ELE 417 – Embedded System Design Project

Oğulcan Uğuroğlu 21828973
Barış Büyükyılmaz 21828241
Hasan Can Sert 21828839

Motion Detection and Visualization System

14.01.2023

Instructor : Dr. Ali Ziya Alkar

Contents

- What we aimed?……………………………3
- Project Details
  - Modules……………………………….4
  - Interfaces……………………………..5
  - System Design……………………..15
  - Sensor Fusion……………………...16
  - Cryptology Process………………20
  - Visualization…………………………29
- First Node…………………………………..30
- Second Node………………………………44
- Third Node…………………………………53
- Conclusion………………………………...63
- References…………………………………64
- Video Link………………………………….65

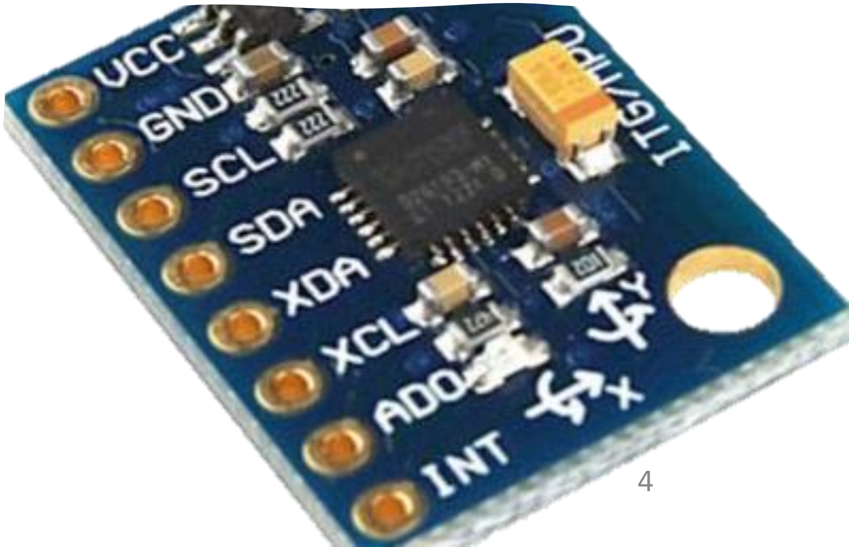
What we aimed?

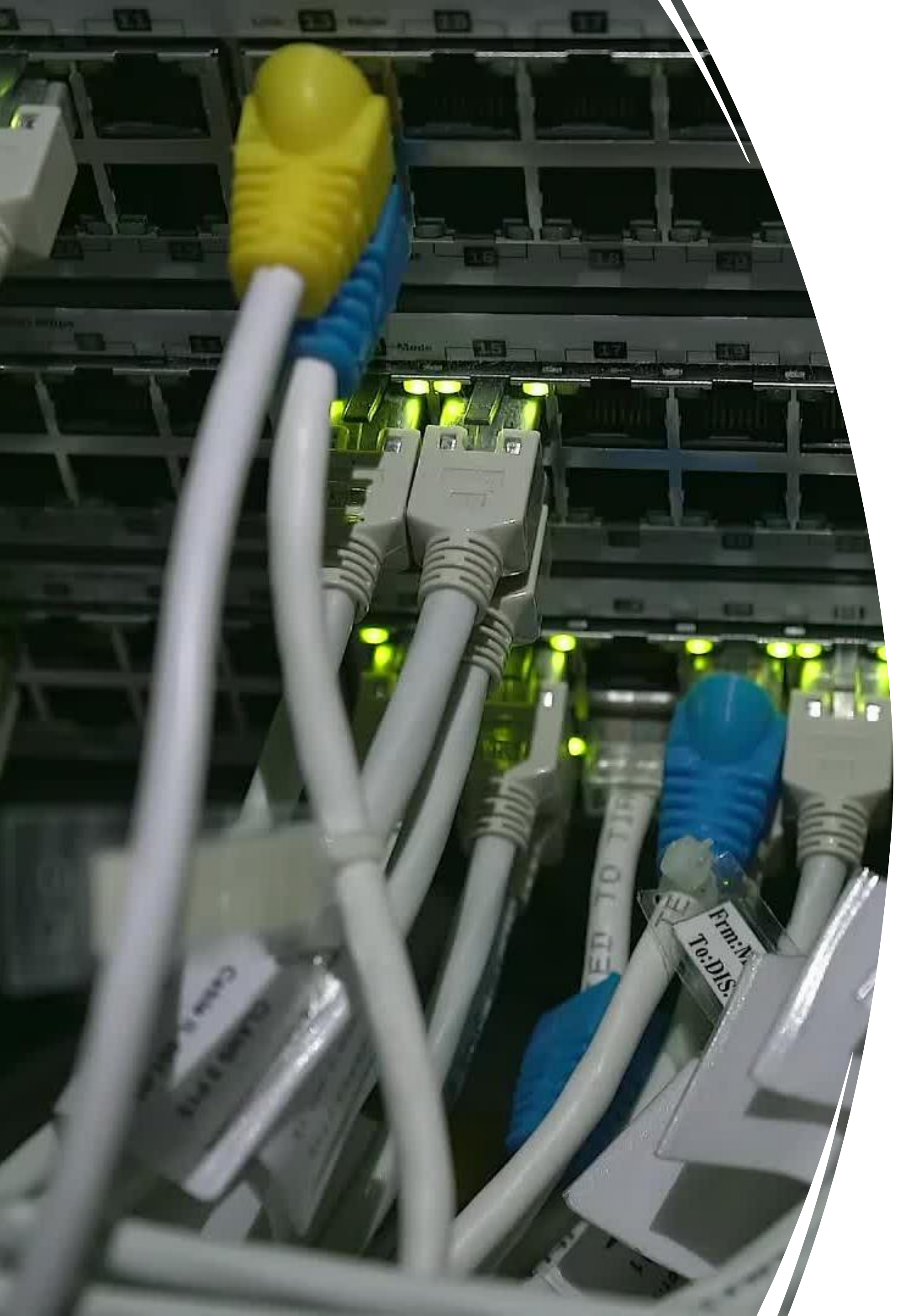
- The mission of the project is to use the MPU6050 accelerometer and gyroscope to visualize the motions of the modal plane. This will likely involve collecting data from the sensors and using it to create visual representations of the motion being measured.

Modules

- MPU6050 is basically a sensor for motion processing devices. It can measure 6 axis acceleration and angular velocity. This sensor module is used for estimate motion of the plane. It uses I2C interface to communicate with MCU.

- NRF24 is a wireless communication module. NRF24 module capable both transmission and recieving. Every node has NRF24 module to communicate each other. It can programmed by using SPI interface.



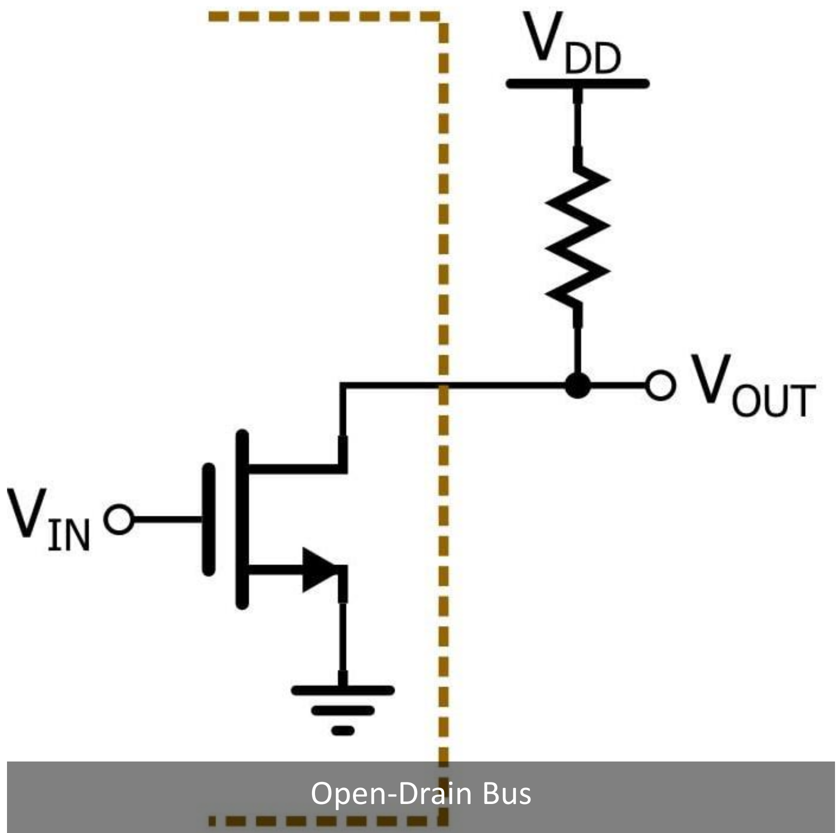


Interfaces

- I2C (Inter Integrated Communication)
- SPI (Serial Peripheral Interface)
- UART (Universal Asynchronous Receiver/Transmitter)

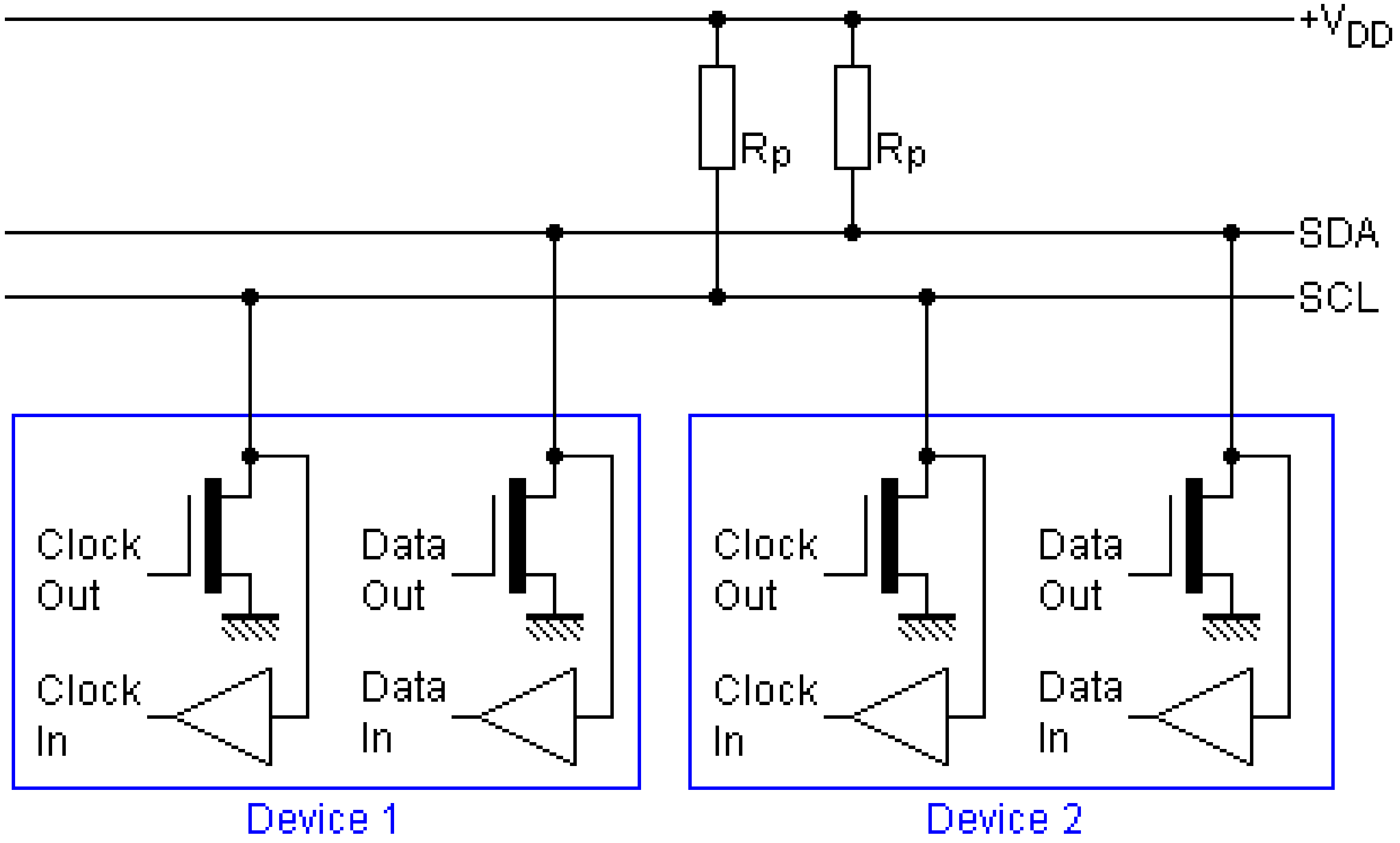
Inter-Integrated Circuit (I2C)

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire. It allows communication between multiple controllers (master) with multiple peripherals (slave). It has only 2 wires SDA, and SCL. SDA is a data bus, all data are transferred by this bus. SCL is a serial clock bus, it provides clock cycles for synchronization. These two bus need to pull up because they are open-drain buses.



Open-Drain Bus

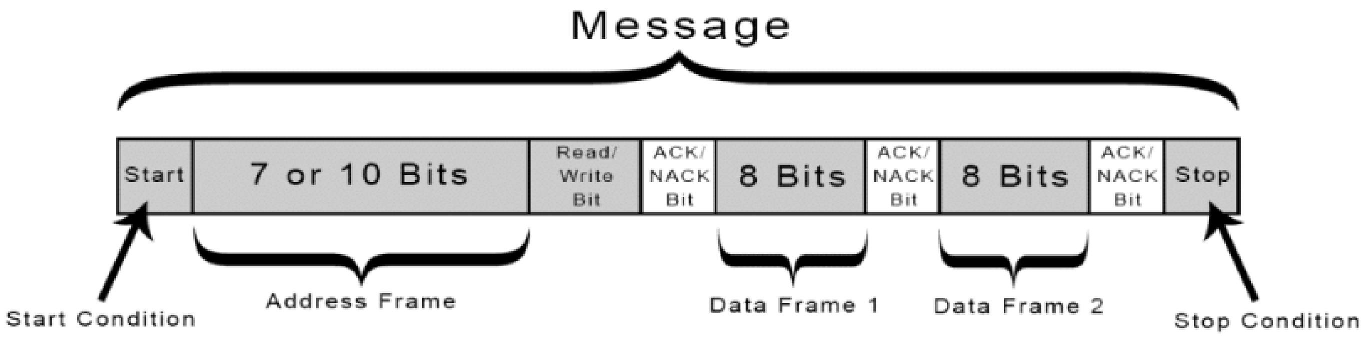
I2C Overall System



I2C Message Frame

- I2C messages are broken up into frames of data. Each message has an address of the slave, and one ore more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits.
- **Start Condition:** The SDA line switches from a high voltage level to a low voltage level before the SCL line switches from high to low.
- **Stop Condition:** The SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high.
- **Address Frame:** Unique slave address.
- **Read/Write Bit:** A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).
- **ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

