I2C DRIVER REPORT

Names	IDS
Aya Ali	20010380
Kilorous Mosa	20011126
Martina Azmy	20011133
Basma Barsom	20010398
Ali Ibrahim	20010930
Amr Reda	20011041
Omar Hisham	20011031
Maryamme Ashraf	20011143

Contents

Chapter 1:

Theory:

I2C, or Inter-Integrated Circuit, is a popular serial communication protocol that allows multiple devices to communicate with each other using only two wires: a clock line (SCL) and a data line (SDA). The STM32F401xx is a series of microcontrollers from STMicroelectronics that includes support for I2C communication.

Features:

The I2C (Inter-Integrated Circuit) peripheral in the STM32F401xx microcontroller series comes with various features that enable easy and flexible communication between devices. Below are some of the key features of the I2C implementation in the STM32F401xx microcontrollers.

Hardware I2C Peripheral:

The STM32F401xx microcontrollers feature a dedicated hardware I2C peripheral, streamlining the implementation of I2C communication.

Multi-Master and Multi-Slave Support:

I2C in STM32F401xx supports a multi-master and multi-slave configuration, enabling multiple devices to communicate concurrently on the same bus.

Programmable Clock Speed:

Users can configure the clock speed of the I2C communication, allowing flexibility to adapt to the requirements of connected devices.

7-bit and 10-bit Addressing:

The I2C peripheral supports both 7-bit and 10-bit addressing modes, providing options for device addressing on the bus, we use 7-bit.

Interrupt-Driven Operation:

STM32F401xx's I2C peripheral facilitates interrupt-driven communication, allowing the microcontroller to respond to events like data reception or transmission completion without continuous polling.

DMA (Direct Memory Access) Support:

DMA capability is incorporated for efficient data transfer, minimizing CPU intervention and enhancing overall system performance.

Configurable Acknowledge (ACK/NACK):

The I2C peripheral allows the master to configure the acknowledge bit, determining whether to acknowledge or not acknowledge the reception of data from a slave.

Repeated Start Condition:

Support for the repeated start condition permits the master to initiate a new data transfer without relinquishing control of the bus.

SMBus Support:

Compatibility with SMBus, a subset of I2C, adds support for system management protocols.

Clock Stretching:

The I2C peripheral supports clock stretching, enabling a slave device to slow down the clock, thereby delaying data transfer.

Digital Noise Filter:

Built-in digital noise filter enhances communication robustness in environments with high levels of electrical noise.

Bus Error Detection:

The I2C peripheral is capable of detecting and reporting bus errors, such as conflicting attempts to use the bus simultaneously.

Fast Mode Plus (Fm+) Support:

STM32F401xx's I2C supports the Fast Mode Plus (Fm+) specification, allowing for higher-speed data rates of up to 1 Mbps.

Configurable Rise Time:

Rise time control is configurable, accommodating different bus capacitances and configurations.

Wake-Up from Stop Mode:

The I2C peripheral can wake up the microcontroller from Stop mode in response to an I2C event.

ICs which support i2c:

The I2C protocol is commonly found in many microcontrollers, including popular ones such as the Arduino boards, Raspberry Pi, ESP8266, ESP32,Microship microcontrollers, ARM microcontrollers, and many others. It is a widely used communication protocol for connecting various peripheral devices to microcontrollers. If you have a specific microcontroller in mind, I can provide more detailed information about its I2C capabilities.

Chapter 2:

Slave Addressing Transmission:

In 7-bit addressing mode, one address byte is sent. As soon as the address byte is sent, The ADDR bit is set by hardware and an interrupt is generated if the ITEVFEN bit is set. Then the master waits for a read The master can decide to enter Transmitter or Receiver mode depending on the LSB of the slave address sent.

• In 7-bit addressing mode, to enter Transmitter mode, a master sends the slave address with LSB reset. To enter Receiver mode, a master sends the slave address with LSB set. of the SR1 register followed by a read of the SR2 register.

