_ack_pload(0,
pload_pl, RF_PAYLOAD_LENGTH);
```

The key point from the application note is that the receiver must pre-load the acknowledgment payload with the **W_ACK_PAYLOAD** command in the receiver's **TX_FIFO before a message is received**. After incorporating the above change we were

able to establish bi-directional communication . For the use case of our project the receiver's acknowledgement messages have a command for the transmitter which can be changed dynamically altered according to the status of the command buttons (*see figure 3*)

3.3. NEW HARDWARE ELEMENT - MAX30102

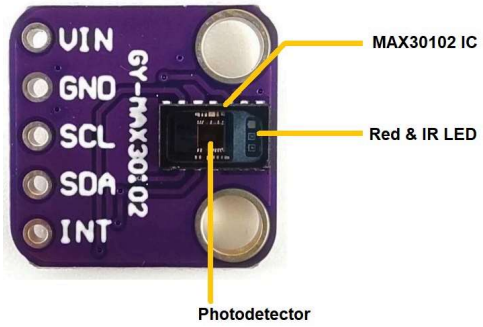


Figure 10: MAX30102

The MAX30102 is an integrated heart rate and spo2 module based on the i2c protocol. There are two on board leds - RED Led and an IR Led. It works on the principle of the amount of light red and ir absorbed and reflected by the user's skin

The sensor incorporates infrared and red LEDs, along with a photonic filter, to generate appropriate light that penetrates the skin and red-colored tissues. The infrared LED penetrates deeper into the skin, while the red LED penetrates further away. When the light is projected through the skin, it encounters reflection from the flowing blood in the blood vessels beneath the skin. (following excerpt taken from link)

The MODE_CONFIG_REG decides the operating mode of the chip we have selected the spo2 mode. We also need to configure the intensity of both red and Infrared leds, while experimenting with the sensor we found it is very sensitive to surrounding light.

Code Initialization routine for the chip is given below

```
void MAX30102_init(){
      I2C_init_config(); //initialize i2c
          // Reset the sensor
          MAX30102_WRITE_REGISTER(MODE_CONFIG_REG, 0x40); // Reset the device
          //led brightness as 60
```

```
            MAX30102_WRITE_REGISTER(LED1_PA_REG, 0x3c); //power set as 60
            MAX30102_WRITE_REGISTER(LED2_PA_REG, 0x3c); // power set as 60
            //led mode 3 both ir and red led are needed
            MAX30102_WRITE_REGISTER(MODE_CONFIG_REG, 0x3); // spo2
            //sample average as 4 and enable fifo rollover 0x50 0101 0000
            MAX30102_WRITE_REGISTER(FIFO_CONFIG_REG, 0x50); //
            //spo2 config adc range as 4096  samples as 100 led pulse as 411
            MAX30102_WRITE_REGISTER(SPO2_CONFIG_REG, 0x27);
            // FIFO pointers
            MAX30102_WRITE_REGISTER(FIFO_READ_PTR, 0x00); // Reset FIFO write
pointer
            MAX30102_WRITE_REGISTER(FIFO_WR_PTR, 0x00); // Reset FIFO read pointer
}
```

The data sheet explains reading of the raw values of the IR and reads as “Each data sample in SpO2 mode comprises two data triplets (3 bytes each), To read one sample, requires an I2C read command for each byte. Thus, to read one sample in SpO2 mode, requires 6 I2C byte reads. The FIFO read pointer is automatically incremented after the first byte of each sample is read. ”

Below is a logic analyzer screenshot of 6 burst reads from the MAX30102 chip

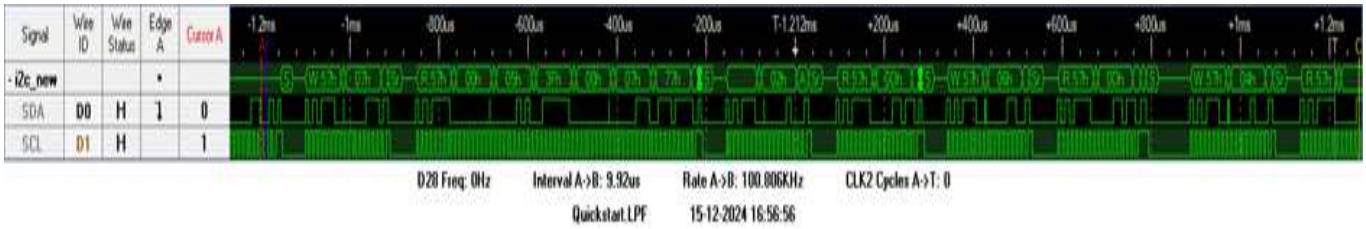


Figure 11: Logic Analyzer snippet

To calculate the values from the raw values of the IR and Led buffer I have utilized the following library provided by Sparkfun and have duly credited them in the code.

We have also utilized the onboard temperature sensor on the chip which was fairly straightforward and just requires a simple configuration and 2 registers reads from below register. The integer and fractional values are stored in different registers and just need to be read and combined.

3.4. NEW HARDWARE ELEMENT - NEO 6M GPS MODULE

The Neo 6M GPS Module is a compact and versatile GPS receiver designed for easy integration into various projects. It features the U-Blox Neo 6M chipset, which is capable of tracking up to 22 satellites over 50 channels .

Pin Number	Pin Name	Description
1	VCC	Power supply (5V)
2	GND	Ground
3	TX	PA10 of MCU
4	RX	PA 9 of MCU

Figure 12: Neo 6M GPS Module and Connections to the MCU

This module works through the UART interface at 9600 baud rate by design. We have used USART1 to connect it with our STM board. The connections made are shown in the table above.

The data from this module is obtained in the NMEA (National Marine Electronics Association) format. It is a standard format which displays the time, date, latitude, longitude, altitude and estimated velocity.