I2C Message Frame

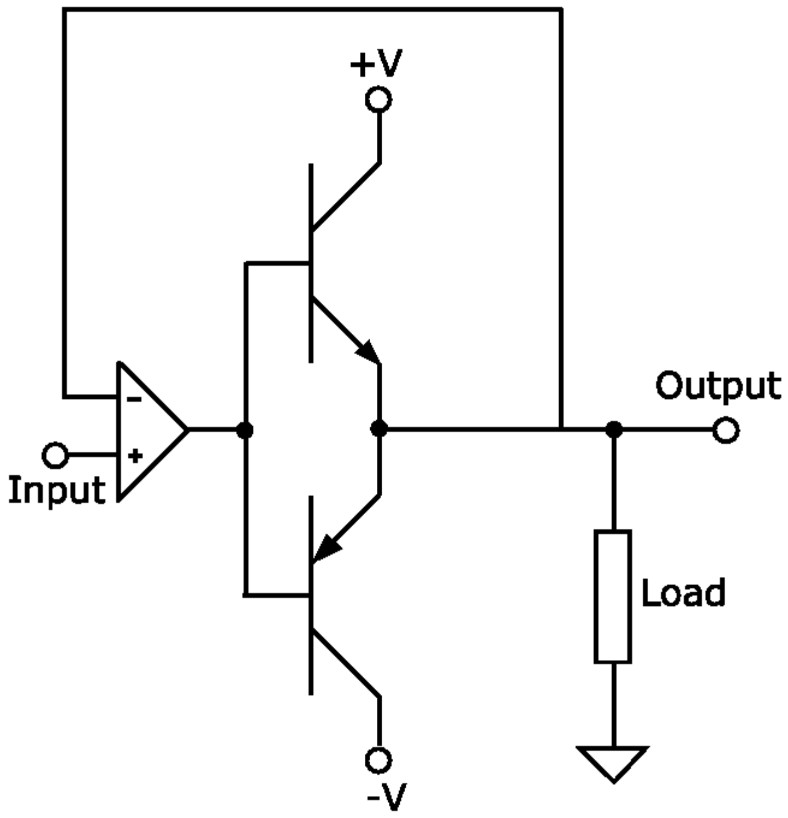


I2C C Implementation

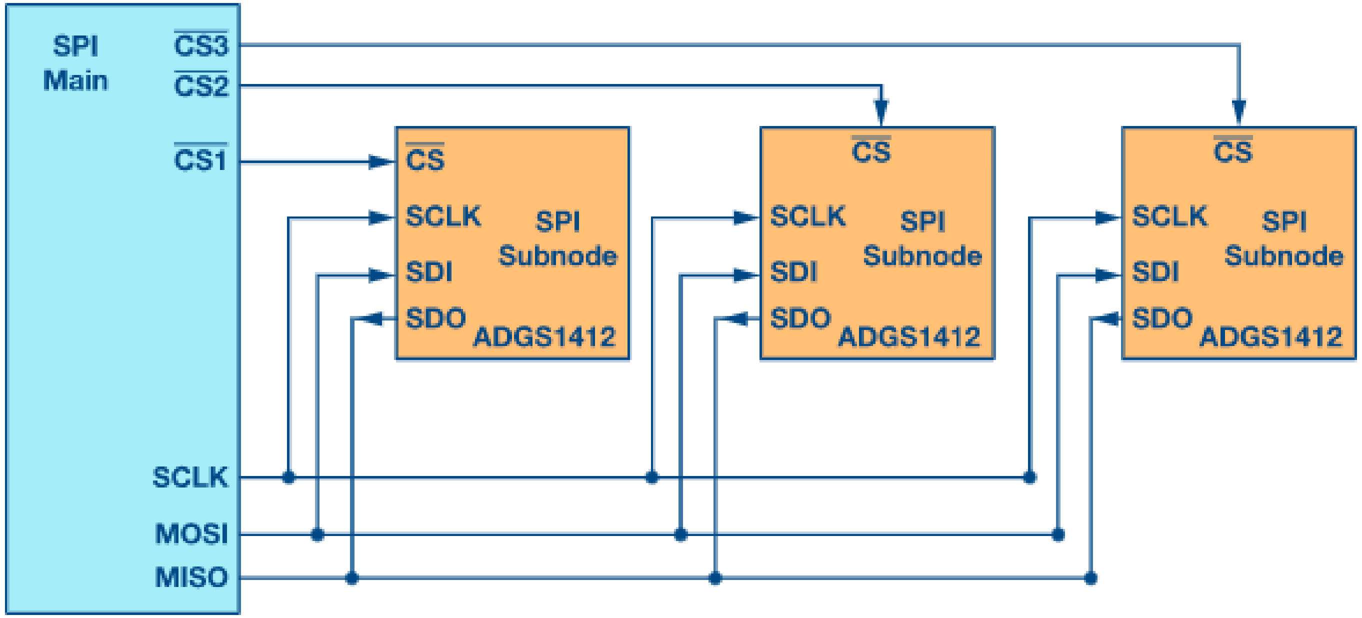
```
• bool i2c_write_byte(uint8_t byte)
• {
•     bool success = false;
•     UCB0CTL1 |= UCTXSTT + UCTR; /* Set up master as TX and send start condition */
•     /* Note, when master is TX, we must write to TXBUF before waiting for UCTXSTT */
•     UCB0TXBUF = byte; /* Fill the transmit buffer with the byte we want to send */
•     while (UCB0CTL1 & UCTXSTT); /* Wait for start condition to be sent */
•     success = !(UCB0STAT & UCNACKIFG);
•     if (success) {
•         while (!(IFG2 & UCB0TXIFG)); /* Wait for byte to be sent */
•         success = !(UCB0STAT & UCNACKIFG);
•     }
•     UCB0CTL1 |= UCTXSTP; /* Send the stop condition */
•     while (UCB0CTL1 & UCTXSTP); /* Wait for stop condition to be sent */
•     success = !(UCB0STAT & UCNACKIFG);
•     return success;
• }
```

Serial Peripheral Interface (SPI)

- SPI is a synchronous, full duplex interface. The SPI interface can be either 3-wire or 4-wire. These are MOSI, MISO, SCLK, and CS. MOSI and MISO are the data lines, CS is the chip select indicates the slave that communicate, and SCLK is clock line for synchronization. Lastly it has push/pull configuration hence it does not need any pull-up resistors.



SPI Overall System

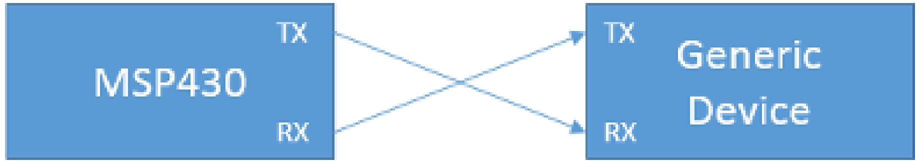


SPI C Implementation

- `uint8_t spi_transfer(uint8_t inb)`
- `{`
- `UCA0TXBUF = inb;`
- `while ( !(IFG2 & UCA0RXIFG) )  // Wait for RXIFG indicating remote byte received via SOMI`
- `;`
- `return UCA0RXBUF;`
- `}`

Physical Interface of UART Communication

From a pinout perspective, UART signals only require one line for unidirectional communications, although two are typically used for bi-directional transmit and receive. Being asynchronous, UART signals don't require any other clock line because the two UART devices agree on a common baud rate, stop, start and data bits. This makes the receiver capable of decoding the data. UART is connected by crossing the TX and RX lines, as shown below:

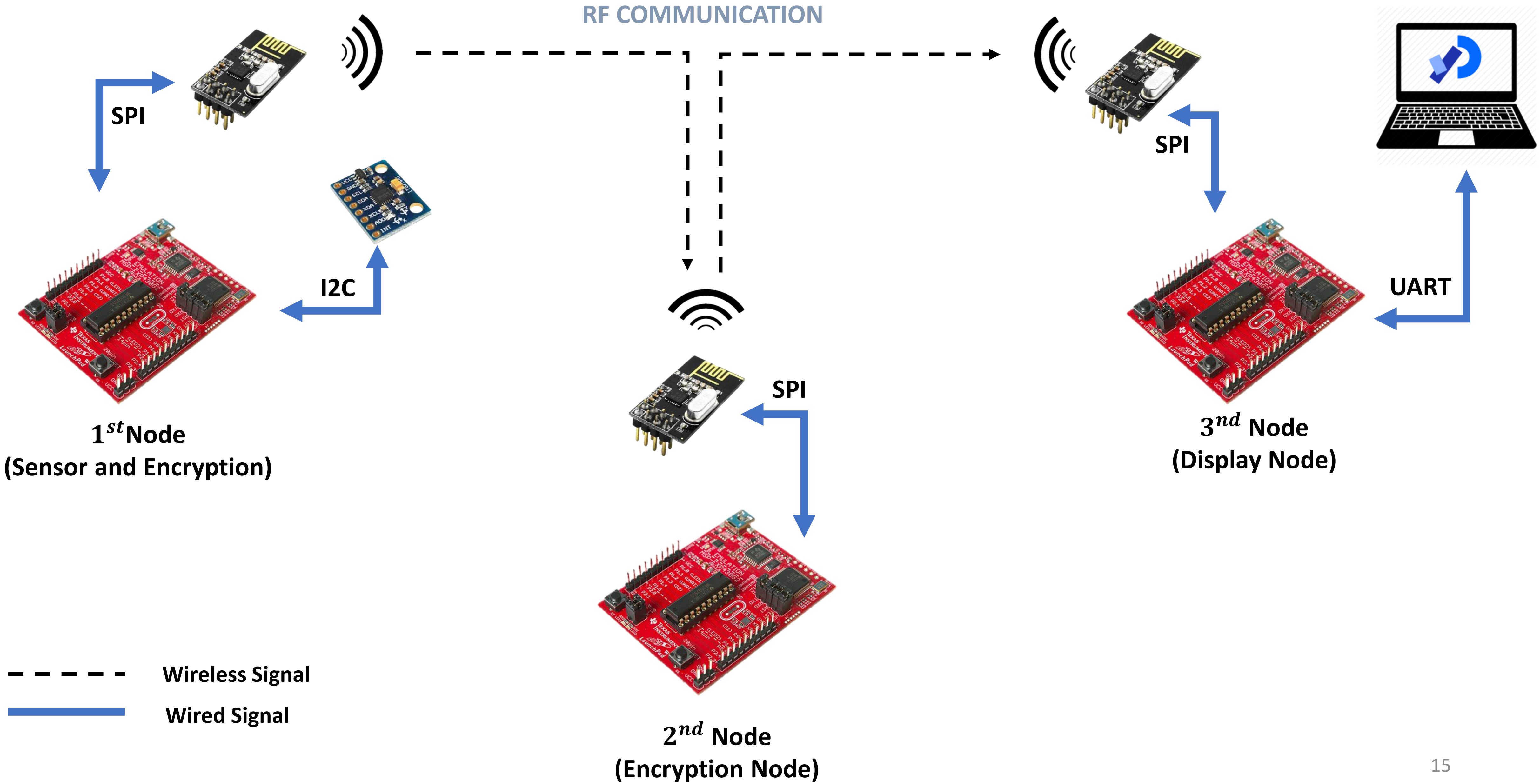


The smallest element of transmission is the UART frame or character, which consists of the Start bit/s, data bits, stop bits and optional parity bits, as shown below:

Bit number	1	2	3	4	5	6	7	8	9	10	11
Start bit	Start bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Data Bit	Stop Bit	Stop Bit

The start bits alerts the receiver that data is coming. Without it, if the first bit was a '1', it would be seen as an idle line since an idle UART line is also high. The number of data bits is typically 8, but it can be configured for 7 bits as well. Although some UART receivers can use a different number of bits, only 8 or 8 bits are supported by the MSP430. After the data bits stop bits are sent along with an optional parity bit.

System Design

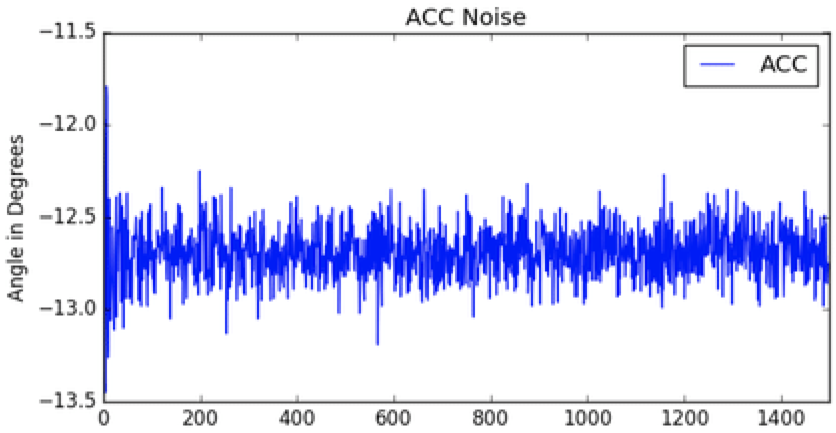


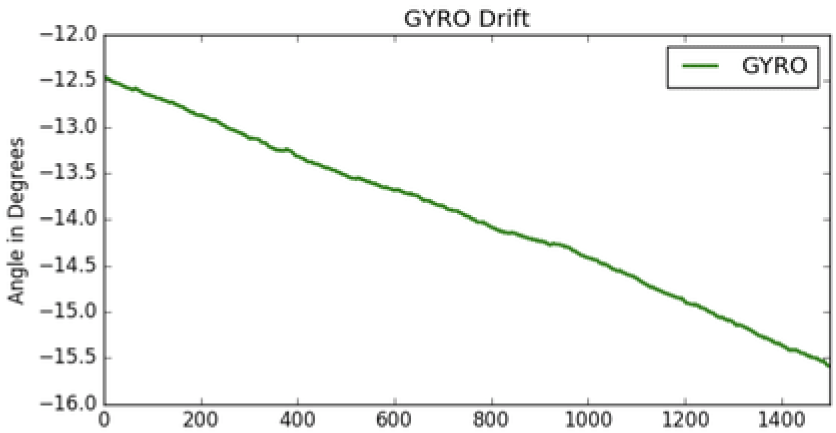
Sensor Fusion

Why we need sensor fusion?

-Accelerometer data are too noisy to estimate short-term changing in acceleration. But it is very stable long term because gravity does not change over time.

-On the Gyroscope side, it is highly accurate for short-term measuring. But we need to integrate data to calculate angles. So it means that noise also sums up actual data. Hence the data shifts over time.





Sensor Fusion

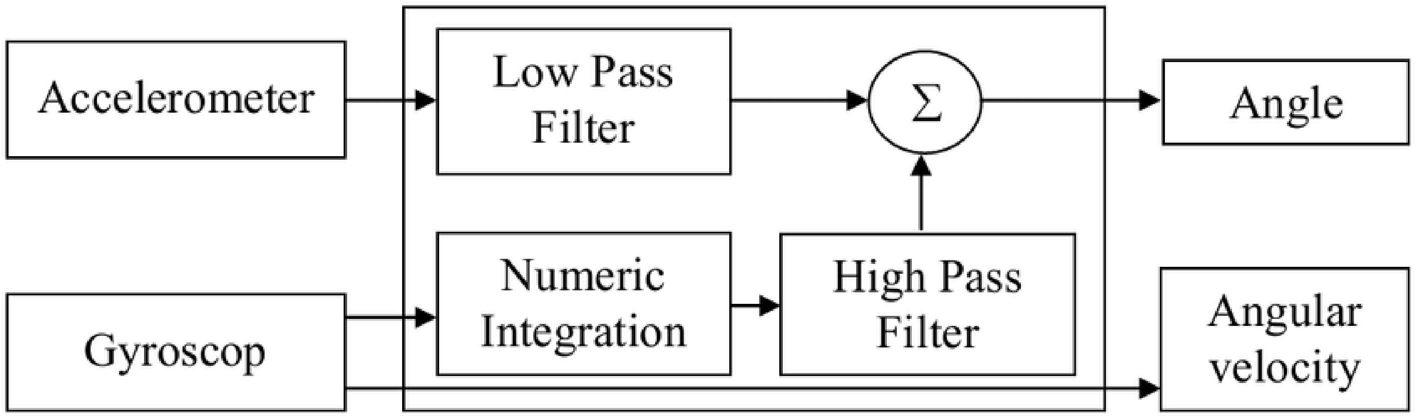
How we prevent this shifting and noise on our system?

- If we use sensor fusion than we can handle this situation.

How we do that?

- Accelerometer is accurate on long term and gyroscope is accurate on short term if we combine long term accelerometer data, and short term gyroscope data we can get more realistic, accurate data. To do that only we need to do pass the accelerometer data through low pass filter, and pass the gyroscope data through high pass filter than add them up. Finally done the fusion.

Sensor Fusion Block Diagram



Sensor Fusion

First, write the complementary sensor fusion equation.

$$\theta_b = \left(\frac{1}{1+as}\right)\theta_A + \frac{1}{s}\left(\frac{as}{1+as}\right)\omega_G$$

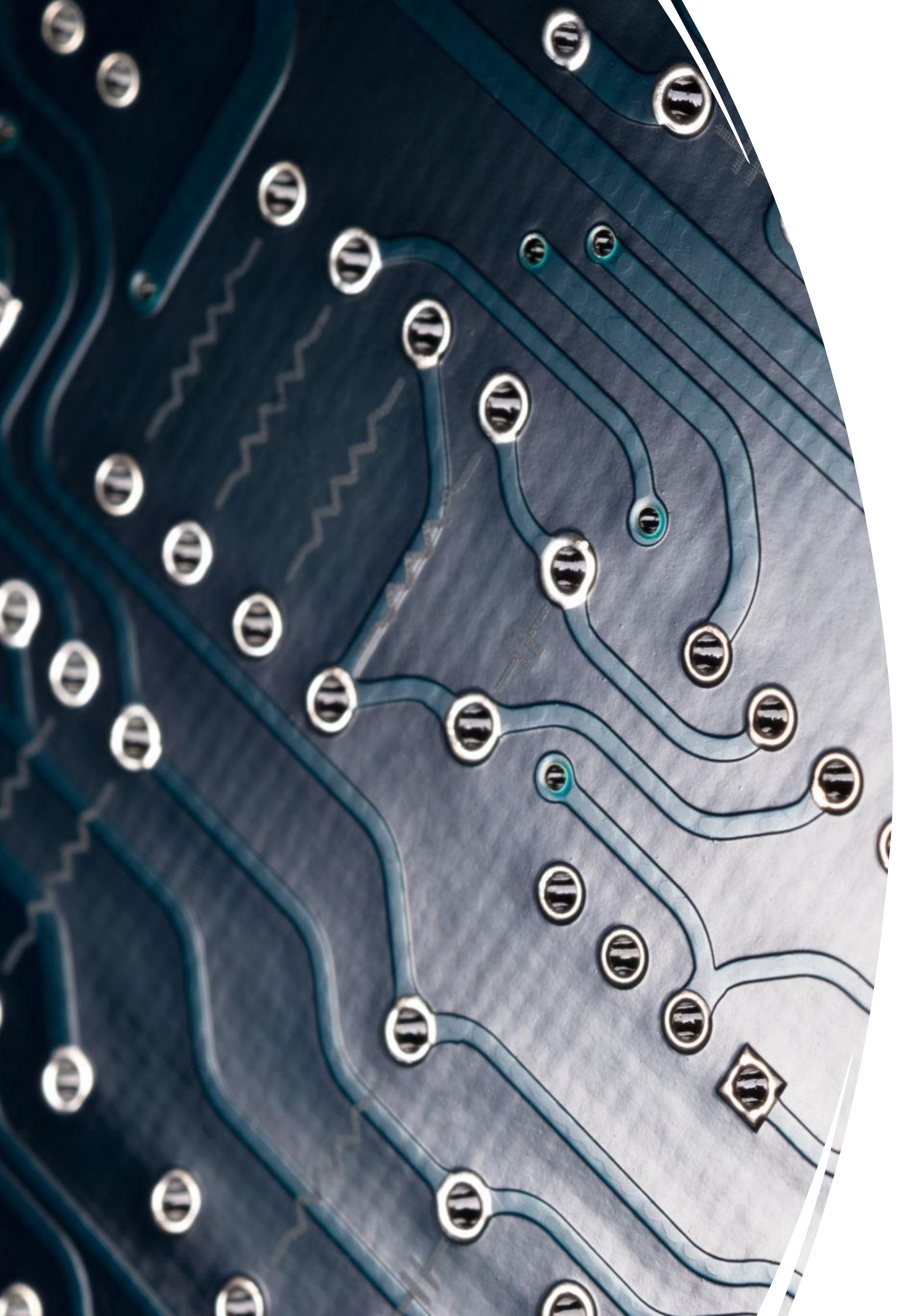
Since it is in the frequency domain then take the inverse Laplace of it and change the differentiations by different equations. Finally, make some algebration then the equation becomes as below.

$$\theta_b(n) = b\left[\theta_b(n-1) + T\omega_G(n)\right] + (1-b)\theta_A(n)$$

After implement this discrete time function into C program.