Master transmitter:

Following the address transmission and after clearing ADDR, the master sends bytes from the DR register to the SDA line via the internal shift register. The master waits until the first data byte is written into I2C_DR. When the acknowledge pulse is received, the TxE bit is set by hardware and an interrupt is generated if the ITEVFEN and ITBUFEN bits are set. If TxE is set and a data byte was not written in the DR register before the end of the last data transmission, BTF is set and the interface waits until BTF is cleared by a write to I2C_DR, stretching SCL low.

Slave / Master receive the data:

The data will be transferred from the SDA line and go through noise filter then go to the data control unit which push it to the shift register that push it to the data register.

Slave/ Master transmit the data:

The data will transfer from the data register in the Micro controller prephrial to the data shift register then pass on data control and noise filter.

I2C Master:

By default the I2C interface operates in Slave mode. To switch from default Slave mode to Master mode a Start condition generation is needed. The peripheral input clock must be programmed in the I2C_CR2 register in order to generate correct timings. The peripheral input clock frequency must be at least: • 2 MHz in Sm mode • 4 MHz in Fm mode As soon as a start condition is detected, the address is received from the SDA line and sent to the shift register. Then it is compared with the address of the interface (OAR1) and with OAR2 (if ENDUAL=1) or the General Call address (if ENGC = 1). Note: In 10-bit addressing mode, the comparison includes the header sequence (11110xx0), where xx denotes the two most significant bits of the address.

Header or address not matched: the interface ignores it and waits for another Start condition. Header matched (10-bit mode only): the interface generates an acknowledge pulse if the ACK bit is set and waits for the 8-bit slave address. Address matched: the interface generates in sequence: • An acknowledge pulse if the ACK bit is set • The ADDR bit is set by hardware and an interrupt is generated if the ITEVFEN bit is set. • If ENDUAL=1, the software has to read the DUALF bit to check which slave address has been acknowledged. In 10-bit mode, after receiving the address sequence the slave is always in Receiver mode. It will enter Transmitter mode on receiving a repeated Start condition followed by the header sequence with matching address bits and the least significant bit set (11110xx1). The TRA bit indicates whether the slave is in Receiver or Transmitter mode. Slave transmitter Following the address reception and after clearing ADDR, the slave sends bytes from the DR register to the SDA line via the internal shift

register. The slave stretches SCL low until ADDR is cleared and DR filled with the data to be sent. When the acknowledge pulse is received:

• The TxE bit is set by hardware with an interrupt if the ITEVFEN and the ITBUFEN bits are set. If TxE is set and some data were not written in the I2C_DR register before the end of the next data transmission, the BTF bit is set and the interface waits until BTF is cleared by a read to I2C_SR1 followed by a write to the I2C_DR register, stretching SCL low.

Slave receive:

Following the address reception and after clearing ADDR, the slave receives bytes from the SDA line into the DR register via the internal shift register. After each byte the interface generates in sequence:

An acknowledge pulse if the ACK bit is set The RxNE bit is set by hardware and an interrupt is generated if the ITEVFEN and ITBUFEN bit is set. If RxNE is set and the data in the DR register is not read before the end of the next data reception, the BTF bit is set and the interface waits until BTF is cleared by a read from the I2C_DR register, stretching SCL low.

Masters and Slaves

The devices on the I2C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. A slave cannot initiate a transfer over the I2C bus, only a master can do that. There can be, and usually are, multiple slaves on the I2C bus, however there is normally only one master. It is possible to have multiple masters,

I2C Transaction Handling Modes:

DMA:

DMA requests (when enabled) are generated only for data transfer. DMA requests are generated by Data Register becoming empty in transmission and Data Register becoming full in reception. The DMA must be initialized and enabled before the I2C data transfer. The DMAEN bit must be set in the I2C_CR2 register before the ADDR event. In master mode or in slave mode when clock stretching is enabled, the DMAEN bit can also be set during the ADDR event, before clearing the ADDR flag. The DMA request

must be served before the end of the current byte transfer. When the number of data transfers which has been programmed for the corresponding DMA stream is reached, the DMA controller sends an End of Transfer EOT signal to the I2C interface and generates a Transfer Complete interrupt if enabled:

• Master transmitter: In the interrupt routine after the EOT interrupt, disable DMA requests then wait for a BTF event before programming the Stop condition.