```
$GPGGA,110617.00,41XX.XXXXX,N,00831.54761,W,1,05,2.68,129.0,M,50.1,M,,*42
$GPGSA,A,3,06,09,30,07,23,,,,,,,,4.43,2.68,3.53*02
$GPGSV,3,1,11,02,48,298,24,03,05,101,24,05,17,292,20,06,71,227,30*7C
$GPGSV,3,2,11,07,47,138,33,09,64,044,28,17,01,199,,19,13,214,*7C
$GPGSV,3,3,11,23,29,054,29,29,01,335,,30,29,167,33*4E
$GPGLL,41XX.XXXXX,N,00831.54761,W,110617.00,A,A*70
$GPRMC,110618.00,A,41XX.XXXXX,N,00831.54753,W,0.078,,030118,,,A*6A
$GPVTG,,T,,M,0.043,N,0.080,K,A*2C
```

Figure 13: GPS Data in the NMEA Format

All NMEA messages start with the '$' character and each field is separated by a comma.

```
$GPGGA,110617.00,41XX.XXXXX,N,00831.54761,W,1,05,2.68,129.0
,M,50.1,M,,*42
```

If we consider a sample line in the NMEA format,

- GP indicates that it is a GPS position
- GGA indicates GPS Fixed Data
- 1100617 indicates the time at which the location was taken, 11:06:17 UTC
- 41XX.XXXXX,N – Latitude 41 deg XX.XXXXX' N
- 00831.54761,W – Longitude 008 deg 31.54761' W

Based on this, we are parsing the NMEA format for the Latitude and Longitude information.

The main issue we faced is of random characters and Non-ASCII characters in the GPS Buffer and we were not able to parse for the required information. To overcome this, I have used an NMEA library and Ring Buffer approach developed by Controller Tech. In this approach, we wait till the required character (comma) is obtained in the GPS buffer and then traverse through the buffer based on the NMEA structure. Once we reach the position of the latitude, we extract the required information.

```
int decodeGGA (char *GGAbuffer, GGASTRUCT *gga)
{
    inx = 0;
    char buffer[12];
    int i = 0;
    while (GGAbuffer[inx] != ',') inx++;  // 1st ','
    inx++;
    while (GGAbuffer[inx] != ',') inx++;  // After time ','
    inx++;
    while (GGAbuffer[inx] != ',') inx++;  // after latitude ','
    inx++;
    while (GGAbuffer[inx] != ',') inx++;  // after NS ','
    inx++;
    while (GGAbuffer[inx] != ',') inx++;  // after longitude ','
    inx++;
    while (GGAbuffer[inx] != ',') inx++;  // after EW ','
    inx++;  // reached the character to identify the GPS Position Fix
    if ((GGAbuffer[inx] == '1') || (GGAbuffer[inx] == '2') || (GGAbuffer[inx] == '6'))   // 0 indicates no fix yet
    {
        gga->isfixValid = 1;   // fix available
        inx = 0;   // reset the index and extract information now
    }
```

Figure 14: Code Snippet for extracting information from GPS Buffer

A ring buffer is also used to receive the GPS Data as UART data arrives asynchronously, at any point of time. This approach ensures that incoming data is not lost and can be processed at a later point by the UART interrupt handler.

Once the required latitude and longitude information is extracted, it is added to the payload that will be transmitted wirelessly to the receiver.

3.5. NEW HARDWARE ELEMENT - SD Card Module

The SD Card Module is used to interface an SD (Secure Digital) Card to a microcontroller or embedded system. In our project, we are using the SD Card to log our payload information, consisting of temperature, SpO2 and GPS coordinates. This is used as a fail-safe mechanism to ensure that this data is not lost, even if it did not get transmitted correctly to the receiver.

SD Card Reader	STM32 MCU
VCC	5V
GND	GND
CS	PB12
SCK	PB10
MISO	PC2
MOSI	PC3

Figure 15: SD Card Module and Connections to the MCU

We have used SPI as the mode of communication to this module. The connections to the MCU are made as shown in the diagram above.

The SD Card uses the FAT32 file system. FAT stands for File Allocation Table. FAT32 is a disk format or filing system used to organise the files on a disk drive. The disk drive is divided into addressable chunks called sectors (with a size of 32 bits) and the file allocation table is created to identify each of the files on the drive.

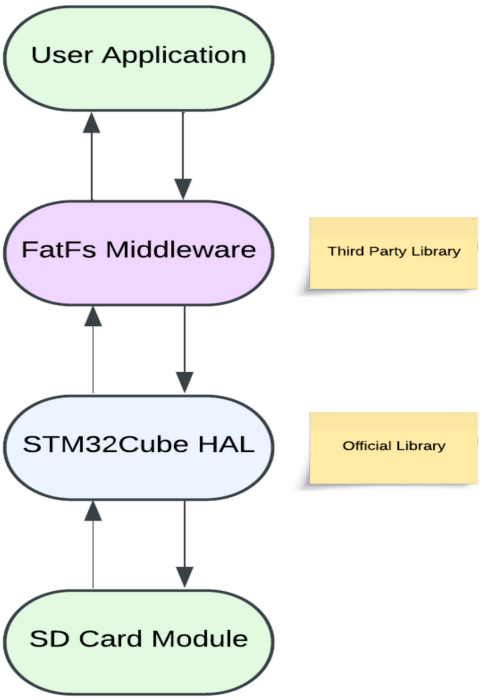


Figure 16: Flowchart of FatFs and HAL for SD Card Module

The adjacent block diagram illustrates how we have interfaced the SD Card Module in our project.

FatFs is an open source software library that can translate FAT file structures in memory into actual files. It is built into STM32CubeIDE as a Middleware that can be integrated with STM HAL functions. The SD Card Disk operations included in FatFs include

- Disk_status: Obtain Device Status
- Disk_initalize: Initialize the SD Card Device
- Disk_read: Read Data
- Disk_write: Write Data
- Disk_ioctl: Device Dependent Functions

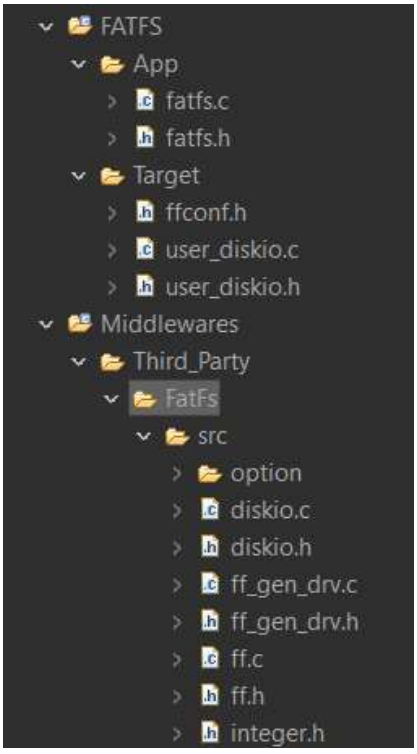


Figure 17: Snippet of FatFs integrated with our existing project

FatFs can be selected directly in the STM32CubeIDE .ioc file.

Once selected as “User Defined”, the FatFs code is automatically integrated with our existing project as shown below.

I have integrated the code developed by Embetronicx for this purpose. This uses SPI2 through HAL to interface the various SD Card functions.

Once this is done, the functions in “user_diskio.c” are modified to execute our defined functions as shown in the snippet below,