/* implement complementary filter for sensor fusion */

```
angle_x = (0.98)*(angle_x + (gyro_data[0] * 0.02)) + (0.02 * x_rot);
angle_y = (0.98)*(angle_y + (gyro_data[1] * 0.02)) + (0.02 * y_rot);
```



Cryptology Process

In cryptology processes, we follow the AES (Advenced Encryption Standard). AES is one of the most used algorithms for block encryption and decryption.

We choose AES128 at ECB (Electronic Code Book) Mode to implement to the MSP430. AES128 can handle 128 bits of data and this size is enough for our project ECB mode is very efficient for time and memory consumption than other modes of operations.

AES-128

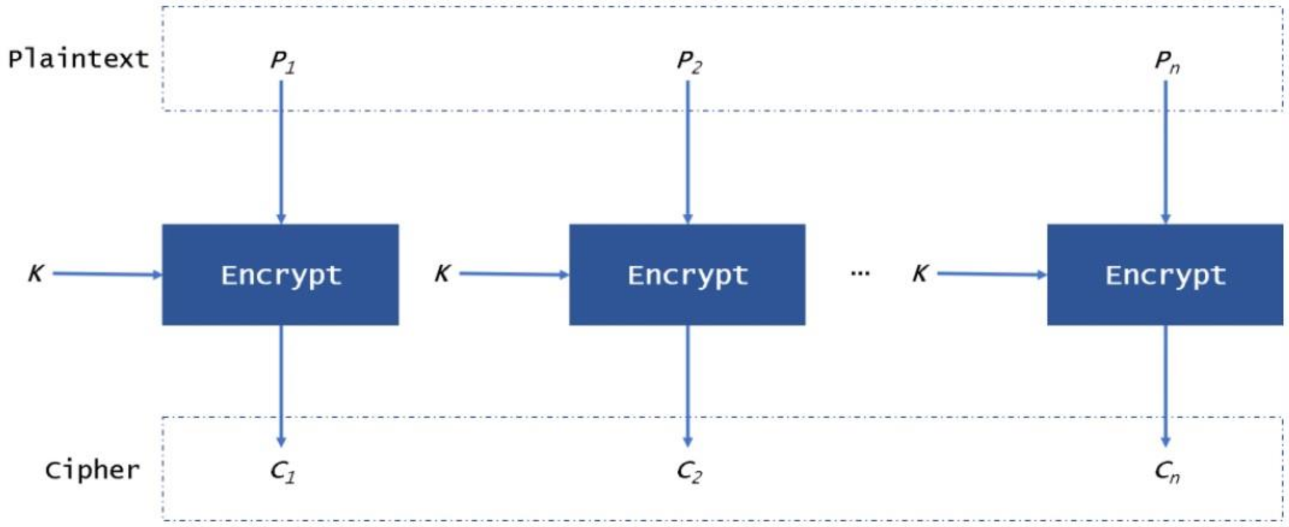
Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES-128 is a symmetric key encryption algorithm that uses a 128 bit key to encrypt and decrypt data.

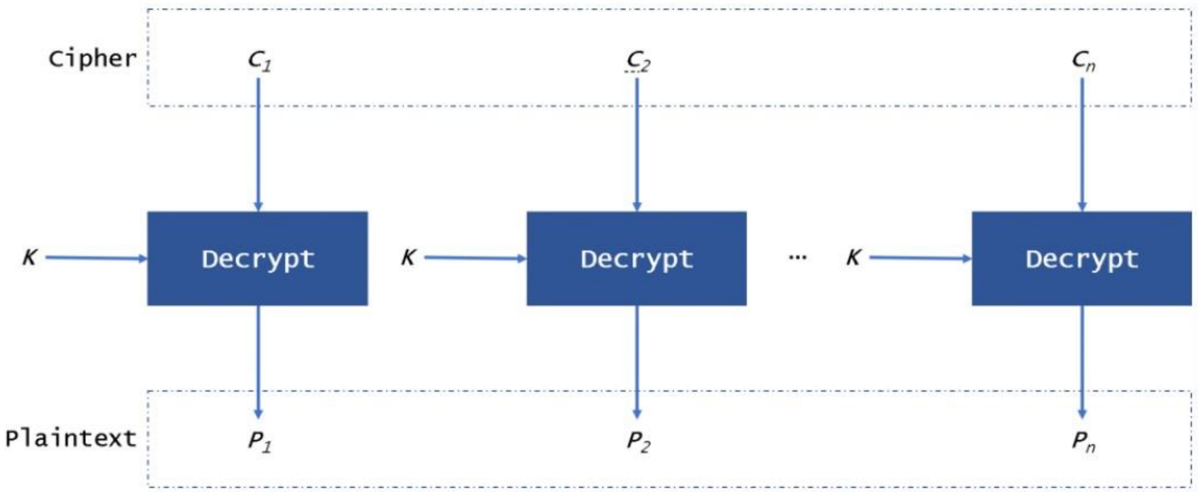
In AES128 algorithm, there is a 16-byte PlainText array, in our code the PlainText array is 16-byte buf[] array that includes the data from MPU6050.

In the AES128, there are some special matrixes called S-box, InvS-box and RoundKey. We composed a 16-byte key[] array with determined array elements. S-box and InvS-box is already defined matrices in ‘aes.h’ header that used for substitution of the elemants in encryption and decryption.

ECB Mode

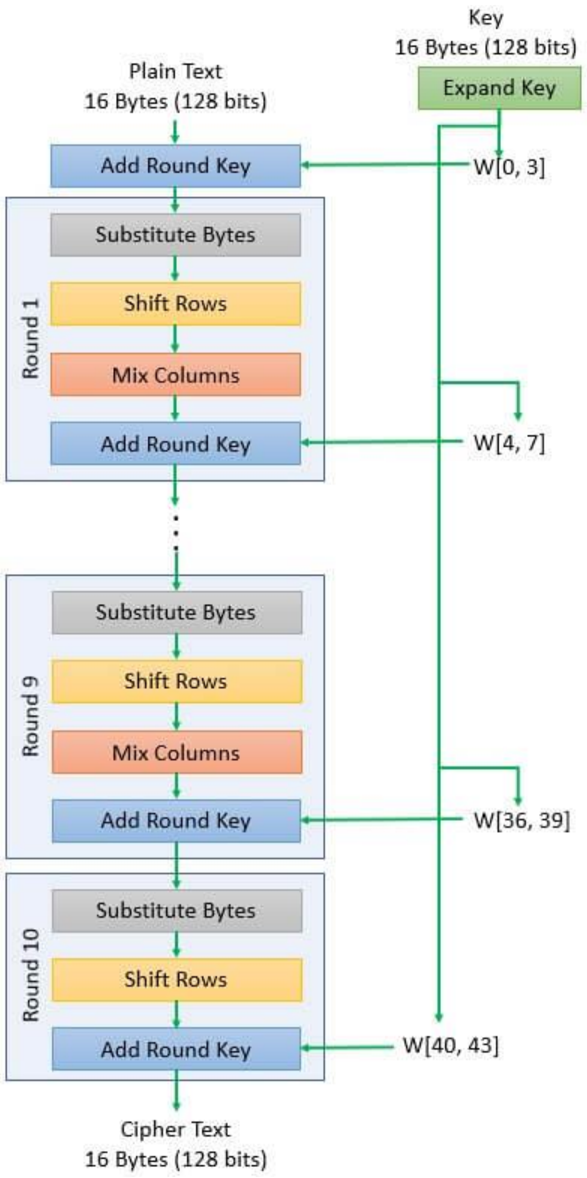
In ECB mode, the plaintext is divided into fixed-size blocks (128 bits in the case of AES-128), and each block is independently encrypted using the same key.





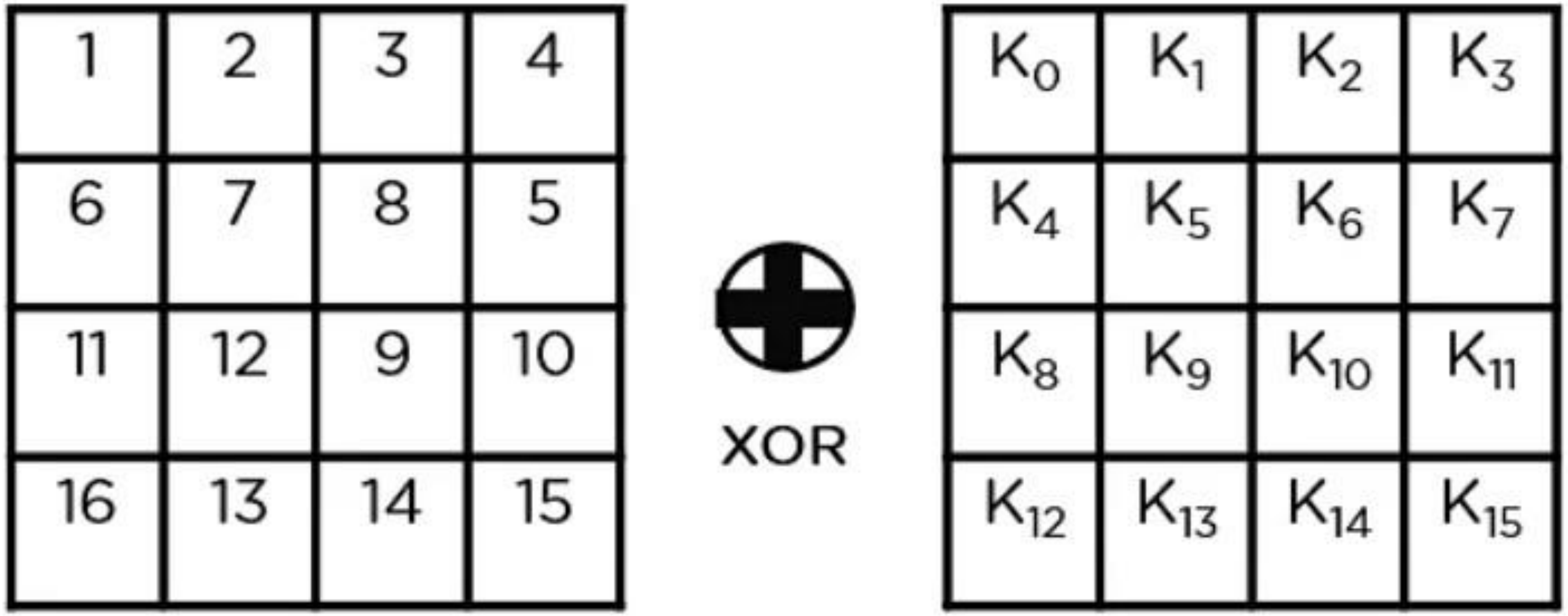
Encryption Process

The AES-128 encryption algorithm consist of ten rounds. After the ten rounds, we obtain encrypted data which is in the 128 bits Cipher Text array.



1) Add Round Key

Every data at the Plaintext is XORed with correspounding element in Key array.



2) Sub-Bytes

It converts each byte of the state array into hexadecimal, divided into two equal parts. These parts are the rows and columns, mapped with a substitution box (S-Box) to generate new value for the final state array.

3) Shift Rows

It swaps the row elements among each other. It skips the first row. It shifts the elements in the second row, one position to the left. It also shifts the elements from the third row two consecutive positions to the left, and it shifts the last row three positions to the left.

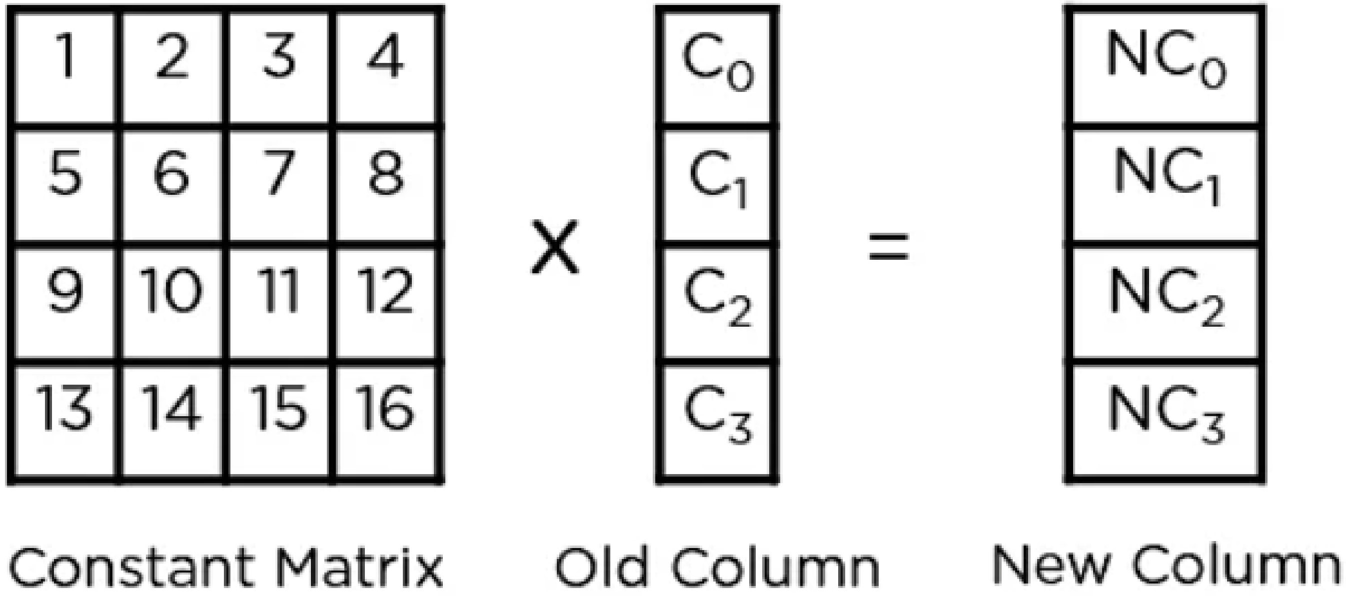
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

→

1	2	3	4
6	7	8	5
11	12	9	10
16	13	14	15

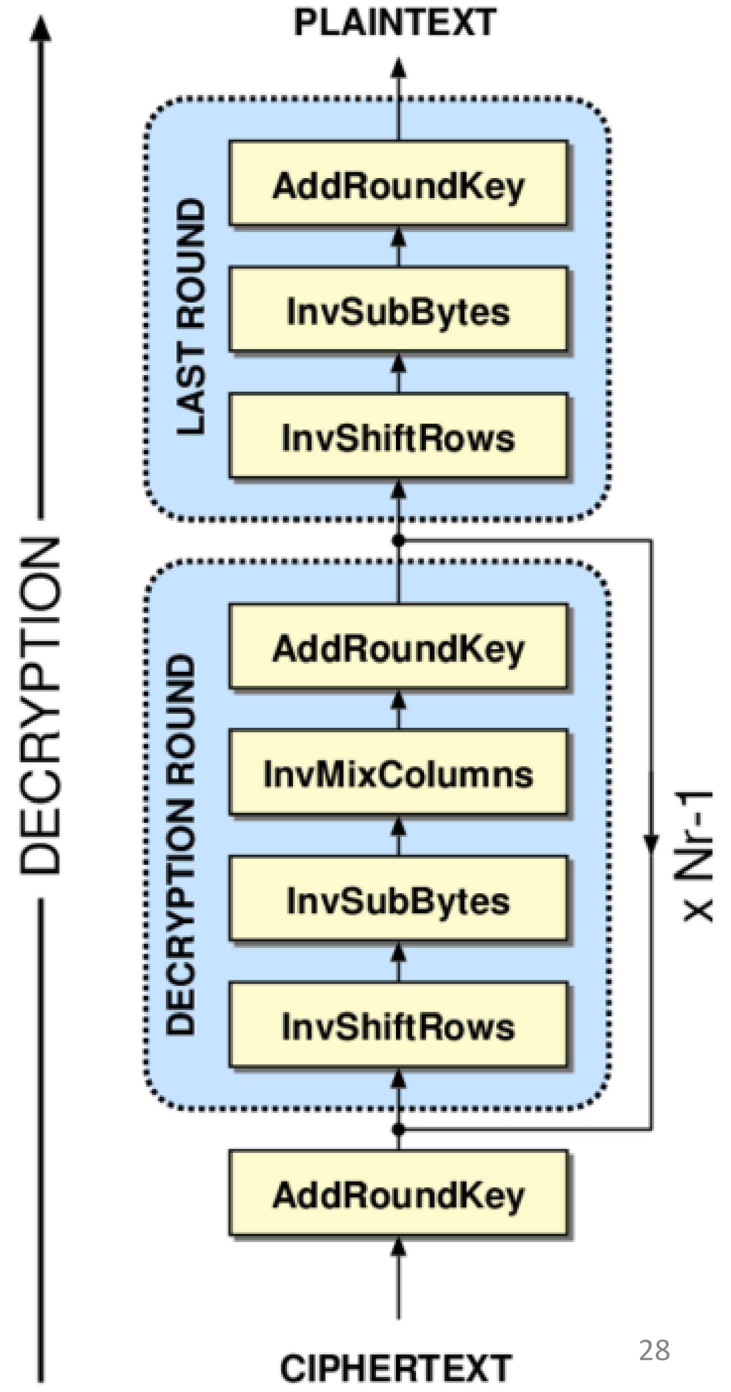
4) Mix Columns

It multiplies a constant matrix with each column in the state array to get a new column for the subsequent state array. Once all the columns are multiplied with the same constant matrix, you get your state array for the next step. This particular step is not to be done in the last round.