• Master receiver – When the number of bytes to be received is equal to or greater than two, the DMA controller sends a hardware signal, EOT_1, corresponding to the last but one data byte (number_of_bytes – 1).

If, in the I2C_CR2 register, the LAST bit is set, I2C automatically sends a NACK after the next byte following EOT_1. The user can generate a Stop condition in the DMA Transfer Complete interrupt routine if enabled. – When a single byte must be received: the NACK must be programmed during EV6 event, i.e. program ACK=0 when ADDR=1, before clearing ADDR flag. Then the user can program the STOP condition either after clearing ADDR flag, or in the DMA Transfer Complete interrupt routine.

Interrupt Mode:

I2C could work at the interrupt mode and at this mode the Micro Controller will execute the I2C functions when an event happenes.

Polling:

The default statues in the microcontroller.

The polling method means that the code is still waiting for an action to happened to continue the code implementation, for I2C protocol that means the microcontroller will still waiting for data to be received when it in the receiving mode, or will still transmitting the data if it is the transmitter, means it will still receive or send data all the time.

I2C master mode:

In Master mode, the I2C interface initiates a data transfer and generates the clock signal. A serial data transfer always begins with a Start condition and ends with a Stop condition. Master mode is selected as soon as the Start condition is generated on the bus with a START bit. The following is the required sequence in master mode. • Program the peripheral input clock in I2C_CR2 Register in order to generate correct timings •

Configure the clock control registers • Configure the rise time register • Program the I2C_CR1 register to enable the peripheral • Set the START bit in the I2C_CR1 register to generate a Start condition The peripheral input clock frequency must be at least: • 2 MHz in Sm mode • 4 MHz in Fm mode. Master is responsible for generating the clock of the communication.

Master transmitter

Following the address transmission and after clearing ADDR, the master sends bytes from the DR register to the SDA line via the internal shift register. The master waits until the first data byte is written into I2C_DR.

When the acknowledge pulse is received, the TxE bit is set by hardware and an interrupt is generated if the ITEVFEN and ITBUFEN bits are set. If TxE is set and a data byte was not written in the DR register before the end of the last data transmission, BTF is set and the interface waits until BTF is cleared by a write to I2C_DR, stretching SCL low.

Master receiver

Following the address transmission and after clearing ADDR, the I2C interface enters Master Receiver mode. In this mode the interface receives bytes from the SDA line into the DR register via the internal shift register. After each byte the interface generates in sequence: 1. An acknowledge pulse if the ACK bit is set 2. The RxNE bit is set and an interrupt is generated if the ITEVFEN and ITBUFEN bits are set

If the RxNE bit is set and the data in the DR register is not read before the end of the last data reception, the BTF bit is set by hardware and the interface waits until BTF is cleared by a read in the DR register, stretching SCL low.

Slave transmitter

Following the address reception and after clearing ADDR, the slave sends bytes from the DR register to the SDA line via the internal shift register. The slave stretches SCL low until ADDR is cleared and DR filled with the data to be sent. When the acknowledge pulse is received:

• The TxE bit is set by hardware with an interrupt if the ITEVFEN and the ITBUFEN bits are set. If TxE is set and some data were not written in the I2C_DR register before the end of the next data transmission, the BTF bit is set and the interface waits until BTF is

cleared by a read to I2C_SR1 followed by a write to the I2C_DR register, stretching SCL low.

Slave receiver

Following the address reception and after clearing ADDR, the slave receives bytes from the SDA line into the DR register via the internal shift register. After each byte the interface generates in sequence: • An acknowledge pulse if the ACK bit is set

• The RxNE bit is set by hardware and an interrupt is generated if the ITEVFEN and ITBUFEN bit is set. If RxNE is set and the data in the DR register is not read before the end of the next data reception, the BTF bit is set and the interface waits until BTF is cleared by a read from the I2C_DR register, stretching SCL low.