```
DSTATUS USER_initialize (
    BYTE pdrv           /* Physical drive nmuber to identify the drive */
)
{
  /* USER CODE BEGIN INIT */
    return SD_disk_initialize(pdrv); // Changes made here by Abhirath Koushik
  /* USER CODE END INIT */
}

/**
  * @brief  Gets Disk Status
  * @param  pdrv: Physical drive number (0..)
  * @retval DSTATUS: Operation status
  */
DSTATUS USER_status (
    BYTE pdrv       /* Physical drive number to identify the drive */
)
{
  /* USER CODE BEGIN STATUS */
    return SD_disk_status(pdrv); // Changes made here by Abhirath Koushik
  /* USER CODE END STATUS */
}
```

After this, the FatFs libraries can be used to execute the SD Card operations, including f_mount, f_open, f_puts and f_close used to write the data onto a text file and save it to the SD Card. The following code snippet performs this operation,

```
  // Mount the SD Card
  file_result = f_mount(&Fat_handler, "", 1);
  if (file_result != FR_OK)
  {
    printf("No SD Card found : (%i)\r\n", file_result);
    break;
  }
  print_success("\n\rSD Card Mounted Successfully!\r\n");
  // Open the File in Append Mode to Write Data
  file_result = f_open(&file_var, "logger.txt", FA_OPEN_APPEND| FA_WRITE);
  if(file_result != FR_OK)
  {
    printf("File creation/open Error : (%i)\r\n", file_result);
    break;
  }
  printf("\n\rWriting data to logger.txt!\r\n");
  strcat(log_str, "\n\r"); // Adding \n\r to add the data to new line in the file
  // Write the data into the text file
  f_puts(log_str, &file_var);
  // Close the text file after adding the data
  f_close(&file_var);
} while(false);
// Unmount the SD Card
f_mount(NULL, "", 0);
print_success("\n\rSD Card Unmounted Successfully!\r\n");
```

3.6. NEW HARDWARE ELEMENT - Graphical LCD Display

A Graphical LCD Display is a versatile output device that can render complex visuals such as text, graphics, and animations, making it ideal for presenting detailed information. In our project, it is used on the receiver side to display payload data, including Temperature, SpO2 levels, and GPS coordinates. We are using a 240x320 resolution LCD with ST7789V controller.

This LCD communicates through SPI and we have developed its functions and operations through Bare Metal programming. The connections to the MCU are as shown in the table below.

ST7789 LCD	STM32 MCU
VCC	3.3V
GND	GND
DIN/MOSI	PA7
CLK	PA5
CS	PB6
DC	PB5
RST	PB7
BL	PB4

Figure 18: Graphical LCD and Connections to the MCU

This LCD uses a four-wire SPI communication which ensures fast communication speeds. This includes the DIN/MOSI, CLK, CS and DC Pins.

The diagram below represents the timing diagram with the various LCD Signals.

- RESX - Reset signal for the LCD. When its High, its ready for operation
- CSX - Chip Select (Active Low). The SPI communication is enabled when this pin goes low.
- DC - Data/Command. This is used to distinguish between a data or a command operation. 1 implies data and 0 implies command.
- SDA - Serial Data. Transmits the pixel data in 16-bit format and is made up of RGB components.
- SCL - Serial Clock. Clock signal is used to synchronize the data on SDA. Its rising edge determines when each bit of data is sampled.

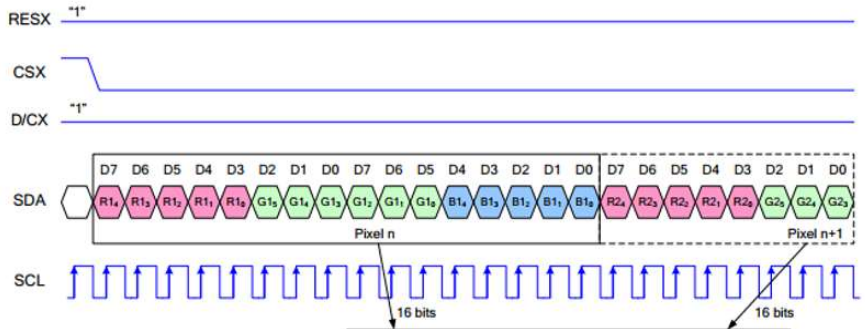


Figure 19: Timing Diagram for SPI Communication in LCD

As we had developed the SPI communication for the NRF modules, the same functions were used and integrated while calling the LCD initialization and data transfer functions. In the code snippet below, we can see that the SPI Initialization followed by the Initialization of LCD Pins (DC, RST, CS and BL).

```
void LCD_spi_init(void)
{
   SPI_INIT(); // Calling the common SPI Initialization Function
   /* Configure DC, RST, CS, and BL pins */
   RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN; // Enable GPIOB clock for the Pins
   GPIOB->MODER &= ~(GPIO_MODER_MODE5_Msk | GPIO_MODER_MODE6_Msk |
GPIO_MODER_MODE7_Msk | GPIO_MODER_MODE4_Msk); // Clear the mode bits for
the Pins
   GPIOB->MODER |= (GPIO_MODER_MODE5_0 | GPIO_MODER_MODE6_0 |
GPIO_MODER_MODE7_0 | GPIO_MODER_MODE4_0); // Set the Pins to output mode
}
```

The main functions of the LCD are the data write and the command writes. In the code snippet below, we can see that the SPI_TX_MULTI function is a common function that is called for any SPI Transmission. DC Low here indicates that a command is being written to the LCD.