Decryption Process

The decryption process is the reverse of the encryption process. It involves the use of the same key that was used for encryption and involves several rounds of transformation to transform the ciphertext back into the original plaintext.



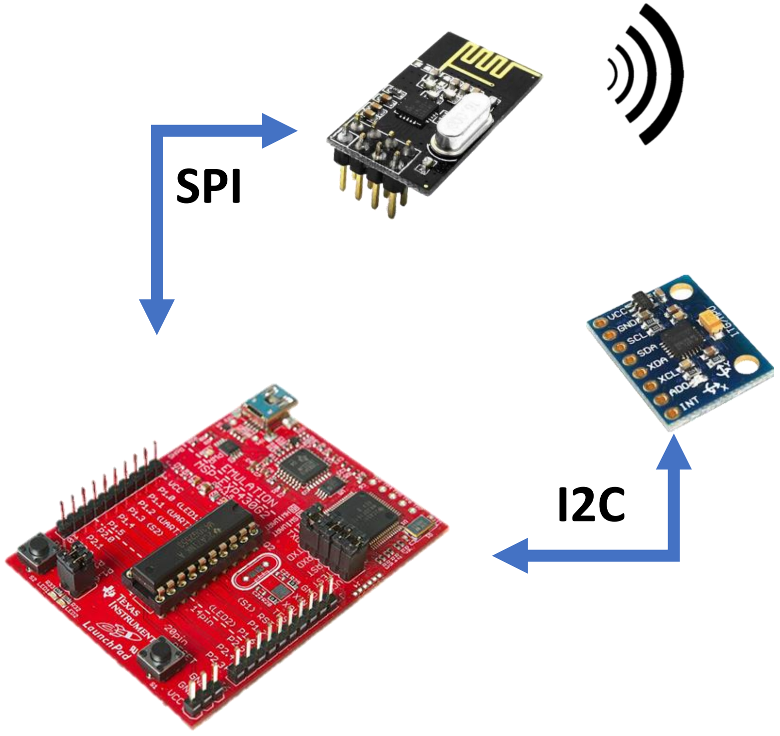
Visualization : Processing IDE

Processing is a free graphical library and integrated development environment (IDE) built for the electronic arts, new media art, and visual design communities with the purpose of teaching non-programmers the fundamentals of computer programming in a visual context.

Processing IDE reaches the data from MSP through the COM port of the computer. This data changes the box position.

First Node (Sensor and Encryption)
Barış BÜYÜKYILMAZ

- Main Functions:
  - Getting Sensor Data
  - Signal Processing
  - Encrypting
  - Transmitting

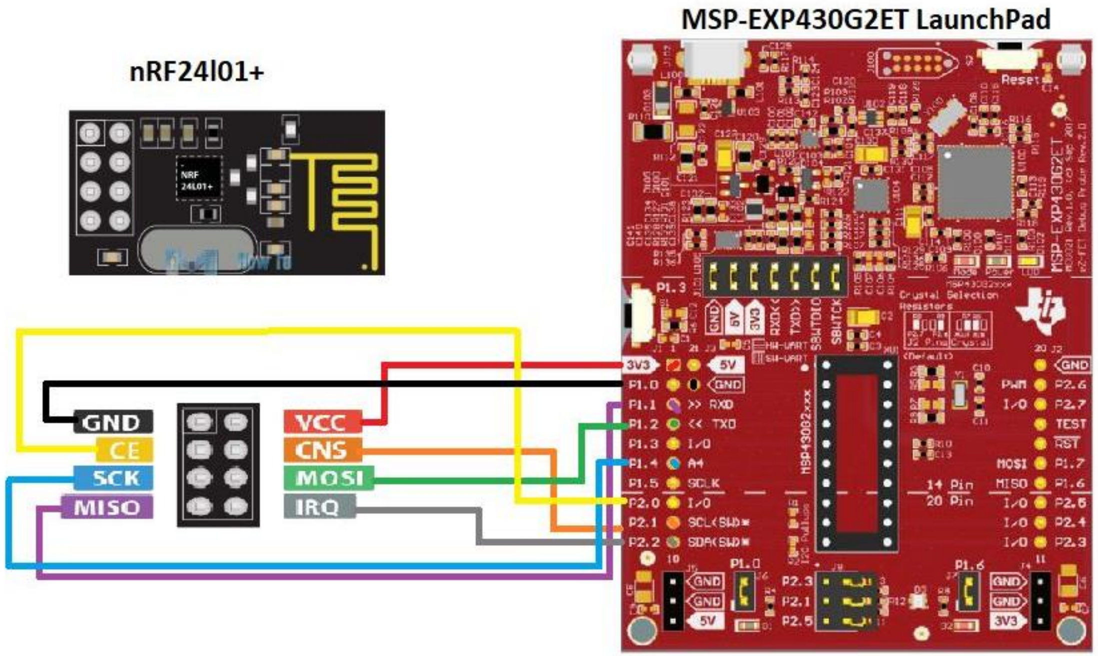


1^{st}Node

(Sensor and Encryption)

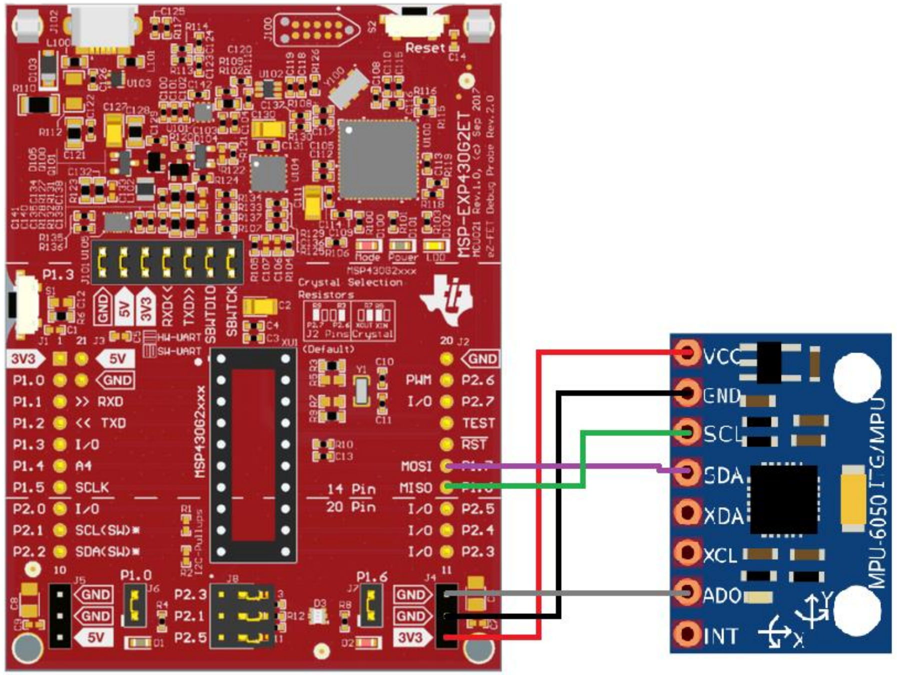
First Node NRF24 Connections

- In MSP430g2553 P1.1,P1.2 and P1.4 are configured for the SPI interface. P1.1 is selected as MISO, P1.2 is selected as MOSI, P1.4 is selected as SCLK. P2.0, P2.1, and P2.2 are connected the NRF24 module. P2.2 is interrupt of NRF24 module it used for the wake up CPU when data is transmitted.

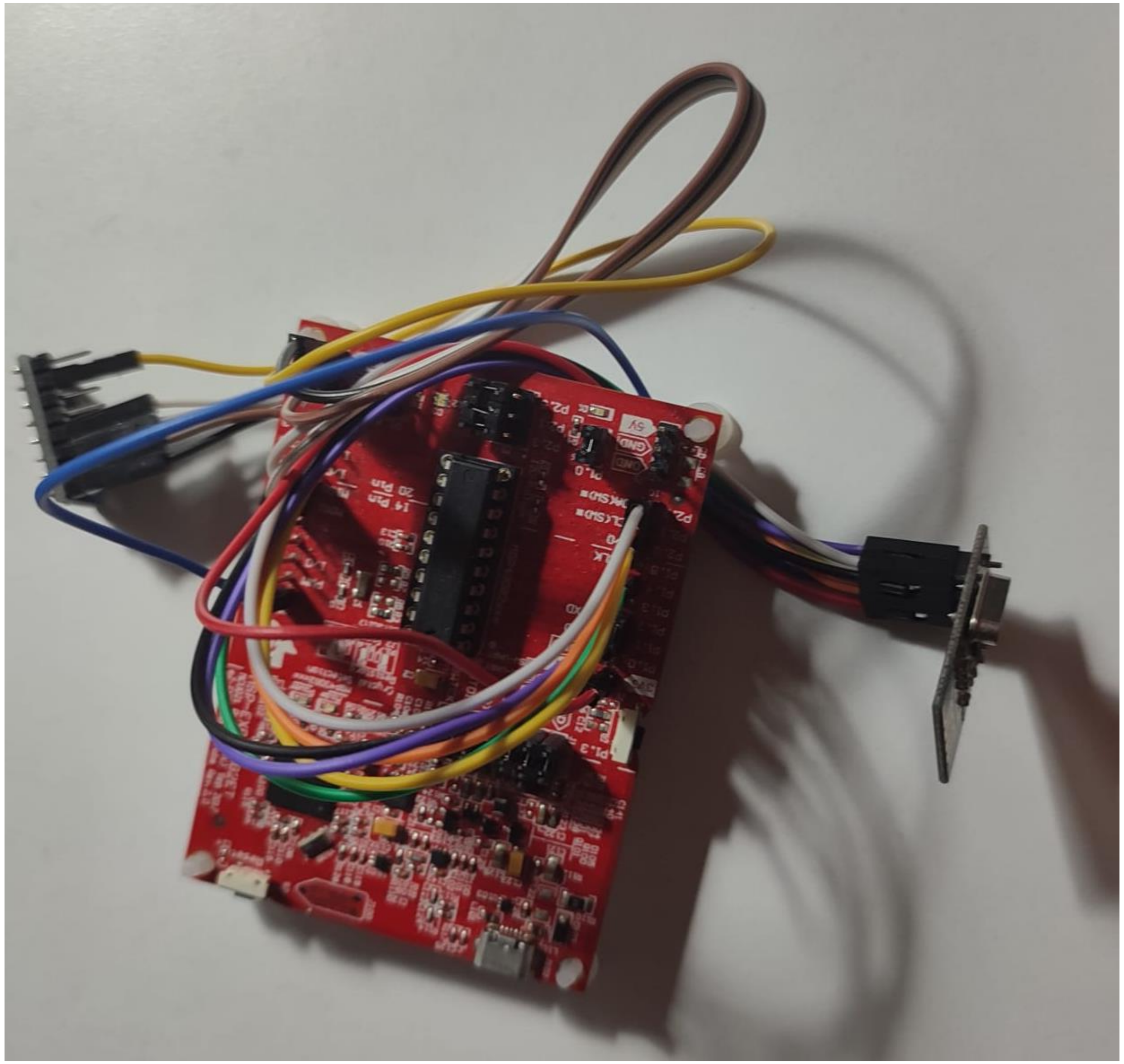


First Node MPU6050 Connections

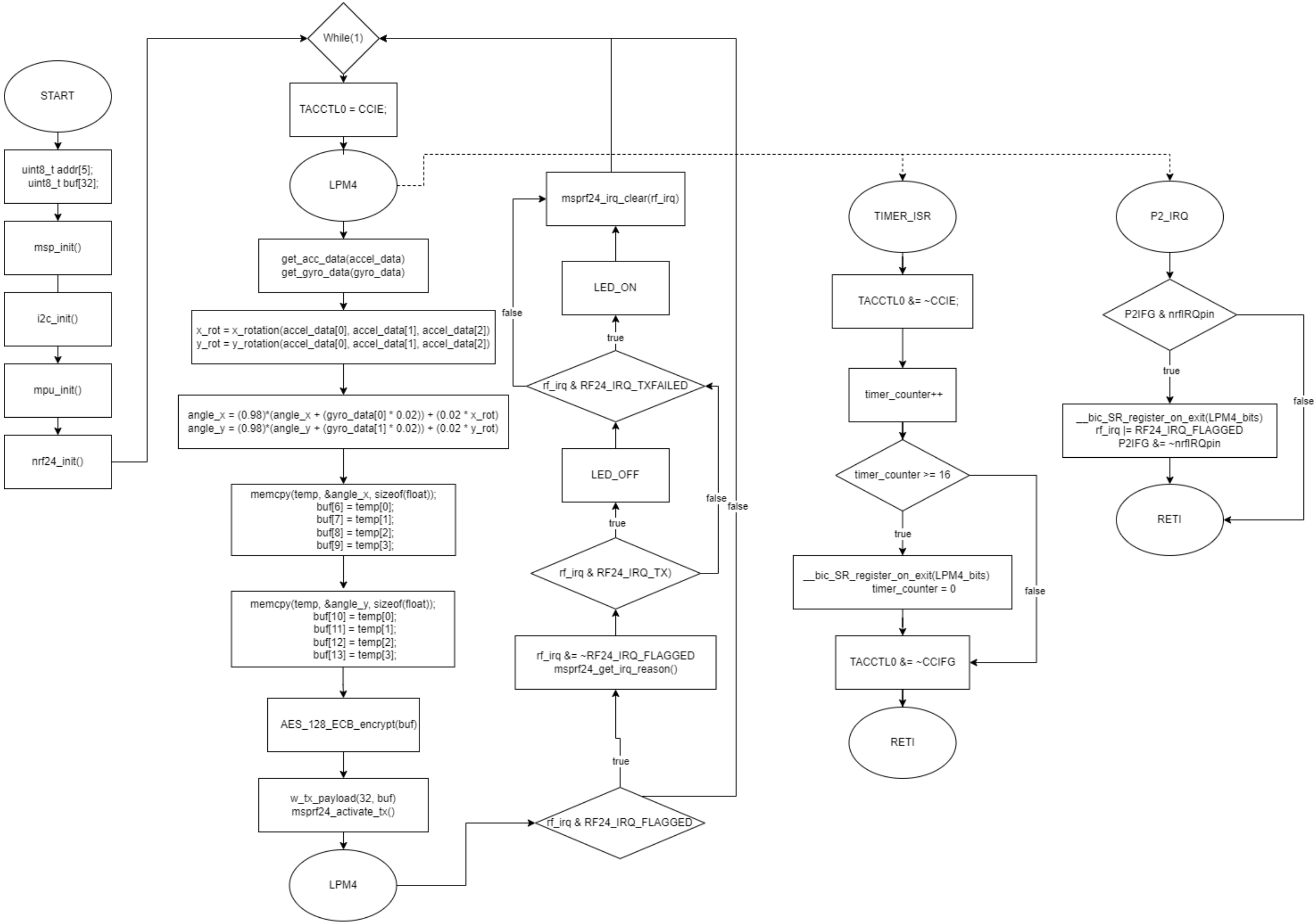
- USCI_B used for I2C interface. P1.7 is configured as SDA and P1.6 is configured as SCL. These ports also connected to MPU6050 module as in the figure.