Block Diagram:

In Master mode, the I2C interface initiates a data transfer and generates the clock signal. A serial data transfer always begins with a start condition and ends with a stop condition. Both start and stop conditions are generated in master mode by software. In Slave mode, the interface is capable of recognizing its own addresses (7 or 10-bit), and the General Call address. The General Call address detection may be enabled or disabled by software. Data and addresses are transferred as 8-bit bytes, MSB first. The first byte(s) following the start condition contain the address (one in 7-bit mode, two in 10-bit mode). The address is always transmitted in Master mode.

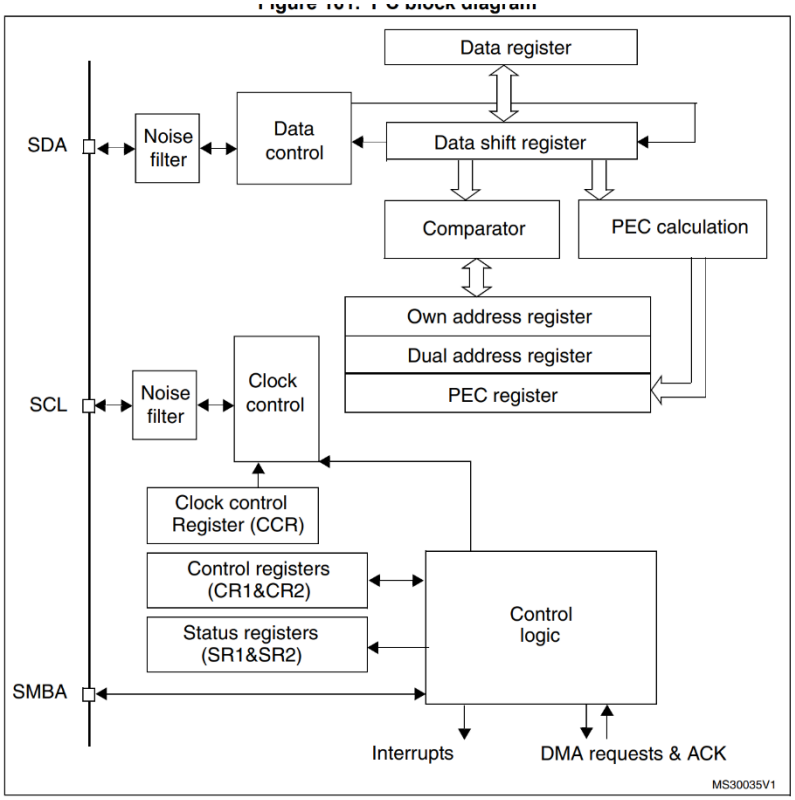

Figure 1: I2C Block Diagram.

Register Used in Driver:

Register	Functionality
RCC_APB1ENR BIT 21(I2C1EN)	Enable I2C clock
RCC_ APB1ENR BIT 1 (TIM3EN)	Enable GPIOB clock
GPIOB_MODER BIT 16, 17	Alternate function for pin PB8
GPIOB_MODER BIT 18, 19	Alternate function for pin PB9
GPIO_OSPEEDR Bit 16, 17	Very High Speed for pin PB8
GPIO_OSPEEDR Bit 18, 19	Very High Speed for pin PB9
GPIO_PUPDR bit 16, 17	Pull up for pin PB8, PB9
GPIOB_AFRH bit 0, 1, 2,3	AF4 for pin PB8
GPIOB_AFRH bit 4,5, 6, 7	AF4 for pin PB9
I2C1_CR1 bit 15 (SWRST)	I2C peripheral under reset state
I2C_CR2 bit 0, 1, 2, 3, 4, 5	Set frequency in MHZ
I2C_CCR bit 0 to 11	Configure the clock control register
I2C1_TRISE bit 0 to 5	Configure the Rise time register
I2C1_CR1 bit 10 (ACK)	Enable the I2C
I2C1_CR1 bit 8 (START)	Generate start condition
I2C1_SR1 bit 0 (SB)	start condition is generated
I2C1_DR bit 0 to 7 DR [7:0]	8-bit data register
I2C1_SR1 bit 1(ADDR)	Address sent (master mode)/matched (slave mode)
I2C1_OAR1 bit 9, 8 (ADD)	Take the interface address
I2C_CR1 bit 9 (STOP)	Stop generation
I2C1_SR2 bit 0 (MSL)	Master/slave
I2C_CR1 bit 10 (ASK)	Enable Acknowledge
I2C1_SR1 bit 1 (ADDR)	Address sent (master mode)/matched (slave mode)
I2C1_SR1 bit 2 (BTF)	Byte transfer finished
I2C1_SR1 bit 7 (TXE)	Data register empty (transmitters)
I2C_SR1 bit 6 (RXNE)	Data register not empty (receivers)
I2C1_CR1 bit 0 (PE)	Peripheral enable or disable
I2C1_SR1 bit 6 (RXNE)	Data register not empty (receivers)