```
void LCD_write_command(uint8_t cmd)
{
      GPIOB->BSRR = GPIOB_BSRR_RESET_DC; // DC Low indicating Command
      GPIOB->BSRR = GPIOB_BSRR_RESET_CS; // Chip Select Low
      SPI_TX_MULTI(&cmd,  1);
      GPIOB->BSRR = GPIOB_BSRR_SET_CS;   // Chip Select High
}
```

While writing data to the LCD, the same SPI_TX_MULTI function is called but as DC is High, it indicates that data is being written to the LCD.

```
void LCD_write_data(uint8_t *buff, uint32_t buff_size)
{
      GPIOB->BSRR = GPIOB_BSRR_SET_DC;   // DC High indicating Data
      GPIOB->BSRR = GPIOB_BSRR_RESET_CS; // Chip Select Low
      SPI_TX_MULTI(buff, buff_size);
      GPIOB->BSRR = GPIOB_BSRR_SET_CS;   // Chip Select High
}
```

Based on these functions, we were able to derive the other functions to fill the LCD screen with color and display a string.

Additionally, we explored creating custom icons for temperature, SpO2 and GPS coordinates. We were able to use an online tool [LCD Image Convertor] to obtain the bitmap array for a given image. This array is then used to draw pixels at each of the positions in the LCD to draw the complete custom image.

Figure 20: Text and Custom Icons on the Graphical LCD

Overall, the most important new hardware design elements in our project are summarized below,

1. STM32f411VE MCU: This is our central processing unit for both the transmitter and receiver units. It provides all the required functionality with GPIO, SPI, I2C and UART interfaces for the communications.

2. NRF24L01 Transceiver Module: This is a very low cost transceiver module developed by Nordic Semiconductor, interface through SPI. This is the module used for the bi-directional wireless communication.

3. MAX30102 Heart Rate and SpO2 Sensor: This module provides real-time health monitoring data (heart rate and SpO2 levels) with high sensitivity. We have also configured this sensor to obtain the temperature information.

4. NEO 6M GPS Module: Enables geolocation functionality by receiving satellite data in NMEA format, with robust parsing methods like ring buffers to handle asynchronous data streams.

5. Graphical LCD with ST7789V Controller: Used on the receiver side to display data visually, including temperature, SpO2 levels, and GPS coordinates.

6. SD Card Module: Implements fail-safe data logging using the FAT32 file system via SPI. Ensures critical payload data is preserved even if wireless transmission fails.

7. Status LEDs and Command Buttons: Status LEDs are used as an indication regarding the acknowledgement after a transaction. Additionally, the command button can be used to disable/enable the sensors remotely.

3.7. SOFTWARE SECTION

The software section can be divided into 4 layers as explained by the diagram below.

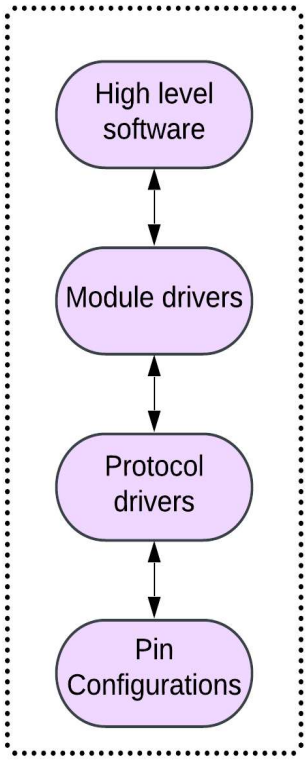


Figure 21: Software layers

We will discuss these layers in a bottom up fashion.

a. Pin Configurations - These modules are responsible for configuring the hardware to support much more complex functions. These modules generally include configuring pins to input/output or alternate functionality and other fundamental aspects. Below is an example of the usart_init() function which describes how GPIOs PA2, PA3 would be configured in Alternate functionality mode and further selection of the correct AFR functionality.

```
void usart_init() {
   RCC->APB1ENR |= RCC_APB1ENR_USART2EN; // Enable USART2 clock
   RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;  // Enable GPIOA clock
   // Set PA2 and PA3 as alternate function
   GPIOA->MODER |= (ALTERNATE_MODE << GPIO_MODER_MODER2_Pos);
   GPIOA->MODER |= (ALTERNATE_MODE << GPIO_MODER_MODER3_Pos);
   // Set PA2 to USART_TX and PA3 to USART_RX
   GPIOA->AFR[0] |= (7 << GPIO_AFRL_AFSEL2_Pos); // TX alternate function
   GPIOA->AFR[0] |= (7 << GPIO_AFRL_AFSEL3_Pos); // RX alternate function
   // Enable USART, transmitter, and receiver
   USART2->CR1 |= USART_CR1_UE | USART_CR1_TE | USART_CR1_RE;
   // Set baud rate to 9600 (make sure BAUD_9600 is defined based on your
clock setup)
   USART2->BRR = BAUD_9600;
   // Test character output to verify USART setup
}
```

b. Protocol drivers - This module will have code specific to one protocol for example I2C, SPI etc. This layer serves as a base to write module specific drivers. For instance if 2 modules are SPI based they will have their own individual drivers, however internally they would be utilizing the same Protocol driver layer (with a different set of control signal pins of course). Below code shows function headers for the SPI library.

```
/*
* Function to write multiple bytes to spi1 interface
* Parameters:
*                   uint8_t *data_ptr  : pointer of data buffer
*           int size           : number of bytes to be written
* Returns:
*                  none
*/
void SPI_TX_MULTI(uint8_t *data_ptr, int size);
/*
* Function to read multiple bytes from spi1 interface
* Parameters:
*                   uint8_t *data_ptr  : pointer to data buffer
*           int size           : number of bytes to be read
* Returns:
*                  none
*/
void SPI_READ_MULTI(uint8_t *data_ptr, int size);
```

c. Module drivers - Module/Device drivers are built specific to each hardware module. This layer is used to interface the module with the mcu. One important aspect is that this layer should make sure that the correct timing considerations are met for the module in consideration as per the datasheet. Below is an excerpt from code developed for the MAX30102 module for a basic read and write operation to a register.