Node 1 Connections



First Node Flowchart



First Node Implementation

```
• #include <msp430.h>
• #include "msprf24.h"
• #include "nrf_userconfig.h"
• #include "stdint.h"
• #include "I2C.h"
• #include "MPU6050.h"
• #include <stdlib.h>
• #include <stdio.h>
• #include "aes.h"

• volatile unsigned int user;
• char temp[sizeof(float)];

• int16_t accel_data[3];
• int16_t gyro_data[3];
• int16_t accel_offset[3];
• int16_t gyro_offset[3];
• char buffer_x[20], buffer_y[20];
• float x_rot, y_rot;
• float angle_x = 0;
• float angle_y = 0;
• uint16_t timer_counter = 0;

• void AES_128_ECB_encrypt(uint8_t *in);
• void msp_init(void);
• nrf24_init(void);
```

```
int main()
{
    uint8_t addr[5];
    uint8_t buf[32];

    msp_init();

    // Red, Green LED used for status
    P1DIR |= 0x01;              // P1.6 changed used for I2C SCK
    P1OUT &= ~0x01;             // P1.6 changed used for I2C SCK


    i2c_init();
    mpu_init();
    nrf24_init();
```

First Node Implementation

```
•   while(1){
•           //__delay_cycles(800000);
•           LPM4;
•           get_acc_data(accel_data);
•           get_gyro_data(gyro_data);

•           /* calculate x and y axes rotation */
•           x_rot = x_rotation(accel_data[0], accel_data[1], accel_data[2]);
•           y_rot = y_rotation(accel_data[0], accel_data[1], accel_data[2]);

•           /* implement complementary filter for sensor fusion */
•           angle_x = (0.98)*(angle_x + (gyro_data[0] * 0.02)) + (0.02 * x_rot);
•           angle_y = (0.98)*(angle_y + (gyro_data[1] * 0.02)) + (0.02 * y_rot);

•           memcpy(temp, &angle_x, sizeof(float));
•           buf[6] = temp[0];
•           buf[7] = temp[1];
•           buf[8] = temp[2];
•           buf[9] = temp[3];

•           memcpy(temp, &angle_y, sizeof(float));
•           buf[10] = temp[0];
•           buf[11] = temp[1];
•           buf[12] = temp[2];
•           buf[13] = temp[3];

•           AES_128_ECB_encrypt(buf);

•           w_tx_payload(32, buf);
•           msprf24_activate_tx();
•           LPM4;
```

```
•           if (rf_irq & RF24_IRQ_FLAGGED) {
•               rf_irq &= ~RF24_IRQ_FLAGGED;

•               msprf24_get_irq_reason();
•               if (rf_irq & RF24_IRQ_TX){
•                   P1OUT &= ~BIT0; // Red LED off
•                   P1OUT |= 0x40;  // Green LED on
•               }
•               if (rf_irq & RF24_IRQ_TXFAILED){
•                   P1OUT &= ~BIT6; // Green LED off
•                   P1OUT |= BIT0;  // Red LED on
•               }

•               msprf24_irq_clear(rf_irq);
•               user = msprf24_get_last_retransmits();
•           }
•       }
•   }
```

First Node Implementation

```
• void AES_128_ECB_encrypt(uint8_t *in)        // Encrypt datas at in[] array and store at in[]
• {
•     struct AES_ctx ctx;

•     uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };

•     AES_init_ctx(&ctx, key);
•     AES_ECB_encrypt(&ctx, in);
• }
• void msp_init(void)
• {
•     WDTCTL = WDTHOLD | WDTPW;
•     DCOCTL = CALDCO_16MHZ;
•     BCSCTL1 = CALBC1_16MHZ;
•     BCSCTL2 = DIVS_1;  // SMCLK = DCOCLK/2
•     // SPI (USCI) uses SMCLK, prefer SMCLK < 10MHz (SPI speed limit for nRF24 = 10MHz)
• }
```

First Node Implementation

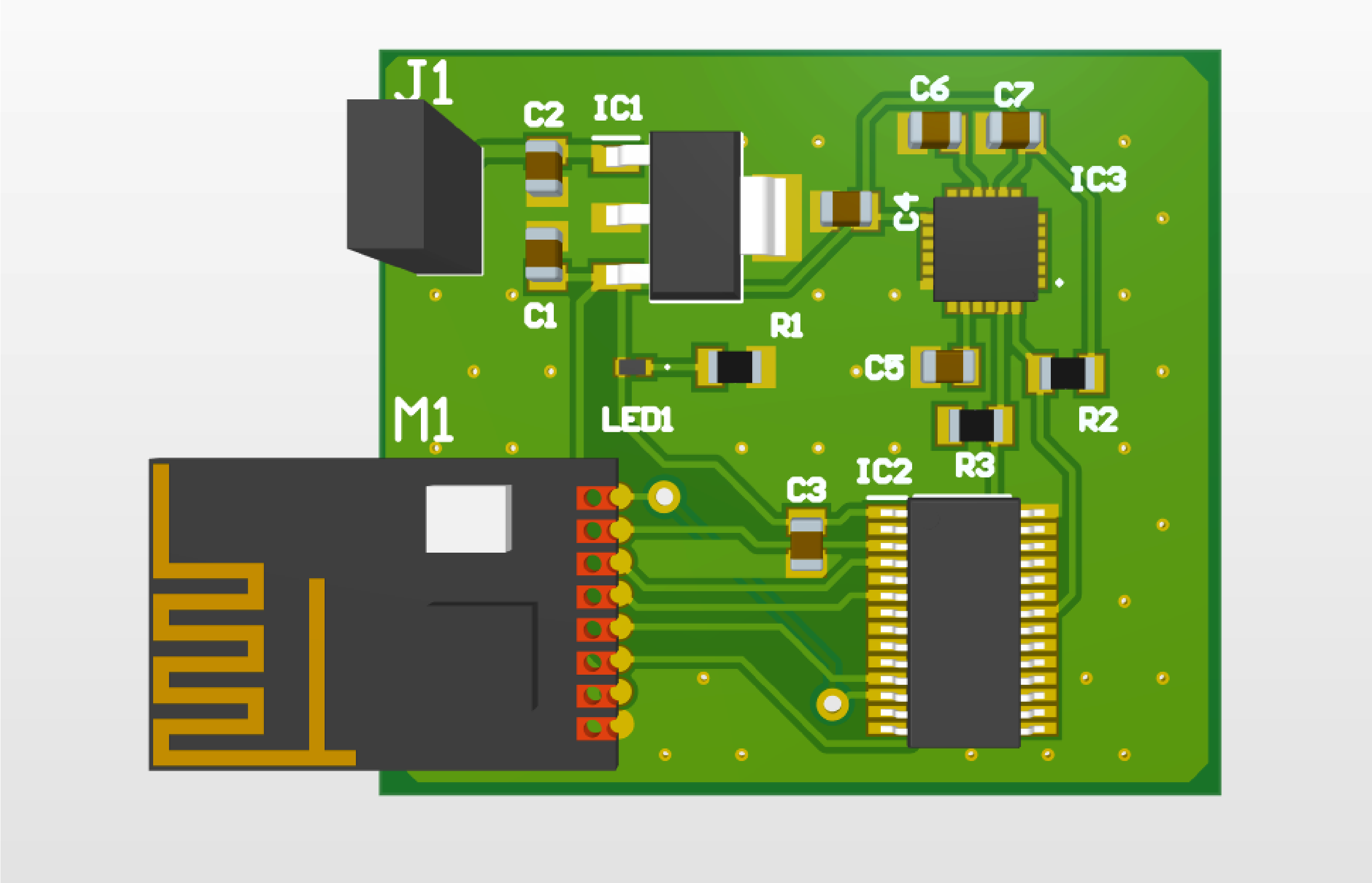
```
• void nrf24_init(void)
• {
•     /* Initial values for nRF24L01+ library config variables */
•        rf_crc = RF24_EN_CRC | RF24_CRCO; // CRC enabled, 16-bit
•        rf_addr_width      = 5;
•        rf_speed_power     = RF24_SPEED_1MBPS | RF24_POWER_0DBM;
•        rf_channel         = 120;

•        msprf24_init();  // All RX pipes closed by default
•        msprf24_set_pipe_packetsize(0, 32);
•        msprf24_open_pipe(0, 1);  // Open pipe#0 with Enhanced ShockBurst enabled for receiving Auto-ACKs
•            // Note: Pipe#0 is hardcoded in the transceiver hardware as the designated "pipe" for a TX node to receive
•            // auto-ACKs.  This does not have to match the pipe# used on the RX side.

•        // Transmit to 'rad01' (0x72 0x61 0x64 0x30 0x31)
•        msprf24_standby();
•        user = msprf24_current_state();
•        addr[0] = 0xDE; addr[1] = 0xAD; addr[2] = 0xBE; addr[3] = 0xEF; addr[4] = 0x00;
•        w_tx_addr(addr);
•        w_rx_addr(0, addr);  // Pipe 0 receives auto-ack's, autoacks are sent back to the TX addr so the PTX node
•                             // needs to listen to the TX addr on pipe#0 to receive them.

• }
• void init_timer()
• {
•     //Configure Timers
•     TA0CCR0 = 50000;
•     TA0CTL |= MC_1 | ID_0 | TASSEL_2 | TACLR;
•     TACCTL0 = CCIE;
• }
```

First Node Implementation

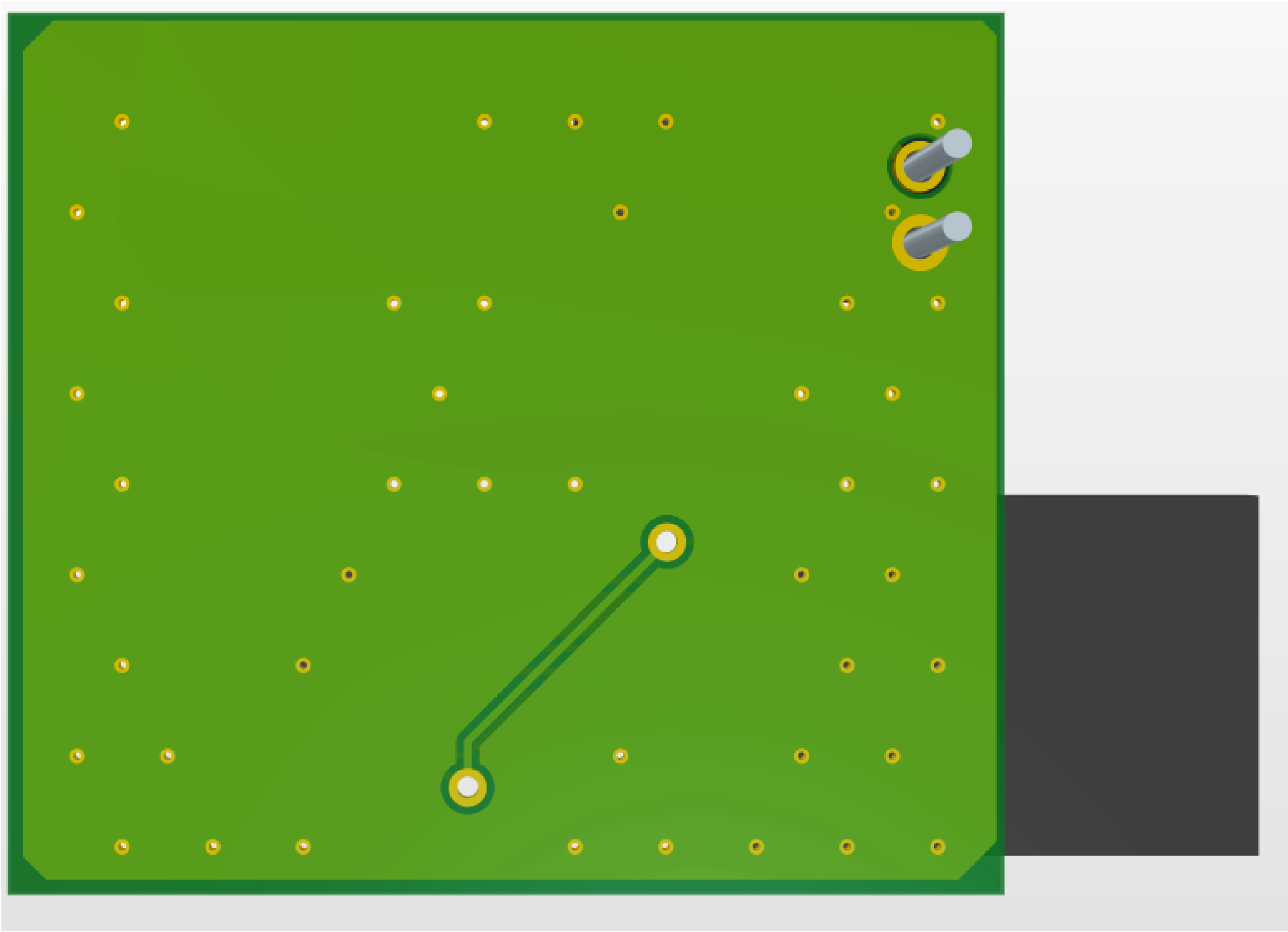
```
•   #pragma vector=TIMER0_A0_VECTOR
•   __interrupt void TIMER_ISR(void)
•   {
•       timer_counter++;
•       if(timer_counter >= 16)
•       {
•           __bic_SR_register_on_exit(LPM4_bits);    // Wake up
•           timer_counter = 0;
•       }
•       TACCTL0 &= ~CCIFG;
•   }
```

```
#pragma vector = PORT2_VECTOR
  __interrupt void P2_IRQ (void) {
   if(P2IFG & nrfIRQpin) {
      __bic_SR_register_on_exit(LPM4_bits);    // Wake up
      rf_irq |= RF24_IRQ_FLAGGED;
      P2IFG &= ~nrfIRQpin;   // Clear interrupt flag
   }
}
```