Chapter 3

Functions Documentation:

```
/*
Description: begin function to set all settings of i2c.
Parameters: addr: the address of the connected Slave.
Return statues: this function is void doesn't return any data
*/
void begin(uint8_t addr){
        // Enable the I2C CLOCK and GPIO CLOCK
        RCC_APB1ENR |= (1<<21);  // enable I2C CLOCK
        RCC_AHB1ENR |= (1<<1);  // Enable GPIOB CLOCK

        // Configure the I2C PINs for ALternate Functions
        //PB8 and PB9 are connected to I2C1_SCL and I2C1_SDA
        GPIOB_MODER |= (2<<16) | (2<<18);  // Bits (17:16)= 1:0 --> Alternate Function for Pin PB8
                                           // Bits (19:18)= 1:0 --> Alternate Function for Pin PB9
        GPIOB_OTYPER |= (1<<8) | (1<<9);  //  Bit8=1, Bit9=1  output open drain
        GPIOB_OSPEEDR |= (3<<16) | (3<<18);  // Bits (17:16)= 1:1 --> Very High Speed for PIN PB8; Bits (19:18)= 1:1 --> Very High Speed for PIN PB9
        GPIOB_PUPDR |= (1<<16) | (1<<18);  // Bits (17:16)= 0:1 --> Pull up for PIN PB8; Bits (19:18)= 0:1 --> pull up for PIN PB9
        GPIOB_AFRH |= (4<<0) | (4<<4);  // Bits (3:2:1:0) = 0:1:0:0 --> AF4 for pin PB8;  Bits (7:6:5:4) = 0:1:0:0 --> AF4 for pin PB9


          // Reset the I2C
          I2C1_CR1 |= (1<<15); //I2C Peripheral under reset state
          I2C1_CR1 &= ~(1<<15); //I2C Peripheral not under reset

          // Program the peripheral input clock in I2C_CR2 Register in order to generate correct timings
          I2C1_CR2 |= (PCLK1Frequency <<0);  // PCLK1 FREQUENCY in MHz

          // Configure the clock control registers
          I2C1_CCR = 210<<0;

          // Configure the rise time register
          I2C1_TRISE = 43;  // I2C1_TRISE=((Tr(SCL))/(Tp(CLK1)))+1 = (1000 ns/(1/42*10^6))+1 =43

          // Program the I2C_CR1 register to enable the peripheral
          I2C1_CR1 |= (1<<0);  // Enable I2c
          /*Initialize the I2C module and join the I2C bus. If
            the address is -1 (define it as a preprocessor macro)
            it joins the bus as a master, otherwise join as a
            slave.*/
          if (addr == -1){
          // -1 = 0xFF
          //join I2C as a master mode
          /******Generate a start condition**********/
          I2C1_CR1 |= (1<<10);  // Enable the ACK
          I2C1_CR1 |= (1<<8);  // Generate START
          while (!(I2C1_SR1 & (1<<0)));  // Wait for SB bit to set to 1, This indicates that the start condition is generated

          /**********Send the Slave Address to the DR Register****************/
          I2C1_DR = addr;  //  send the address
          while (!(I2C1_SR1 & (1<<1)));  // while ADDR=0 address not recieved but when ADDR=1 the address recieved
          temp = I2C1_SR1 | I2C1_SR2;  // read SR1 and SR2 to clear the ADDR bit

        }
       else{
        //I2C default is a slave unless you generate a start condition to join as a master
          I2C1_OAR1 = addr;//own address interface
          I2C1_CR1 |= (1<<10);  // Enable the ACK ,indicate that a byte is received
        }
}
```