```
void MAX30102_WRITE_REGISTER(uint8_t register_Address, uint8_t data){
      I2C_START_COMS();
      uint8_t write_masked_address=0xAE;
      I2C_SEND_ADDRESS(write_masked_address); //same as AE
      I2C_WRITE_DATA(register_Address);
      I2C_WRITE_DATA(data);
      I2C_STOP_COMS();
}
void MAX30102_READ_REGISTER(uint8_t register_Address, uint8_t *recv_buff, uint8_t
recv_size){
      uint8_t write_masked_address=0xAE;
      uint8_t read_masked_address=0xAF; //should be AF
      I2C_START_COMS(); //start i2c
      I2C_SEND_ADDRESS(write_masked_address); //first send slave address
      I2C_WRITE_DATA(register_Address); //send register address
      I2C_START_COMS(); //repeated start
      I2C_READ(read_masked_address,recv_buff,recv_size); //this should be slave
with read
}
```

d. High level software: This layer includes higher level constructs such as state machines , getting temperature value from a module etc and printing it over the serial terminal. This would also include any code that the user would directly interact with. An example is shown below of a module developed to print output in different colours based on the string type.

```
/*
* prints message in green color, used to print success messages
*
* Parameters:
*                    char *str : pointer to message
*
* Returns:
*                   none
*/
void print_success(char *str);
```

```
/*
* prints message in yellow used to info  messages
*
* Parameters:
*                    char *str : pointer to message
*
* Returns:
*                   none
*/
void print_info(char *str);
/*
* prints message in red used to error  messages
*
* Parameters:
*                    char *str : pointer to message
*
* Returns:
*                   none
*/
void print_error(char *str);
```

3.8. STATE MACHINE AND PTX SOFTWARE FLOW

After getting all the individual modules working independently on PTX, we had to think of a way to put all the code together and decide on a software architecture for all the modules to work in a cohesive way. We decided on a state machine as we had good familiarity regarding the same having learnt that in **ECEN 5813** (Principles of Embedded Software Course).

Following code snippet explains the structure of the state machine along with corresponding state variables. The state machine model allows for more states to be easily added and is highly robust.

```
// an enum to define various state transitions
typedef enum
{
   STATE_HEART_DATA_ACQUIRE,
   STATE_GPS_DATA_ACQUIRE,
   STATE_PACKET_FORMATION,
   STATE_STORE_DATA_SD_CARD,
   STATE_TRANSMIT_NRF,
} current_state;
```