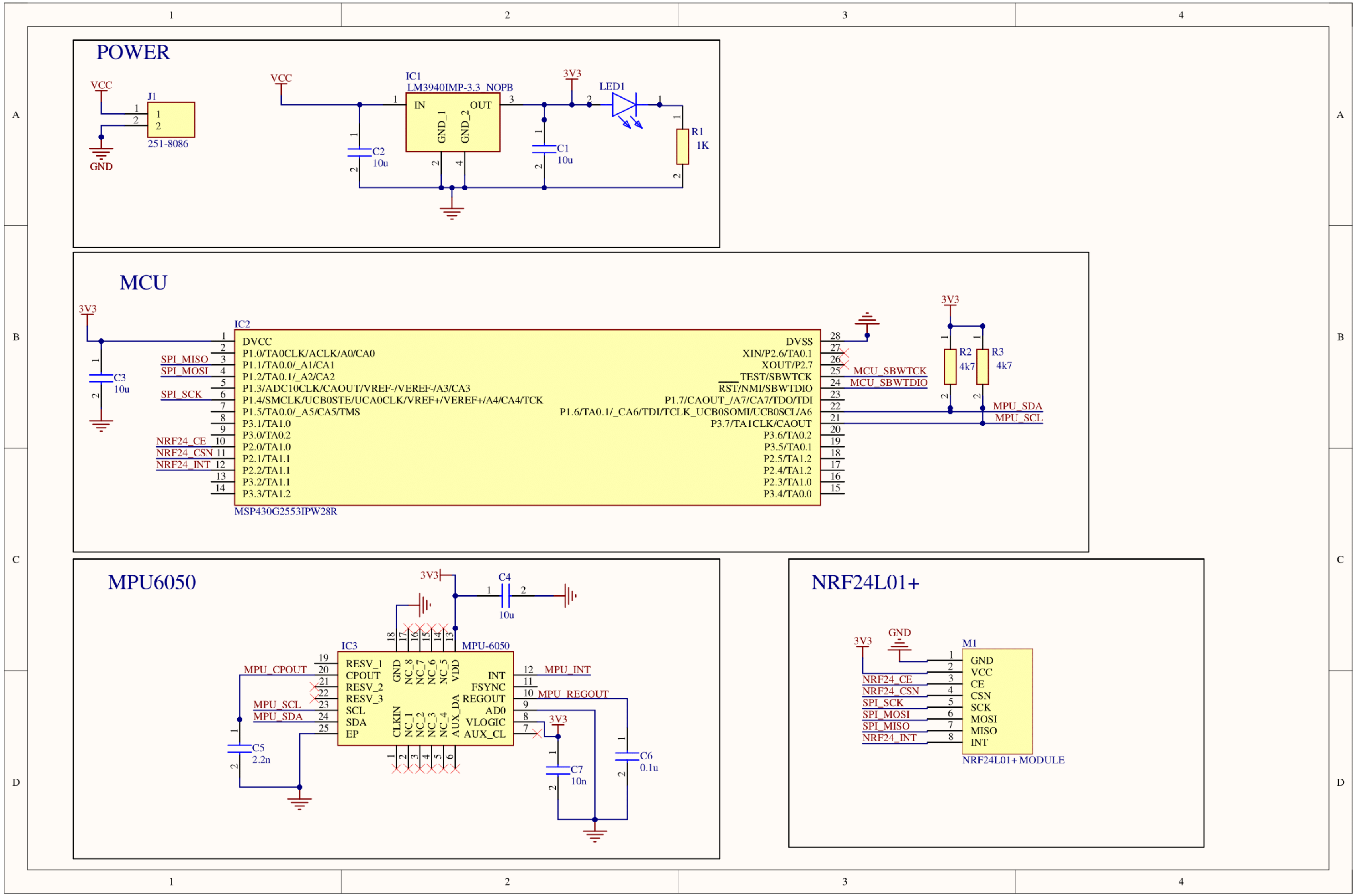
First Node PCB Design



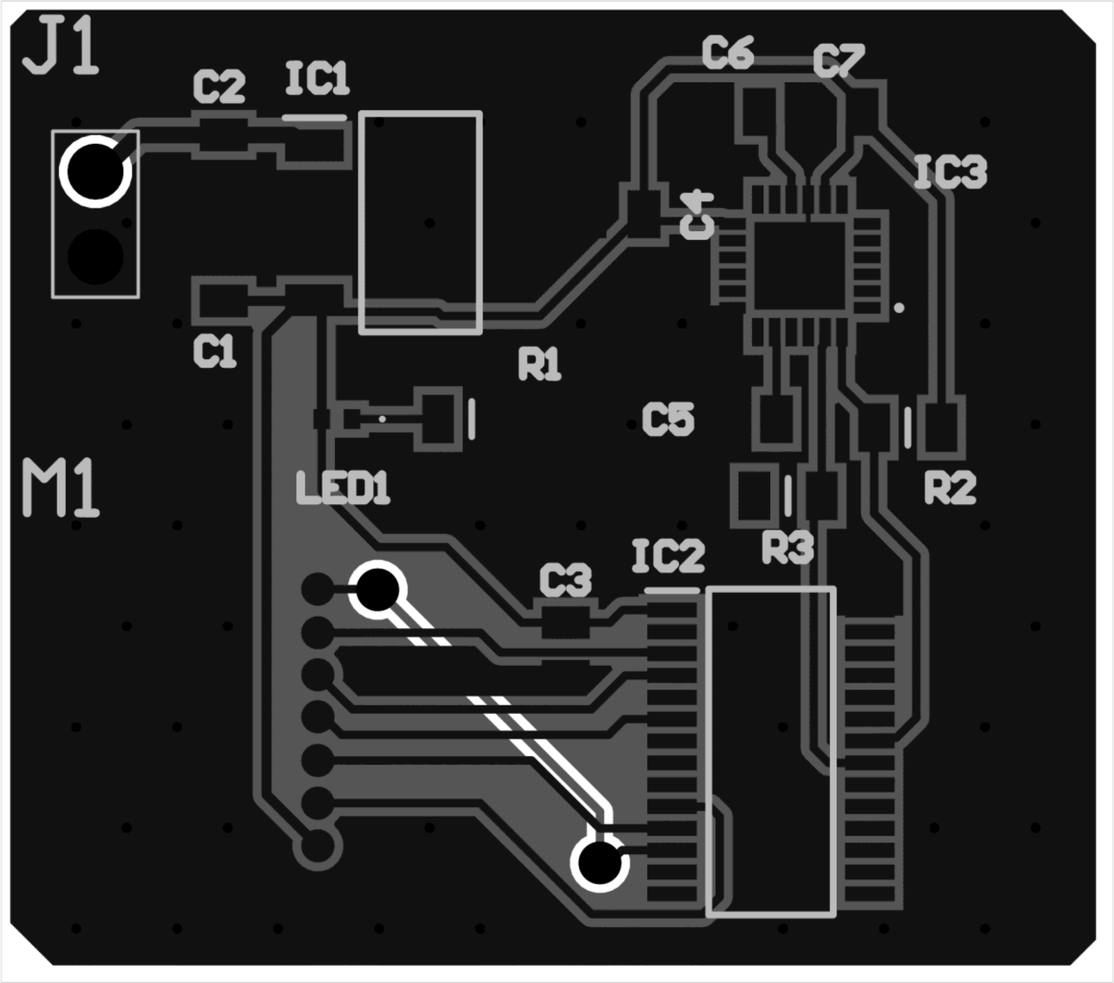
First Node PCB Design



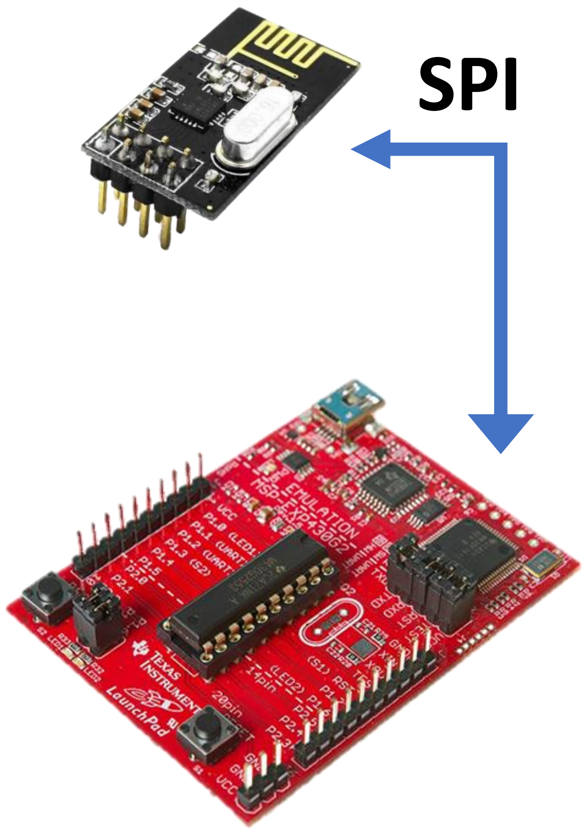
First Node PCB Schematic



First Node PCB Layout

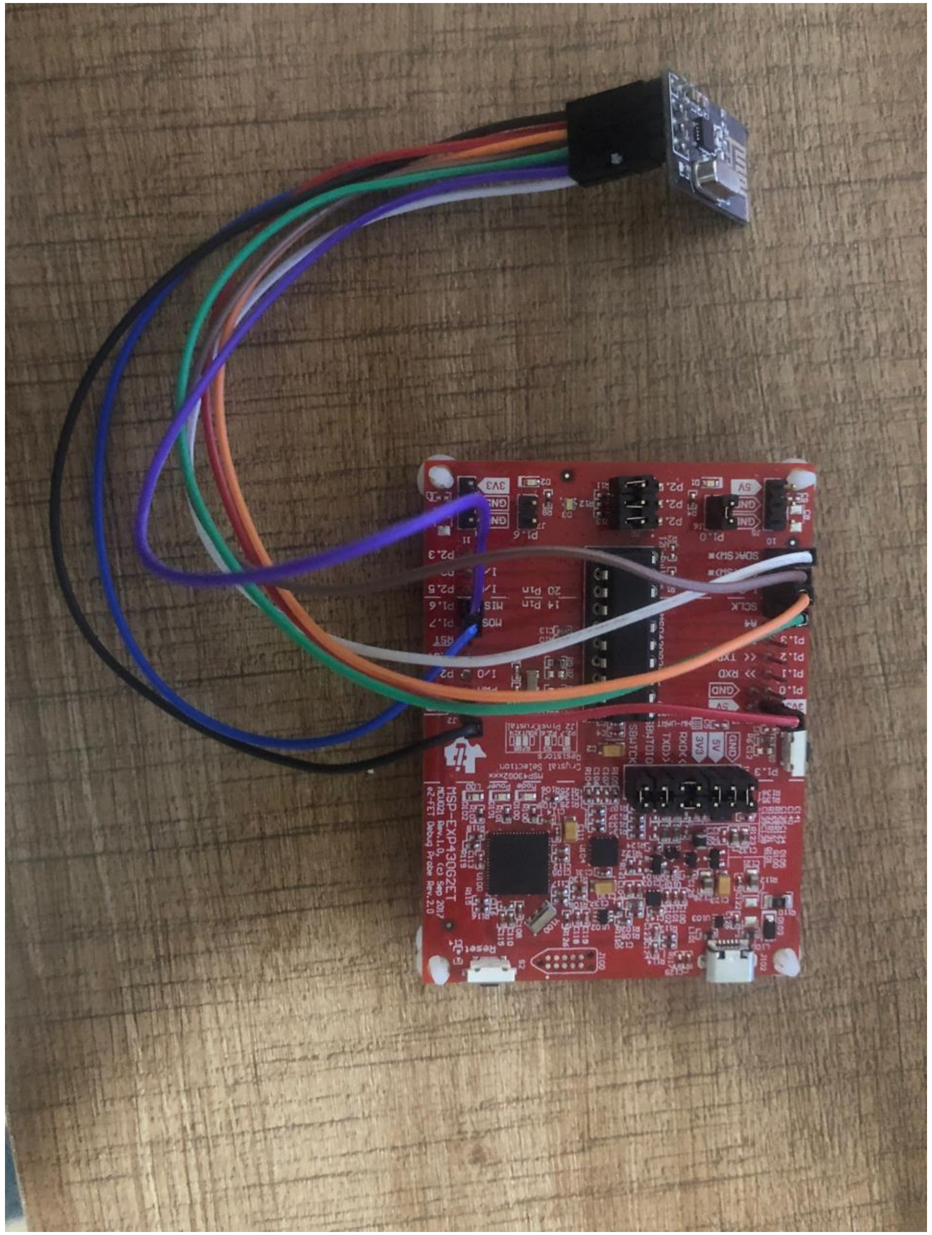


Second Node - Oğulcan Uğuroğlu

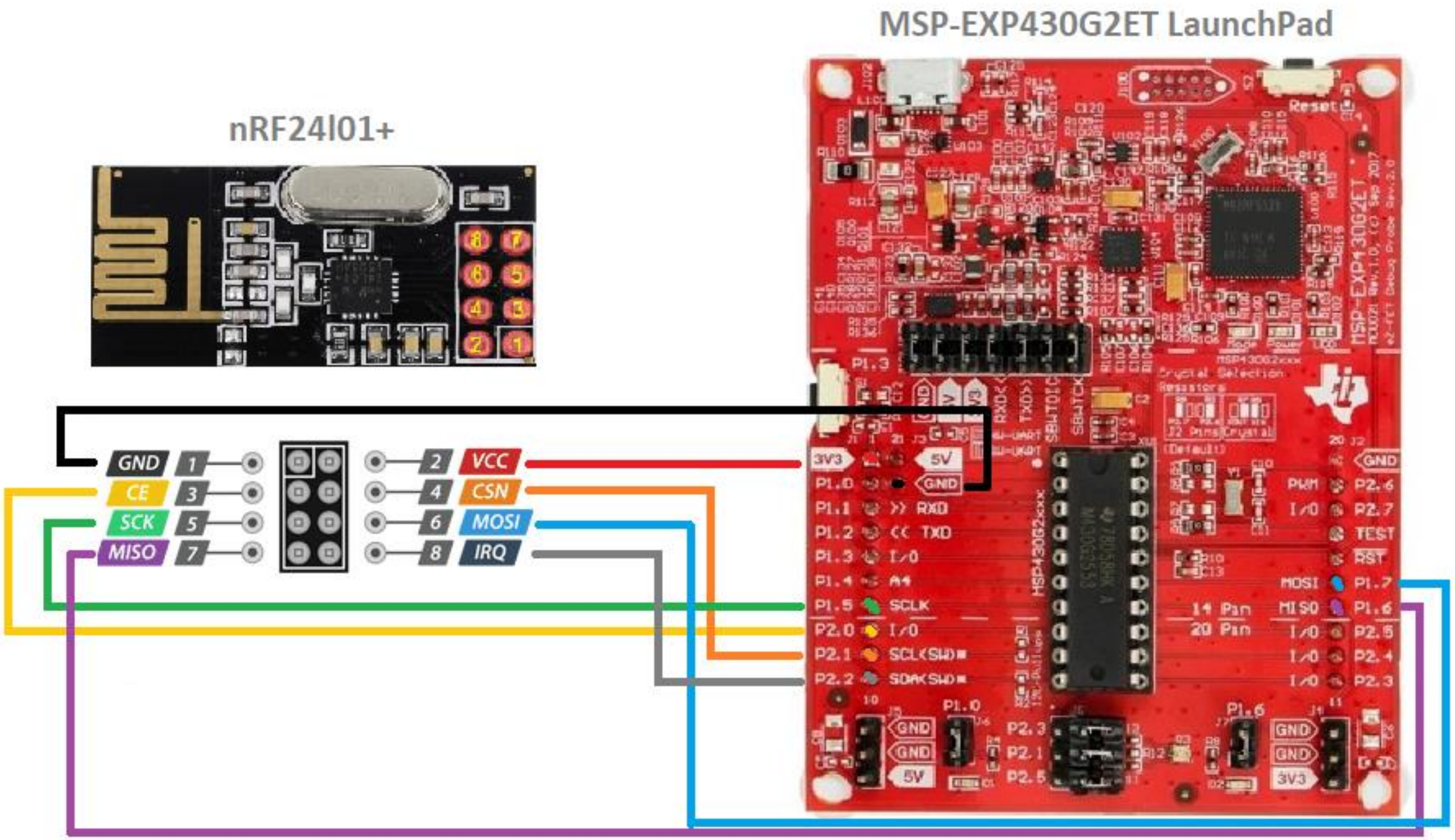


- Main functions of Node2:
- Receiving tha data from the node 1.
- Decrypt the data.
- Transmit the data to node 3.

Second Node Connections



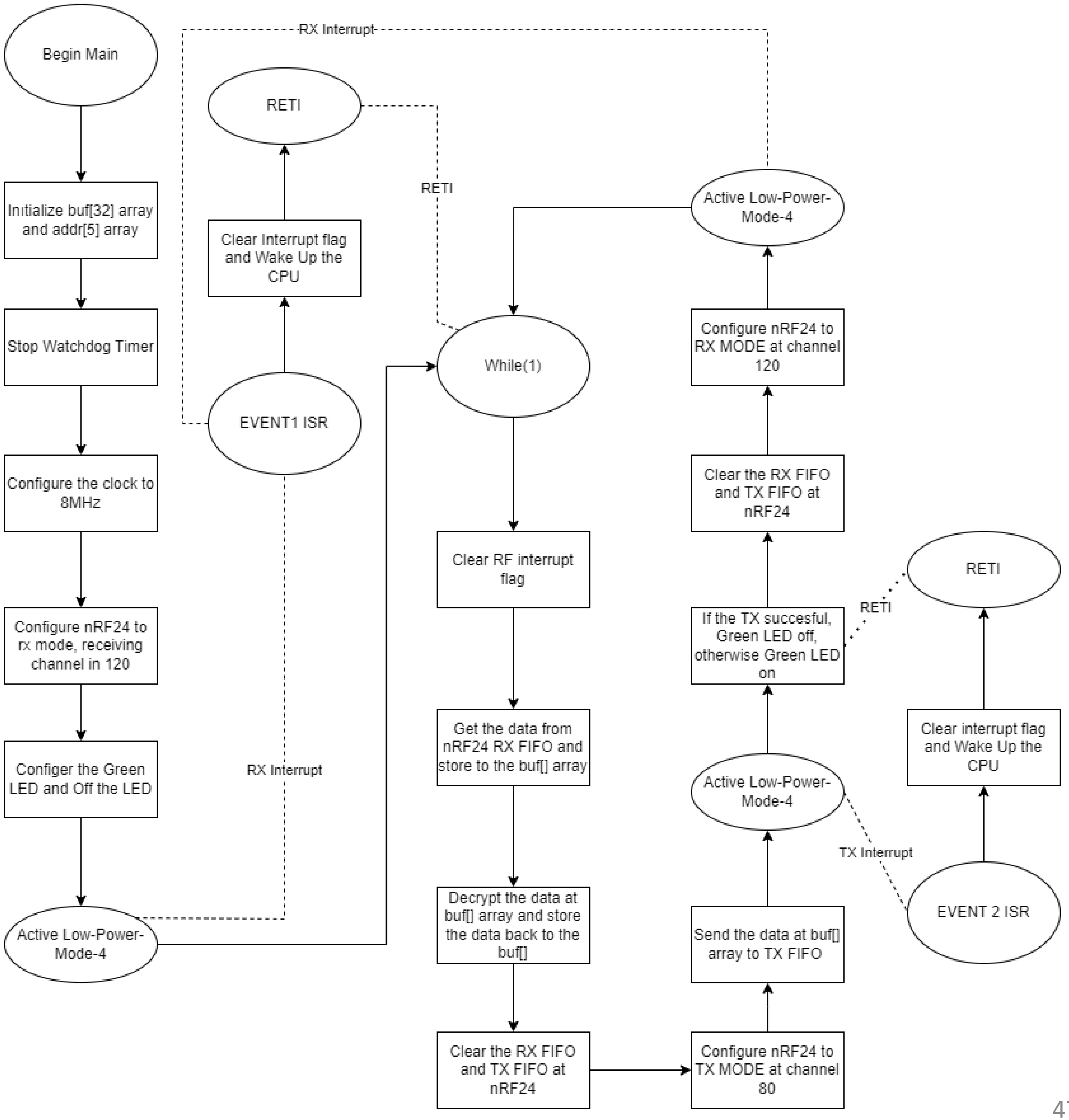
Second Node nRF24 Connections



The USCI_B0 module in the MSP-exp430G2ET launchPad is used for communicate with nRF24 in SPI mode. To use SPI mode, pin configurations were connected as you can see in the table.