Figure 2: Begin Function.

```
/*
Description: this funtion Release the I2C bus.
Parameters: void no parameter needed.
Return statues: this function is void doesn't return any data
*/
void end(void)  {

          I2C1_CR1 |= (1<<9);//stop generation
           // Wait until the STOP condition is complete
          while (I2C1_SR2 & (1<<0)); //Cleared by hardware after detecting a Stop condition on the bus

          // Clear the STOP bit
          I2C1_CR1 &= ~(1<<9);
  }
```

Figure 3:End function.

```
/*
Description: this funtion Request bytes from device of addr.
Parameters: addr: the address of the required Slave.
            quantity: the data you want to send.
Return statues: this function is void doesn't return any data.
*/
 void  requestFrom(uint8_t addr,uint8_t quantity){
            I2C1_CR1 |= (1<<10);  // Enable the ACK
            I2C1_CR1 |= (1<<8);  // Generate start codition
            while (!(I2C1_SR1 & (1<<0)));// wait until the start condition is generated

            I2C1_DR = addr;  //  send the address
                while (!(I2C1_SR1 & (1<<1)));  // while ADDR=0 address not recieved but when ADDR=1 the address recieved
            temp = I2C1_SR1 | I2C1_SR2;   // read SR1 and SR2 to clear the ADDR bit
           availableBytes =  quantity;
  }
```

Figure 5:Requistfrom Function.

```
/*
Description: this funtion Begin enqueueing up the data to transmit to the device given by addr.
Parameters: addr: the address of the connected Slave.
Return statues: this function is void doesn't return any data
*/

void beginTransmision(uint8_t addr) {
    // Send START condition
         I2C1_CR1 |= (1<<10);  // Enable the ACK
         I2C1_CR1 |= (1<<8);  // Generate start codition
         while (!(I2C1_SR1 & (1<<0)));// wait until the start condition is generated

    // Send slave address
         I2C1_DR = addr ;  // send the address
        while (!(I2C1_SR1 & (1<<1)));  // while ADDR=0 address not recieved but when ADDR=1 the address recieved
        temp = I2C1_SR1 | I2C1_SR2; //once read SR1 and SR2 clear the ADDR bit
}
```

Figure 4:Begin Transmission Function.

```
/*
Description: this funtion Transmit the bytes that have been queued and end the transmission.
Parameters: void.
Return statues: this function is void doesn't return any data
*/
void endTransmision(){

         // wait for BTF bit to set
          while (!(I2C1_SR1 & (1<<2))); //waiting while BTF=0 but when BTF=1; Data byte transfer succeeded
          I2C1_CR1 |= (1<<9);// generate stop condition to end the transmsion

}
```

Figure 6:End Transmission Function.

```c
/*
Description: this funtion Modify the peripheral clock frequency (define the maximum and minimum limits as a preprocessor macro).
Parameters: freq: the required frequancy to data to be sent.
Return statues: this function is void doesn't return any data
*/
void setClock(uint8_t freq){
        if ( freq >=2 && freq<= 50){
           PCLK1Frequency = freq;
        }

  }
```

Figure 7: Set Clock Function.

```c
/*
Description: this funtion Reads a byte that was transmitted from a peripheral to the controller.
Parameters: addr: the address of required register we want to read data from.
Return statues: this function is void doesn't return any data
*/
uint8_t Read (uint8_t addr ){
    /**** STEPS FOLLOWED  ************
        If only 1 BYTE needs to be Read
        a) Write the slave Address, and wait for the ADDR bit (bit 1 in SR1) to be set
        b) the Acknowledge disable is made during EV6 (before ADDR flag is cleared) and the STOP condition generation is made after EV6
        c) Wait for the RXNE (Receive Buffer not Empty) bit to set
        d) Read the data from the DR
        */

          uint8_t receivedData = 0;
                I2C1_DR = addr;  //  send the address
             while (!(I2C1_SR1 & (1<<1)));  // wait for ADDR bit to set

          I2C1_CR1 &= ~(1<<10);  // clear the ACK bit
          temp = I2C1_SR1 | I2C1_SR2;  // read SR1 and SR2 to clear the ADDR bit.... EV6 condition
          I2C1_CR1 |= (1<<9);  // Stop I2C

          while (!(I2C1_SR1 & (1<<6)));  // wait for RxNE to set

          // Read the data from the DATA REGISTER
          receivedData = I2C1_DR;
       return receivedData;
    }
```