```
// state machine variables to store data from different modules
typedef struct
{
   current_state current_state;
   int8_t valid_heart_rate; //Unused
   int32_t spo2; //spo2
   int32_t heart_rate;//Unused
   int8_t valid_spo2; //status of spo2
   float gps_lat; //gps latitudeC
   float gps_long; //gps longitude
   char dir1; //direction 1
   char dir2; //direction 2
   float temperature; //temperature
   uint8_t nrf_payload[32]; //payload
} StateMachine;
```

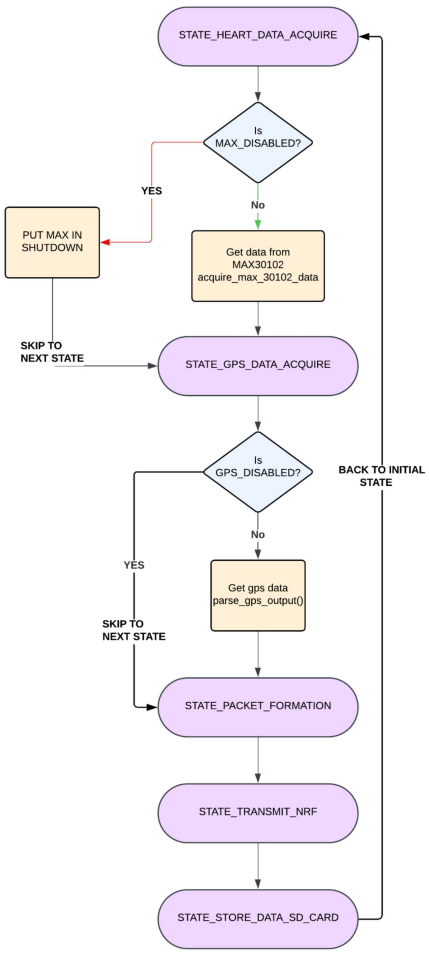


Figure 22: FSM Disagram for Transmitter Unit

The Diagram on the left explains the machine states and the various transitions. A brief description of the states is described below:

A. STATE_HEART_DATA_ACQUIRE: This is the initial
State, first a flag is which decides if the should go to a shutdown or continue normal operation. In normal operation acquire_max_30102_data() function is called along with the relevant state variables.

B. STATE_GPS_DATA_ACQUIRE : Checks for a flag If it's true then this state is skipped else gps data is acquired.

C. STATE_PACKET_FORMATION: Forms a data packet for the NRF module to transmit

D. STATE_TRANSMIT_NRF: The data packet nrf_payload[32] is transmitted using the nrf module, this state also checks for an acknowledgment from the receiver.

E: STATE_STORE_DATA_SD_CARD: Finally the data is logged into the SD CARD-MODULE and the state changes back to STATE_HEART_DATA_ACQUIRE.

3.9. PRX CODE FLOW (Receiver Code Flow)

The receiver code flow is much simpler and is outlined by the figure below.

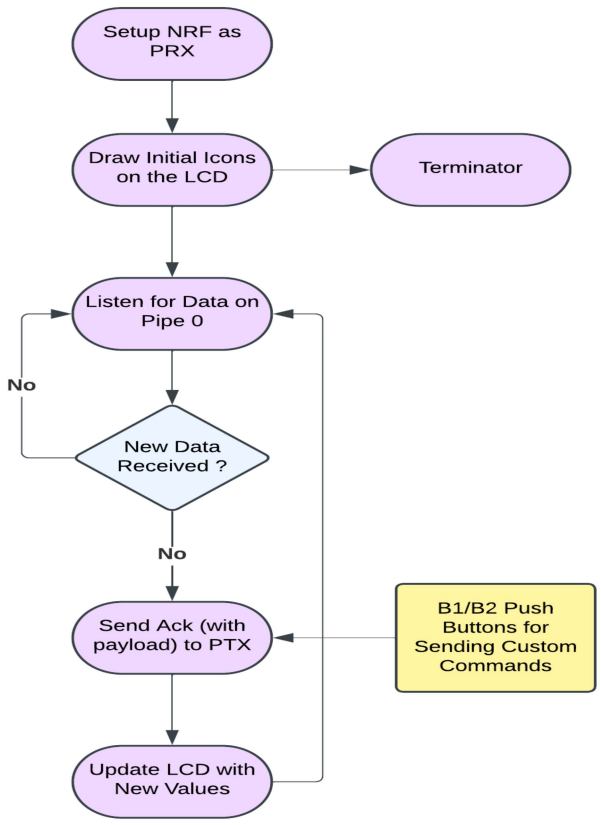


Figure 23: Flowchart for Receiver Unit

1. First the NRF is setup up in Receiver mode
2. Initial icons are drawn on LCD Screen.
3. We continuously listen for new data on pipe 0.
4. If new data is received we the nrf goes to Transmit mode and sends an ack message based on the status of B1/B2 command buttons.
5. The LCD screen is updated with new values and we go back to listening for data on pipe 0.

Overall, the most important new software design elements in our project are summarized below,

1. Layered Software Architecture: The software is organized into four well-defined layers which are the Pin configurations, Protocol drivers, Module drivers and the High-level software.

2. State Machine Architecture: A state machine design is used for managing the flow through multiple modules in the transmitter unit in a simplified manner. It also ensures scalability and structured handling of the hardware modules.

3. Code Reusability and Modularity: Using standardized functions (SPI_TX_MULTI, I2C_READ) allowed us to reuse the same functions across multiple modules, reducing redundant code.

4. Robust Error Handling and Acknowledgements: We have implemented the bi-directional communication in the NRF modules. This ensures that there is an acknowledgement received by both the modules after the transmissions, ensuring robust error checking.

In this project, we have authored about 1600 lines of code for the Transmitter Unit and about 1400 lines of code for the Receiver Unit.

3.10. TESTING PROCESS

We have employed multiple methods for testing each of the components in this project. At the very start, we tested each of the ordered components (NRF module, GPS module, MAX Sensor, SD Card module) with an Arduino to verify that the hardware was functioning correctly.