The master is MSP430 and the slave is nRF24l01+

Flowchart



Begin Main
Initialize buf[32] array and addr[5] array
Stop Watchdog Timer
Configure the clock to 8MHz
Configure nRF24 to rx mode, receiving channel in 120
Configer the Green LED and Off the LED
Active Low-Power-Mode-4
RX Interrupt
EVENT1 ISR
Clear Interrupt flag and Wake Up the CPU
RETI
RETI
RX Interrupt
While(1)
Clear RF interrupt flag
Get the data from nRF24 RX FIFO and store to the buf[] array
Decrypt the data at buf[] array and store the data back to the buf[]
Clear the RX FIFO and TX FIFO at nRF24
Configure nRF24 to TX MODE at channel 80
Send the data at buf[] array to TX FIFO
Active Low-Power-Mode-4
TX Interrupt
EVENT 2 ISR
Clear interrupt flag and Wake Up the CPU
RETI
RETI
If the TX succesful, Green LED off, otherwise Green LED on
Clear the RX FIFO and TX FIFO at nRF24
Configure nRF24 to RX MODE at channel 120
Active Low-Power-Mode-4

Node 2 Code Implementations

```
#include <msp430.h>
#include "msprf24.h"
#include "nrf_userconfig.h"
#include "stdint.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "aes.h"

volatile unsigned int user;

uint8_t buf[32];
uint8_t addr[5];

void Clock_init(void);
void LED_init(void);
void configure_rx(int channel);
void configure_tx(int channel);

void set_channel(uint8_t rf_channel);
void LED_On(void);
void LED_Off(void);

void AES_128_ECB_encrypt(uint8_t *in);
void AES_128_ECB_decrypt(uint8_t *in);

void main(void)
{
WDTCTL = WDTPW | WDTHOLD;// stop watchdog timer

Clock_init();                 // Adjust clock 8MHz
    configure_rx(120);            // configure rx
LED_init();
LPM4;                         // Wait interrupt at LPM4
```

```
while(1)
    {
    if (rf_irq & RF24_IRQ_FLAGGED)
    {
        rf_irq &= ~RF24_IRQ_FLAGGED;        // Clear RF interrupt flag
        msprf24_get_irq_reason();
    }
    r_rx_payload(32, buf);
    msprf24_irq_clear(RF24_IRQ_RX);
    AES_128_ECB_decrypt(buf);           // Decrypte the received datas and store at
//buf[] array
    user = buf[0];
    flush_rx();
    flush_tx();
    configure_tx(80);
       // __delay_cycles(10000);
        w_tx_payload(32, buf); // load payload with data
        msprf24_activate_tx();
        __bis_SR_register(LPM4_bits + GIE); // LPM with interrupts enabled

        if (rf_irq & RF24_IRQ_FLAGGED)  // TX Interrupt flag
        {
            msprf24_get_irq_reason();
            if (rf_irq & RF24_IRQ_TX)   // TX successful
            {
                LED_Off();                // Green LED off
            }
            if (rf_irq & RF24_IRQ_TXFAILED) // TX failed
            {
                LED_On();               // Green LED on
            }
            msprf24_irq_clear(rf_irq); // clear interrupt flag
        }

        flush_rx();
        flush_tx();

        configure_rx(120);          // configure rx
    LPM4;
    }
}
```

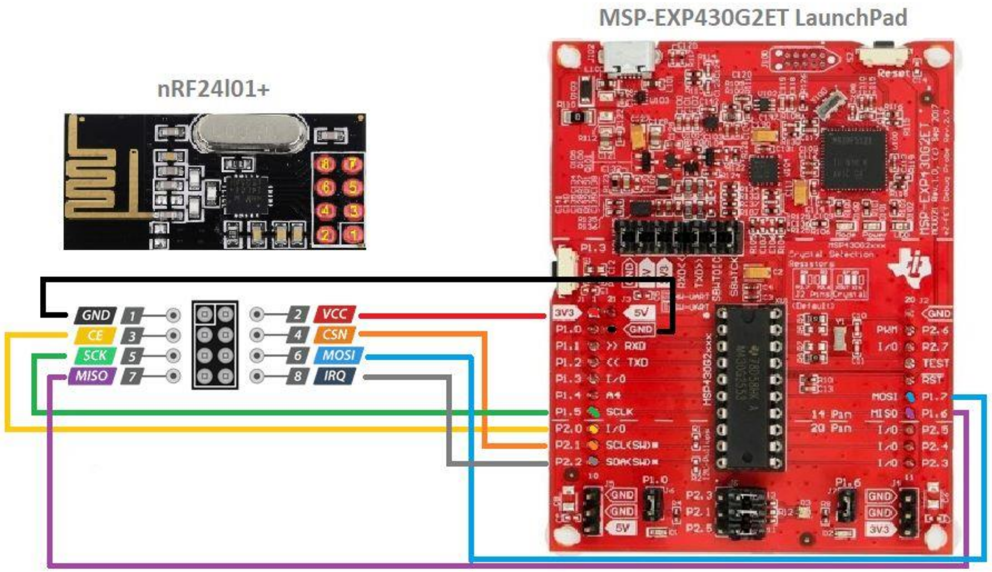
```c
void Clock_init(void)
{
    DCOCTL = CALDCO_16MHZ;
    BCSCTL1 = CALBC1_16MHZ;
    BCSCTL2 = DIVS_1;  // SMCLK = DCOCLK/2
    // SPI (USCI) uses SMCLK, prefer SMCLK < 10MHz (SPI speed limit for nRF24 = 10MHz)
}

void LED_init(void)
{
    P1DIR |= BIT0;      // Green LED output.
    P1OUT &= ~BIT0;
}

void LED_On(void)
{
    P1OUT |= BIT0;       // Green LED on
}

void LED_Off(void)
{
    P1OUT &= ~BIT0;     // Green LED off
}
```

```c
void configure_rx(int channel)
{
    user = 0xFE;

    /* Initial values for nRF24L01+ library config variables */
    rf_crc = RF24_EN_CRC | RF24_CRCO; // CRC enabled, 16-bit
    rf_addr_width = 5;
    rf_speed_power = RF24_SPEED_1MBPS | RF24_POWER_0DBM;
    rf_channel = channel;

    msprf24_init();
    msprf24_set_pipe_packetsize(0, 32);
    msprf24_open_pipe(0, 1);  // Open pipe#0 with Enhanced
//ShockBurst

    // Set our RX address
    addr[0] = 0xDE;
    addr[1] = 0xAD;
    addr[2] = 0xBE;
    addr[3] = 0xEF;
    addr[4] = 0x00;
    w_rx_addr(0, addr);

    // Receive mode
    if (!(RF24_QUEUE_RXEMPTY & msprf24_queue_state()))
    {
        flush_rx();
    }
    msprf24_activate_rx();
}
```

```c
void configure_tx(int channel)
{
    rf_channel = channel;

    msprf24_init();  // All RX pipes closed by default
    msprf24_set_pipe_packetsize(0, 32);
    msprf24_open_pipe(0, 1); // Open pipe#0 with Enhanced ShockBurst
//enabled for receiving Auto-ACKs

    // Transmit to 'rad01' (0x72 0x61 0x64 0x30 0x31)
    msprf24_standby();
    user = msprf24_current_state();
    addr[0] = 0xDE;
    addr[1] = 0xAD;
    addr[2] = 0xBE;
    addr[3] = 0xEF;
    addr[4] = 0x00;
    w_tx_addr(addr);
    w_rx_addr(0, addr); // Pipe 0 receives auto-ack's, autoacks are sent
//back to the TX addr so the PTX node
// needs to listen to the TX addr on pipe#0 to receive them.

}
```

```
void AES_128_ECB_decrypt(uint8_t *in)          // Decrypt datas at
in[] array and store at in[]
{
    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2,
0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };

    struct AES_ctx ctx;

    AES_init_ctx(&ctx, key);
    AES_ECB_decrypt(&ctx, in);
}

void AES_128_ECB_encrypt(uint8_t *in)          // Encrypt datas at
in[] array and store at in[]
{
    struct AES_ctx ctx;

    uint8_t key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2,
0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };

    AES_init_ctx(&ctx, key);
    AES_ECB_encrypt(&ctx, in);
}
```

Third Node – Hasan Can Sert



Main features:

- Taking data to RX buffer with using NRF401L1+ sensor.
- Used pins are P1.0, VCC, P1.5, P1.6, P1.7, P2.0, P2.1, P2.2
- Sending the data to PC COM port with UART communication through usb cable.
- Creating a Simulation enviroment in tthe PC enviroment

Configuring Baud Rate

Clock	Baudrate	UCBRx	UCBRSx	UCBRFx	UCOS16
1,048,576 Hz	9600	109	2	0	0
8MHz	115200	69	4	0	1
16MHz	115200	8	0	11	1

Obtaining the main divisor is often easy, but adjusting modulation and other bits can be tricky since they need careful adjustment to reduce error. TI provides a list of register values for common clock frequencies that makes baud rate configuration easier.

Configuring CLOCK

The clock source for the UART baud generator is SMCLK sourcing the DCO running at 1MHz. Before we can configure the UART peripheral we need to place it in reset mode. Not all registers require this but it is best to do so when first configuring USCI, whether it's for UART or any other mode. Notice that we use the assignment operator, so all the other bits are set to zero. With the reset in place, we make SMCLK the clock source for the UART. Being flexible, there are other possible options for the UCSSELx:

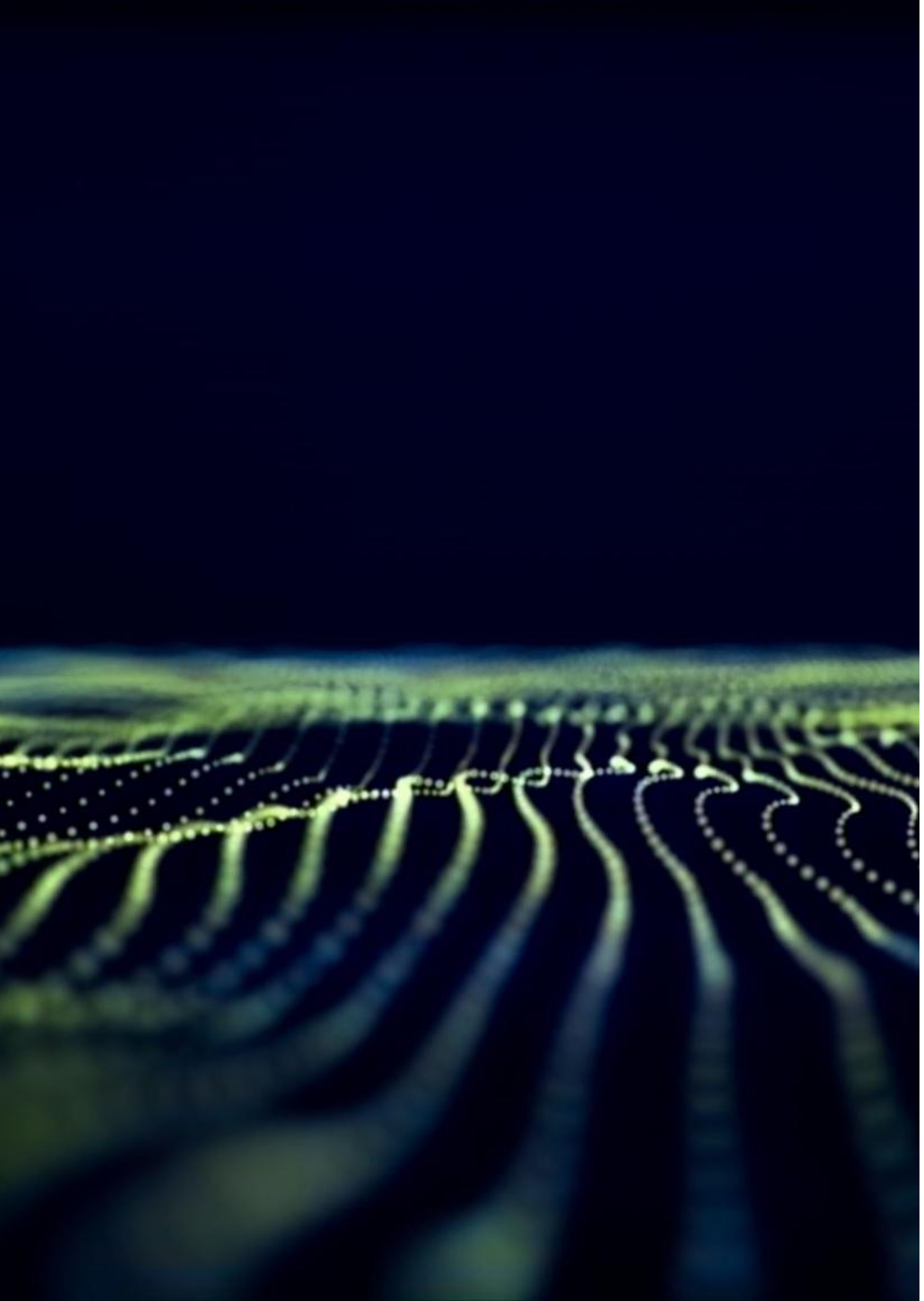
00b = UCAxCLK (external USCI clock)

01b = ACLK

10b = SMCLK

11b = SMCLK

UART can actually use a clock coming on a pin instead of one of the internally generated clocks. This can be useful in reducing the number of clocks in the system and reducing system cost.



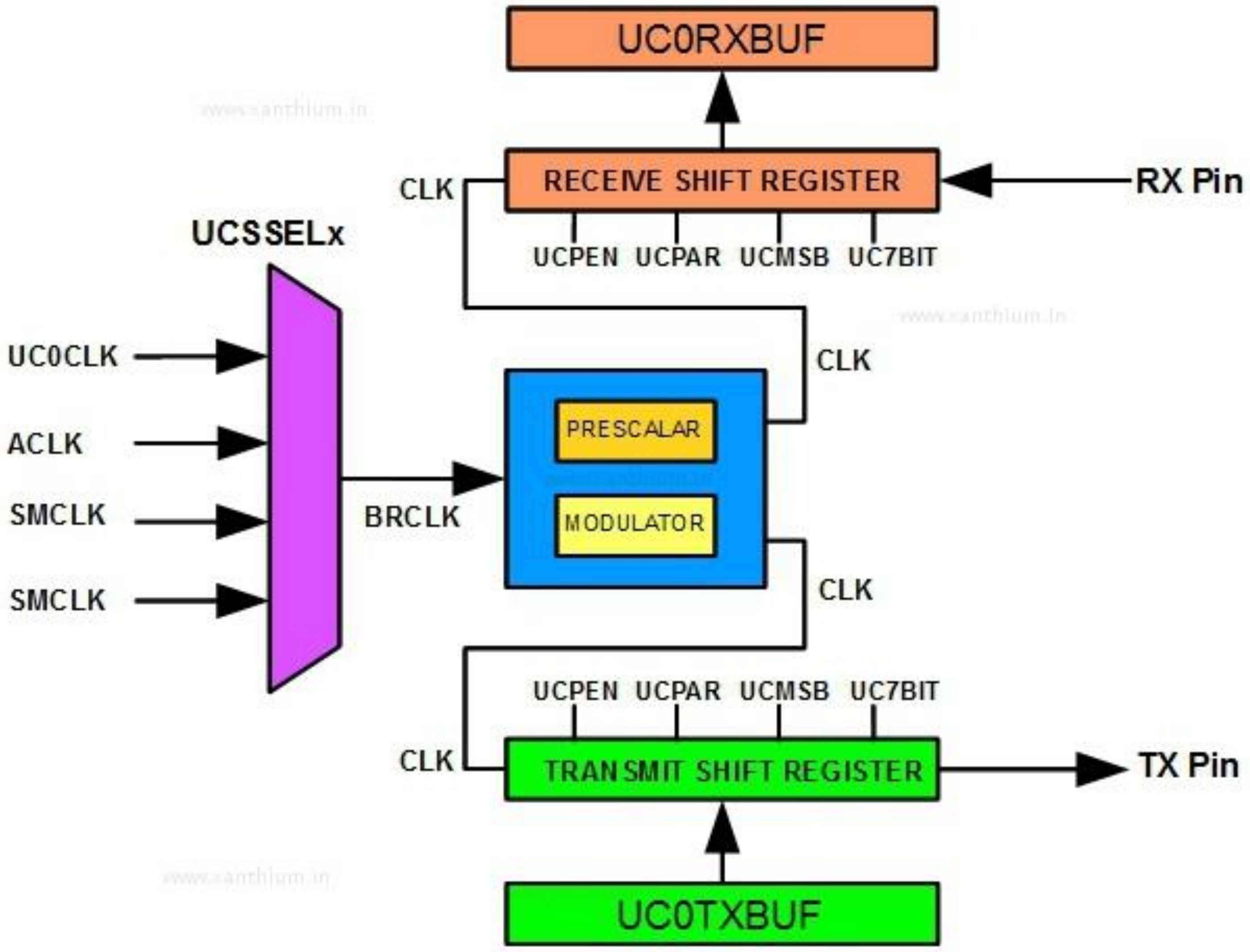
UART Interrupts

UART includes two main interrupt sources: RX and TX. The RX interrupt fires when a character is received and has been placed into the buffer, whereas the TX interrupt is set when the TX buffer is available to be filled with a data.