Figure 8: Read Function.

```c
/*
Description: this funtion Write data from peripheral to controller.
Parameters: dat: the data required to be written in the DR Register.
Return statues: this function is void doesn't return any data
*/
void Write (uint8_t dat){
  /**** STEPS FOLLOWED  ************
  //master transmitter
  1. Wait for the TXE (bit 7 in SR1) to set. This indicates that the DR is empty
  2. Send the DATA to the DR Register
  3. Wait for the BTF (bit 2 in SR1) to set. This indicates the end of LAST DATA transmission
  */
          while (!(I2C1_SR1 & (1<<7)));  // wait for TXE bit to set
          //while TXE=0, I2C1_DR is empty but when TXE=1 by the hardware, I2C1_DR is not empty

              I2C1_DR = dat   ;
              // wait for BTF bit to set
          while (!(I2C1_SR1 & (1<<2))); //waiting while BTF=0 but when BTF=1; Data byte transfer succeeded

  }
```

Figure 9: Write Function.

```c
/*
Description: this funtion Enable the Noise filter.
Parameters: en: if 1 the digital filter is enabled if 0 the analog filter is enabled.
            filter_type: the type of filter required.
Return statues: this function is void doesn't return any data
*/
void enableNoiseFilter (uint8_t en, uint8_t filt_type) {
 if (!(I2C1_CR1 & (1 << 0))){//check if PE is disabled or not
    if (en) {
        if (filt_type == 0x01) {
            // Enable digital filter
            //set bits [0:3] of I2C1_FLTR to 0001;

        } else {
            // Enable analog filter
            //I2C1_FLTR |= (0<<4);
        }
    }
    }
```

Figure 10:Enable Noise Filter Function.

```c
/*
Description: this funtion called when a peripheral receives a transmission through the interrupt service routine (use function pointer).
Parameters: void *function(void): pointer to function thtat will be excuted.
Return statues: void.
*/
void onReceive(void(*function)(void))
/*
1 - start condition detected
2 - ADDR == 1 & RxNE == 1
3 - execute function
4 - read DR
5 - stop generation
*/
{
 while(!(I2C1_SR1 & (1<<1)) & !(I2C1_SR1 & (1<<6))); // wait till ADDR == 1 & RxNE == 1
 handler = function ;         // handler pointer takes value of function
 handler();
 temp = I2C1_DR;     // read DR
 I2C1_CR1 |= (1<<9); // stop generation
}
```

Figure 11: On Receive Function.

```
/*
Description: this funtion check if there is a data recieved or not.
Parameters: void.
Return statues: return the number of bytes available for retrieval.
*/
uint16_t availableBytes(void) {
    uint16_t availableBytes = 0;

    // Check the Receive Data Register Not Empty (RxNE) flag
    if (I2C1_SR1 & (1<<6)) {
        availableBytes++;
    }

    // You might need additional checks or processing based on your implementation

    return availableBytes;
}
/*
```

Figure 12: Available Byte Function.

```
/*
Description: this funtion called when controller requests data through the interrupt service routine(use function pointer).
Parameters: void *function(void): pointer to function thtat will be excuted.
Return statues: void.
*/
void onRequest(void(*function)(void))
/*
1 - start condition detected
2 - ADDR == 1 & Tx == 1
3 - execute function
4 - read DR
5 - stop generation
*/
{
 while(!(I2C1_SR1 & (1<<1)) & !(I2C1_SR1 & (1<<7))); // wait till ADDR == 1 & Tx == 1
 handler = function ;         // handler pointer takes value of function
 handler();
 temp = I2C1_DR;     // read DR
 I2C1_CR1 |= (1<<9); // stop generation
}
```

Figure 13: On Request Function.