As we know that NRF, SD Card and Graphical LCD all work through the SPI interface, we used the Logic Analyzer with the SPI Interpreter to ensure that we were getting the correct output for read and write operations.

Additionally, we have also used the Logic Analyzer to read the Initialization Registers values from the NRF Module. A snippet is shown below,

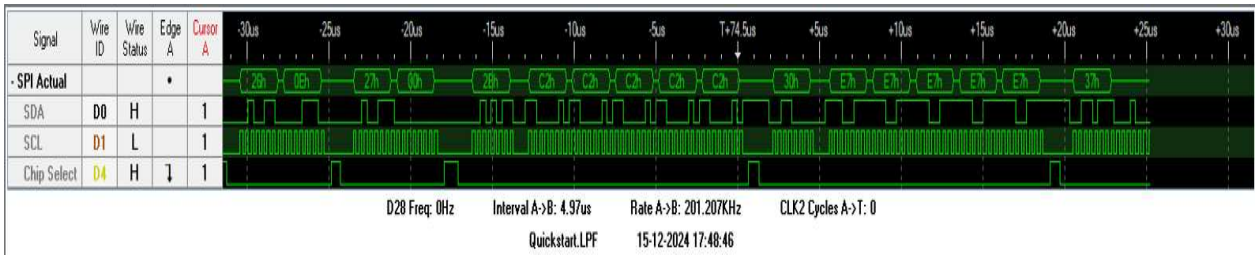


Figure 24: SPI Data Read of NRF Initialization Registers

Similarly, we have used the I2C Interpreter of the Logic Analyzer to test the Max sensor. A snippet of I2C Read on this sensor is shown below,

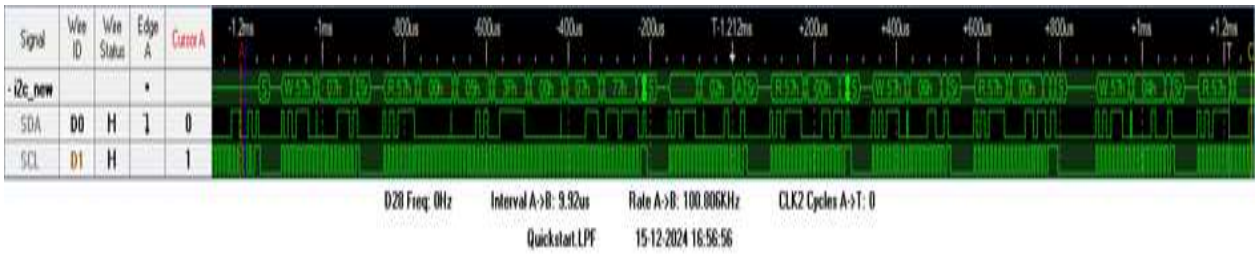


Figure 25: 6 I2C Reads on the Max Sensor

We have also implemented the Status LEDs with the NRF Modules. As it is difficult to debug with wireless modules, we used this approach to indicate if an acknowledgement is received back or not, after a transmission.

After completing the software testing, we practically checked the range of the NRF modules to communicate wirelessly. We observed that the communication was successful when the two units were held up to 120m apart. Here too, the presence of the Status LEDs were an immediate indication if the communication was successful or not.

4. RESULTS AND ERROR ANALYSIS

We were able to successfully demonstrate a wireless sensor communication system!

Through our approach, the data from the Max sensor and the GPS module was successfully transmitted wireless through the NRF Module to the receiver, which was able to display the information on a Graphical LCD. The same information was also stored into an SD Card successfully.

From the Project Deliverables, we were able to complete most of them, including our bonus goal.

- Implementation of Bi-directional communication with NRF ✅
- The transmitter should also be able to backup and log the data on a external memory device (SD Card) ✅
- Implementation of Battery Charging Circuit to power the Transmission Unit components ❌
- The receiver should be capable of displaying this information on a display module ✅
- The receiver should also be able to send commands to the Transmission Unit to modify the data rate, enable/disable specific sensors. (bonus goal) ✅

However, we did perform an error analysis in our project and observed some unexpected behaviors as listed below,

- Delay when the Push buttons to disable the sensors are pressed: As this disable operation is being executed remotely, we believe this delay is because of the packets that are already in the queue for transmission. Hence, the disabled payload gets into the queue only after the previous packets causing the delay.

- Surrounding Light affects the Max Sensor Readings: As this sensor works on the principle of light red and ir absorbed and reflected by the user's skin, a strong ambient light can saturate the detector and mask the reflected light from the user. This led to inaccurate readings during our testing.

- Delay in the GPS Module for Fix and Inaccurate Locations: The Neo 6M GPS Module takes time (~5mins) in a closed room to acquire the position fix. Hence, testing its values was a time-consuming process. We were also not able to work in the ESD Lab as the GPS Fix was not obtained at the Basement level.

5. CONCLUSION

This project gave us a unique opportunity to explore a topic of our interest and develop a unique and interesting project. A lot of skills learnt during the various lab assignments played a key role in overcoming obstacles. Debugging tools such as logic analyzers, and stm32-st link debugger gave us a deeper understanding of protocols and timing issues.