In the RX ISR, the RXIFG receive flag is cleared automatically when RXBUF is read. Always read RXBUF or clear the flag. Otherwise, the flag remains set after the ISR returns and the interrupt will immediately trigger and stay in a loop.

If The data in the RXBUF doesn't need it might also be useful to disable the RXIE interrupt enable. Our approach with the RX ISR is to store the received character in an array and process all the received characters once all have been received.

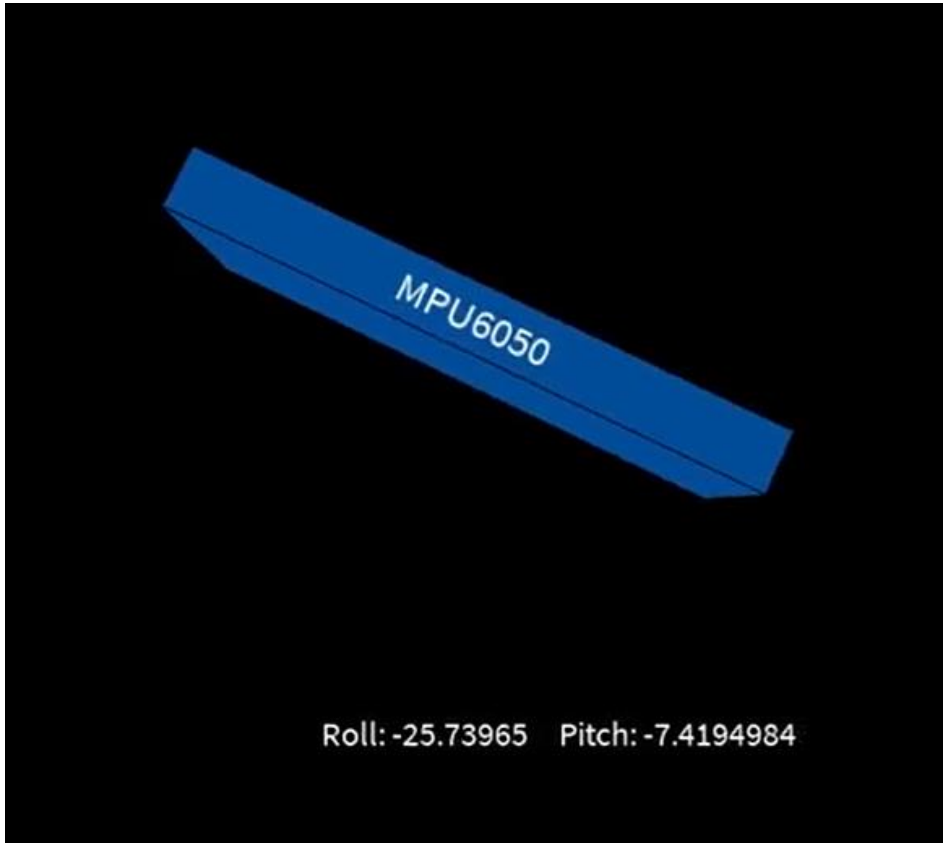
Block diagram of USCI_A0 in UART mode

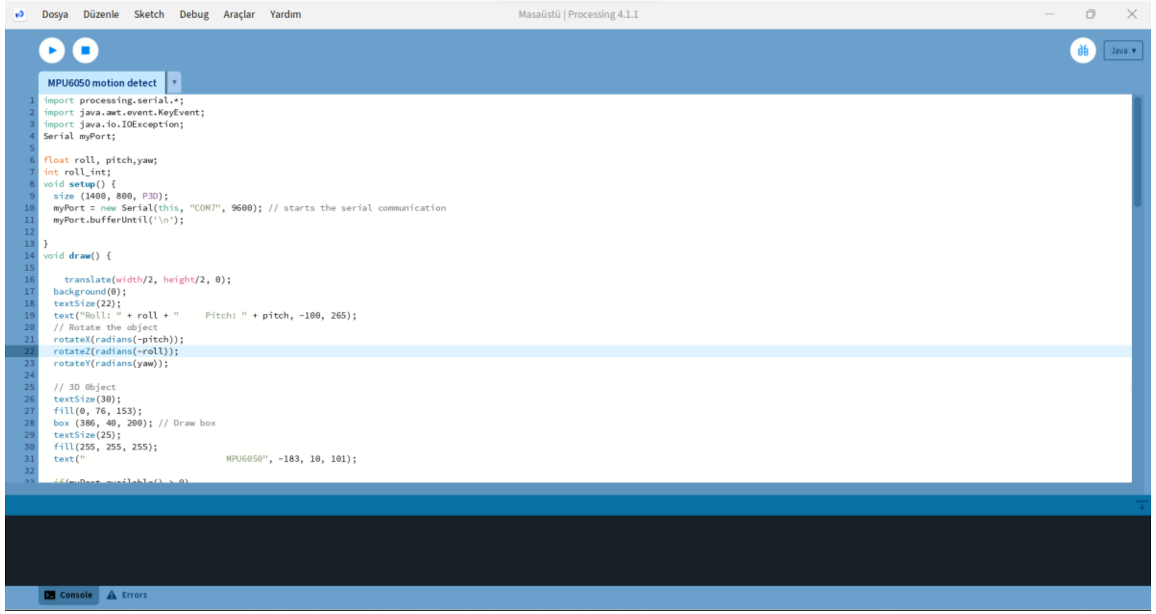


Implementation

```
#include <msp430.h>
#include "msprf24.h"
#include "nrf_userconfig.h"
#include "stdint.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define CHAR_BUFF_SIZE  20
volatile unsigned int user;
float temp_C;
float angle_x;
float angle_y;
char buffer_x[CHAR_BUFF_SIZE], buffer_y[CHAR_BUFF_SIZE];
uint8_t temp[sizeof(float)];
char UART_data[14];
char x;
char y;
int data;
unsigned int counter;
typedef enum {
    false,
    true
}bool;
void TXString( char* string, int length )
{
  int pointer;
  for( pointer = 0; pointer < length; pointer++)
  {
    UCA0TXBUF = string[pointer];
    while (!(IFG2&UCA0TXIFG));              // USCI_A0 TX buffer ready?
  }
}
```

```c
int main()
{
   uint8_t addr[5];
   uint8_t buf[32];

   WDTCTL = WDTHOLD | WDTPW;

   DCOCTL = CALDCO_16MHZ;
   BCSCTL1 = CALBC1_16MHZ;
   BCSCTL2 = DIVS_1;  // SMCLK = DCOCLK/2
   // SPI (USCI) uses SMCLK, prefer SMCLK < 10MHz (SPI speed limit for nRF24 =
10MHz)

   P1SEL = BIT1 + BIT2 ;                   // P1.1 = RXD, P1.2=TXD
   P1SEL2 = BIT1 + BIT2 ;                   // P1.1 = RXD, P1.2=TXD

   UCA0CTL1 |= UCSWRST;
   UCA0CTL1 |= UCSSEL_2;                   // CLK = SMCLK

   UCA0BR0 = 0x34;                         // 8Mhz/9600
   UCA0BR1 = 0x00;                         //
   UCA0MCTL = 0x11;                         // Modulation UCBRSx = 3 UCBRS1 + UCBRS0
   UCA0CTL1 &= ~UCSWRST;                    // **Initialize USCI state machine**
  // IE2 |= UCA0TXIE;                       // Enable USCI_A0 TX interrupt

   // Red LED will be our output
   P1DIR |= BIT0+BIT6;
   P1OUT &= ~(BIT0+BIT6);

   user = 0xFE;

   /* Initial values for nRF24L01+ library config variables */
   rf_crc = RF24_EN_CRC | RF24_CRCO; // CRC enabled, 16-bit
   rf_addr_width      = 5;
   rf_speed_power     = RF24_SPEED_1MBPS | RF24_POWER_0DBM;
   rf_channel         = 120;

   msprf24_init();
   msprf24_set_pipe_packetsize(0, 32);
   msprf24_open_pipe(0, 1);  // Open pipe#0 with Enhanced ShockBurst

   // Set our RX address
   addr[0] = 0xDE; addr[1] = 0xAD; addr[2] = 0xBE; addr[3] = 0xEF; addr[4] = 0x00;
   w_rx_addr(0, addr);

// Receive mode
  if (!(RF24_QUEUE_RXEMPTY & msprf24_queue_state())) {
     flush_rx();
  }
  msprf24_activate_rx();
  LPM4;

  while (1) {
     if (rf_irq & RF24_IRQ_FLAGGED) {
        rf_irq &= ~RF24_IRQ_FLAGGED;
        msprf24_get_irq_reason();
     }
     if (rf_irq & RF24_IRQ_RX || msprf24_rx_pending()) {
        r_rx_payload(32, buf);
        msprf24_irq_clear(RF24_IRQ_RX);

        temp[0] = buf[2];
        temp[1] = buf[3];
        temp[2] = buf[4];
        temp[3] = buf[5];
        temp_C = *(float *)&temp;
        temp[0] = buf[6];
        temp[1] = buf[7];
        temp[2] = buf[8];
        temp[3] = buf[9];
        UART_data[0] = buf[6];
        UART_data[1] = buf[7];
        UART_data[2] = buf[8];
        UART_data[3] = buf[9];
        angle_x = *(float *)&temp;
        UART_data[4] = '/';

        temp[0] = buf[10];
        temp[1] = buf[11];
        temp[2] = buf[12];
        temp[3] = buf[13];
UART_data[5] = buf[10];
        UART_data[6] = buf[11];
        UART_data[7] = buf[12];
        UART_data[8] = buf[13];
        angle_y = *(float *)&temp;

        UART_data[13] = '\n';
        TXString(UART_data, 14);
        user = buf[0];

        if (buf[0] == '0')
           P1OUT &= ~BIT0;
        if (buf[0] == '1')
           P1OUT |= BIT0;
        if (buf[1] == '0')
           P1OUT &= ~BIT6;
        if (buf[1] == '1')
           P1OUT |= BIT6;

     } else {
        user = 0xFF;
     }
     LPM4;
  }
  return 0;
}
```



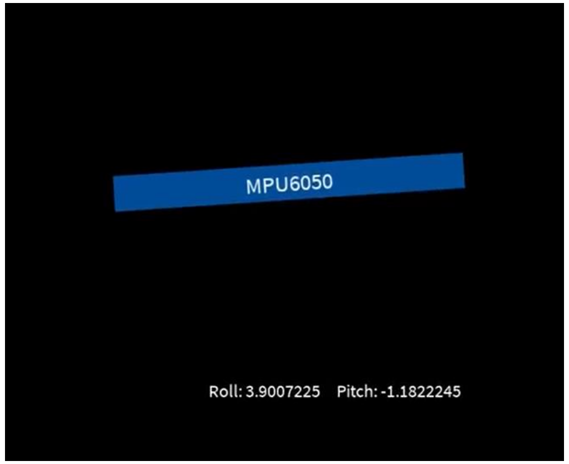


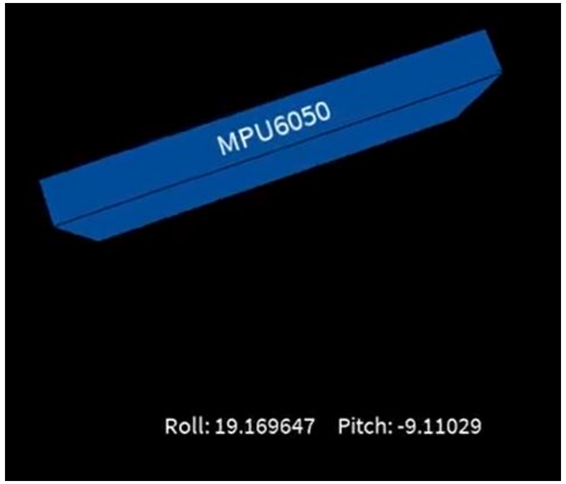
Processing IDE

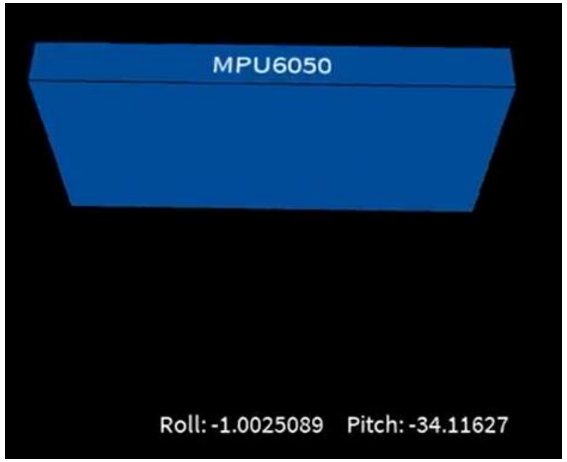
Java Code to create simulation enviroment

```
import processing.serial.*;
import java.awt.event.KeyEvent;
import java.io.IOException;
Serial myPort;

float roll, pitch,yaw;
int roll_int;
void setup() {
  size (1400, 800, P3D);
  myPort = new Serial(this, "COM7", 9600); // starts the serial communication
  myPort.bufferUntil('\n');

}
void draw() {

    translate(width/2, height/2, 0);
  background(0);
  textSize(22);
  text("Roll: " + roll + "     Pitch: " + pitch, -100, 265);
  // Rotate the object
  rotateX(radians(-pitch));
  rotateZ(radians(-roll));
  rotateY(radians(yaw));

  // 3D 0bject
  textSize(30);
  fill(0, 76, 153);
  box (386, 40, 200); // Draw box
  textSize(25);
  fill(255, 255, 255);
  text("                    MPU6050", -183, 10, 101);
```

```
if(myPort.available() > 0)
  {
     byte[] data_bytes = new byte[10];
     // reads the data from the Serial Port up to the character '.' and puts it into the String variable "data".
     String data = myPort.readStringUntil('\n');
     // if you got any bytes other than the linefeed:
     if (data != null) {
    data = trim(data);

    // split the string at "/"
    String items[] = split(data, '/');
    if (items.length > 1) {
     //--- Roll,Pitch in degrees
     data_bytes = items[0].getBytes();
     println(items[0]);
     if(items[0].length() > 3 && items[1].length() > 3)
     {
       data_bytes = items[0].getBytes();
       roll = get4bytesFloat(data_bytes, 0);
       println(roll);


       data_bytes = items[1].getBytes();
       pitch = get4bytesFloat(data_bytes, 0);
       println(pitch);
          }   } }  } }

float get4bytesFloat(byte[] data, int offset) {
  String hexint=hex(data[offset+3])+hex(data[offset+2])+hex(data[offset+1])+hex(data[offset]);
  return Float.intBitsToFloat(unhex(hexint));
}
```







Monitoring angular positions and values with Processing IDE

Conclusion

Firstly, we used data obtained from the mpu6050 and calculate the pitch and row angles of the model plane. Then, we encrypt the data and send to the other MSP430. Cryptology process is critical for data protection and non-theft. The second MSP430, receive and decryt the data for sending the third MSP430. In the last node, the received data are sending to the computer to animates the movement of the model airplane with using Processing IDE.

The movement monitoring of planes is important:

-Navigation: Accurate movement monitoring is critical for navigation.

-Maintenance: Monitoring the movement of planes can also provide valuable data for maintenance purposes. For example, analyzing the accelerometer and gyroscope data can help detect potential mechanical issues or wear and tear on the aircraft.

-Efficiency: By monitoring planes' movements, air traffic controllers can optimize the use of airspace and avoid delays, which improves overall efficiency of the air traffic system.

References

1. Mummadi, C.K.; Leo, F.P.P.; Verma, K.D.; Kasireddy, S.; Scholl, P.M.; Kempfle, J.; van Laerhoven, K. Real-time and embedded detection of hand gestures with an IMU-based glove. Informatics 2018, 5, 28. https://doi.org/10.3390/informatics5020028.
2. MEMS based IMU for tilting measurement: Comparison of complementary and kalman filter based data fusion Pengfei Gui, Liqiong Tang, S. Mukhopadhyay.
3. https://www.youtube.com/watch?v=whSw42XddsU&ab_channel=BrianDouglas – 14.01.2023
4. Complementary Filter Design for Angle Estimation using MEMS Accelerometer and Gyroscope Hyung Gi Min and Eun Tae Jeung Robotics Lab., NTREX Ltd., co., Juan-Dong 5-38, Nam-Gu, Incheon, Korea Department of Control and Instrumentation, Changwon National University, Changwon, 641-773, Korea
5. https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/ - 14.01.2023
6. https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html - 14.01.2023
7. https://fkeng.blogspot.com/2018/05/digital-implementation-of-complementary.html - 14.01.2023

Video Link

https://youtu.be/OB8uu2KnHwE