Aside from all the technical aspects it also taught us other skills such as time management, team working and creative thinking.

6. FUTURE DEVELOPMENT IDEAS

During the PDR we had added the battery charging circuit as a milestone and deliverable, however due to lack of time we were unable to complete it. Implementing a custom battery charging pcb would be an interesting addition to the current implementation.

To make the boards more portable we can port the project to a smaller profile mcu such as stm32 blue-pill board.

7. ACKNOWLEDGEMENTS

We would like to express our gratitude towards Professor Linden McClure for structuring this course in a very practical way and providing us with the opportunity to work on a practical project of our interest.

We would also like to thank the ITLL staff for providing us with arduino kits which were essential for checking the various hardware modules in the project and served as a way to verify their integrity.

We would also like to thank the Course TA's Parth Kharade, Parth Thakkar and Akash Karoshi who were also of invaluable help during the entire duration of the course and provided general guidance regarding the project whenever needed.

Thank You.

8. REFERENCES

Figure No.	**Source link**
Figure 4	DigiKey
Figure 10	MAX30102-IR-Red-LED-photodetector.jpg (715×493)
Figure 12	GY-NEO6MV2 NEO-6M GPS Controller Module with Ceramic Guinea | Ubuy
Figure 13	Guide to NEO-6M GPS Module Arduino | Random Nerd Tutorials
Figure 15	SD Card Socket Module | Makerfabs Electronics
Figure 18	2inch LCD Display Module, IPS Screen, 240×320 Resolution, SPI Interface
Figure 19	2inch LCD Module - Waveshare Wiki

9. DIVISION OF LABOR

Nalin Saxena worked on the following components,

- Interfacing NRF Bi-directional Communication
- Interfacing MAX30102 Heart Rate and SpO2 sensor
- Overall Software Design

In the report, Nalin worked on the Introduction, NRF and Max Hardware element overview and the complete Software section.

Abhirath Koushik worked on the following components,

- Interfacing GPS Module
- Interfacing SD Card Module
- Interfacing Graphical LCD Module

In the report, Abhirath worked on the Board Design, GPS, SD Card and Graphical LCD Hardware element overview. Additionally, Abhirath worked on the Testing Process, Results and Error Analysis and Schematic sections.

10. APPENDICES

The appendix section has information about the following,

- Bill of materials
- Schematics
- Code
- Data sheets

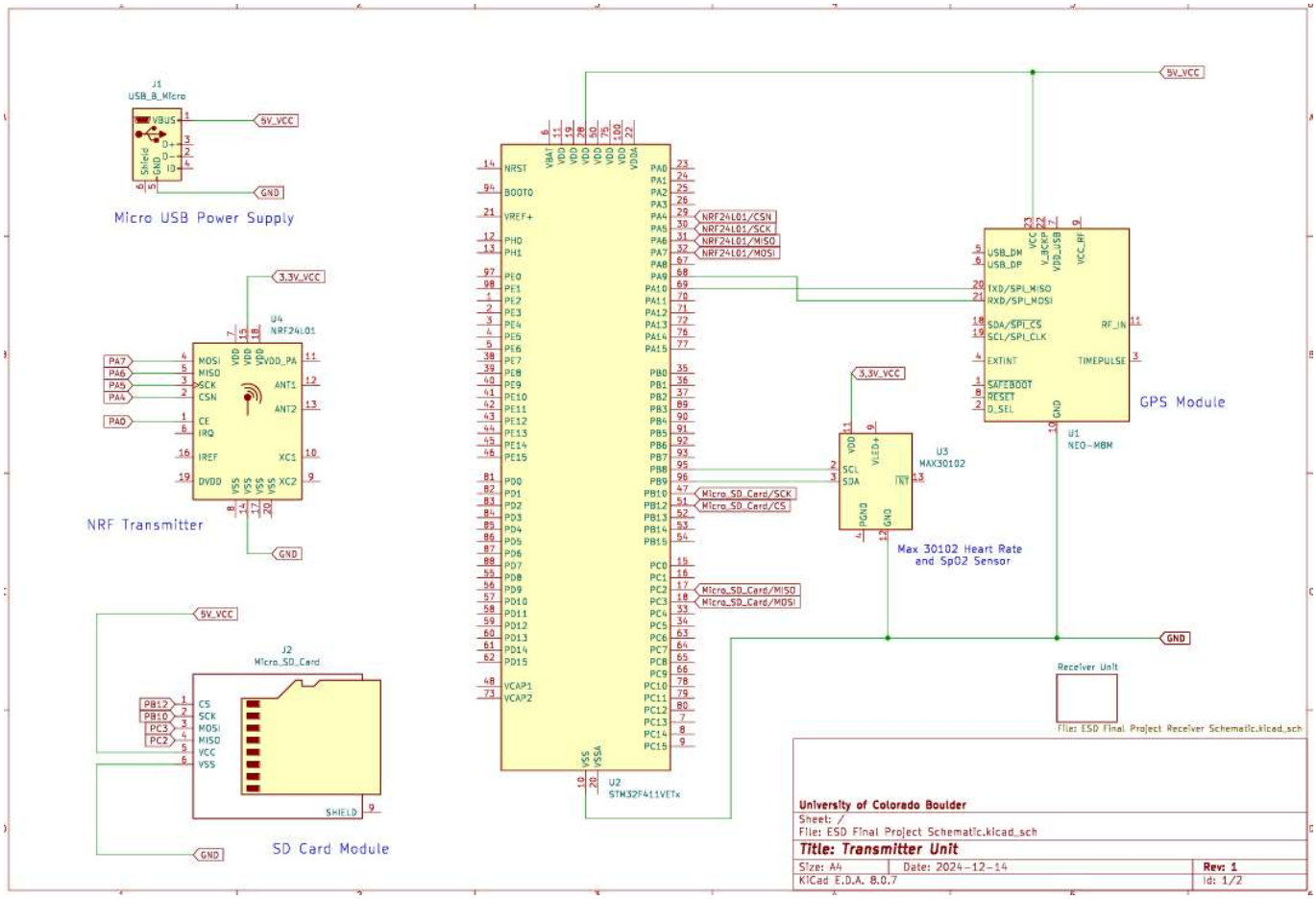
10.1. BILL OF MATERIALS

Module	Price	Purchase source
LCD Display Module	**14.99 $**	Purchase Link
NRF24L01 (4 counts)	**7.89 $**	Purchase Link
MAX30102	**6.99 $**	Purchase Link
SD Card Module (4 counts)	**5.99 $**	Purchase Link
Prototype board (2 counts)	**14 $**	**ECE store**

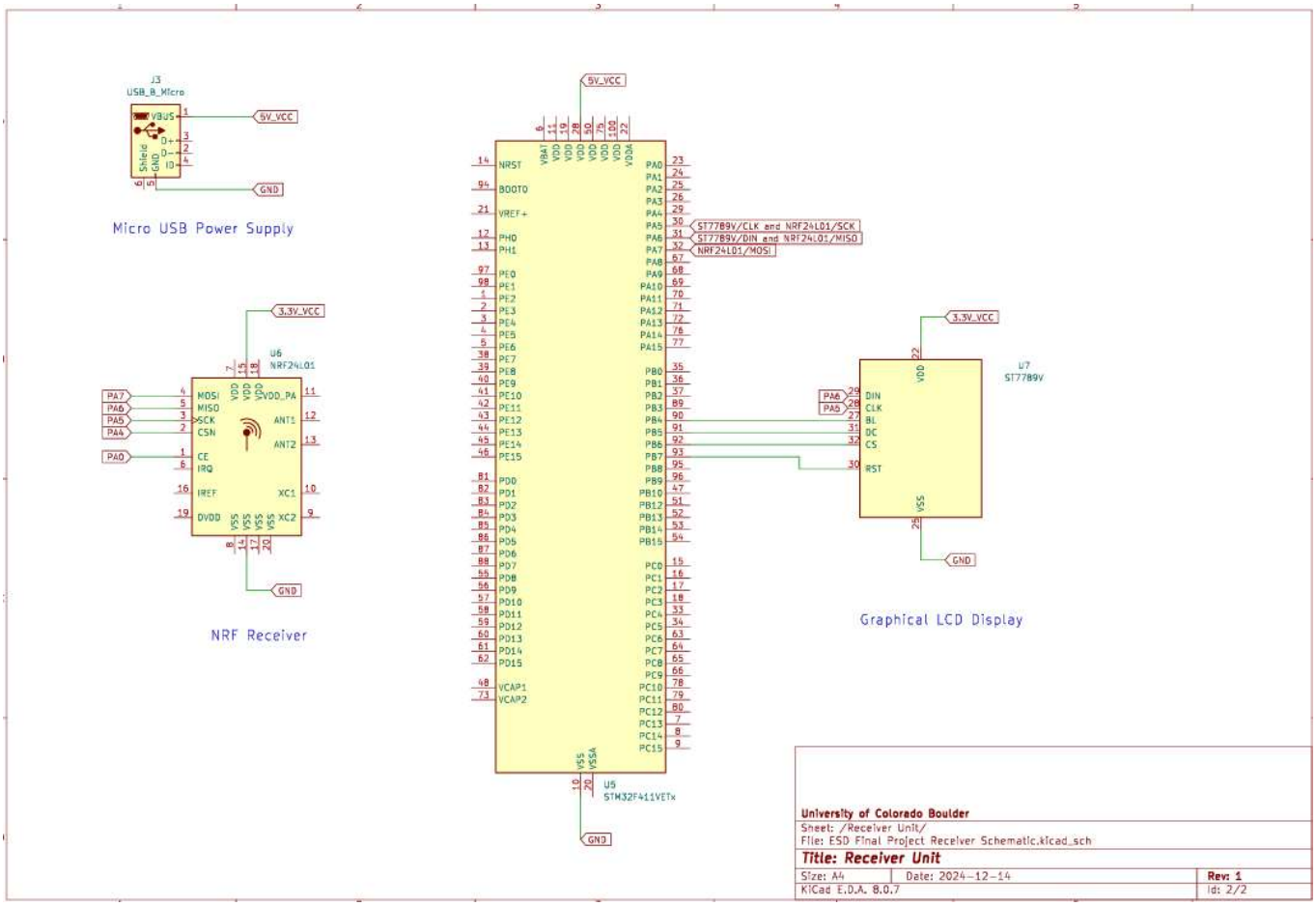
Total Bill of materials - **49.86$**

10.2. SCHEMATIC

Transmitter Unit



Receiver Unit



10.3. CODE

Please refer to the Appendix Code folder in the Report folder for the code files of this project. It is separated into the Transmitter and Receiver folders for easier understanding.

10.4. DATASHEETS AND APPLICATION NOTES

Please refer to Datasheets and Application Notes Folder for these documents.

10.5. BACKUP DOCUMENT

Original Google Docs Document: ESD Final Project Report