CONTROL AND COMPUTER APPLICATIONS - EEP311

INTER-INTEGRATED CIRCUIT (I2C) PROTOCOL REVIEW

December 24, 2023

Ahmed Samy Mohammed Elnozahy	20010099
Mustafa Mohammed Hassan AbdelMagid	20011946
Mohammed ElSayed Ahmed ElSayed	20011499
Ahmed Mohammed Nashaat Ali	20010209
Mohammed AbdElghafar AbdElqader	20011637
Ahmed Mohammed Anwar Ebrahem	20010173
Kirolos John Heshmat	20011125
Bassam Elsayed Ali	20010394
Marwan Ahmed Mohammed Ahmed	20011847

Contents

1 Introduction

1.1 Theory

The Inter-Integrated Circuit (I2C) protocol is a widely used serial communication protocol designed for communication between integrated circuits, modules, sensors, and various other devices. An I2C driver is a software component or module responsible for controlling the hardware I2C interface present in microcontrollers, microprocessors, or dedicated I2C controller chips. It facilitates communication between the master (initiating device) and slave (target device) components connected on the I2C bus.

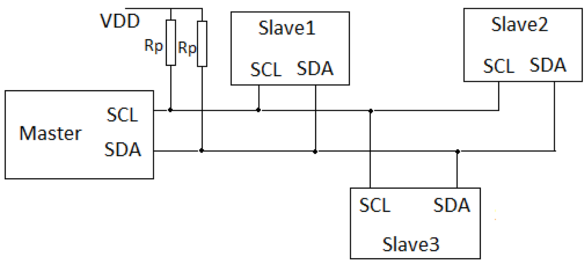

Figure 1: I2C Typical Bus.

Protocol Implementation: The I2C driver implements the I2C protocol specifications defined by the hardware standards. It handles the start, stop, and restart conditions on the bus, initiates data transfers, and manages acknowledgment signals.

Device Addressing: It manages device addressing, both for reading from and writing to specific slave devices on the bus. This includes sending the device's unique address and handling acknowledgment or non-acknowledgment from the addressed device.

Data Transfer: The driver handles the data transmission and reception between the master and slave devices. It encapsulates the data in a format compliant with the I2C protocol, including handling of data bytes, addressing, and acknowledgment checks.

Bus Arbitration: In scenarios where multiple master devices share the same I2C bus, the driver manages bus arbitration to ensure that only one master device has control over the bus at any given time.

Error Handling: It incorporates error detection and handling mechanisms, such as timeouts, bus errors, and acknowledgment failures, ensuring robust communication and error recovery.

Clock Speed and Timing: The driver sets and manages the clock speed for data transmission, adhering to the timing constraints specified by the I2C standard.

Buffer Management: It handles data buffers for transmitting and receiving data, ensuring proper data storage and handling during communication.

1.2 Features

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

Like UART communication, I2C only uses two wires to transmit data between devices.

1.3 Arbitration

Arbitration of the I2C (Inter-Integrated Circuit) protocol refers to the process of determining which master device gains control of the bus when multiple masters are present on the same I2C bus.

Multiple Masters: In some systems, especially complex ones or those with multiple controllers, multiple master devices might be connected to the same I2C bus.

Bus Ownership: Only one master device can control the bus at a time to initiate communication with slave devices. The arbitration process decides which master gains control over the bus.

Arbitration Mechanism: When two or more masters try to access the bus simultaneously (attempting to start communication or sending data), the I2C protocol employs an arbitration mechanism to resolve conflicts and determine which master has priority.

Priority Determination: Arbitration in I2C is typically achieved through a priority-based system where masters contend for control based on their address or message priority. The master device that sends a higher-priority message or has a lower address (configured or fixed) gains priority.

Collision Detection: During simultaneous attempts by masters to access the bus, collision detection mechanisms are employed to identify the conflicting access and resolve the contention.

Bus Release: The winning master (determined through arbitration) gains control over the bus and proceeds with the communication, while the other masters release the bus and wait for a new opportunity to access it.

Standard I2C Behavior: In the standard I2C arbitration scheme, the master devices are expected to monitor the bus while transmitting to detect any inconsistency between the transmitted and received bits. The master immediately stops transmitting if it detects another device transmitting a conflicting bit.

Timing Considerations: Timing constraints are crucial during arbitration to ensure fair and reliable resolution of conflicts without causing bus contention or data corruption.

1.4 Throughput

From a raw protocol perspective the I2C bus uses 8-bit data size with 9th bit acknowledge. So assuming an SCL clock generated at a uniform rate the raw transfer efficiency will be no better than 8/9 or about 88

1.5 Selection of Master and Slaves

Master Device Selection: Control and Initiation: The master device in the I2C bus controls the communication, initiates data transfers, generates clock signals, and addresses slave devices for data read or write operations.

Capability and Intelligence: The master device is typically a microcontroller, microprocessor, or dedicated I2C controller chip capable of managing the bus, handling data transactions, and executing control algorithms.

Availability of I2C Interface: The selected master device should have an integrated I2C interface or support I2C communication through dedicated hardware or software.

System Integration: The master device should be compatible with the overall system architecture and capable of handling the required functionalities and processing tasks.

Addressing Capability: Depending on the application and the number of slave devices, the master device should support the necessary addressing bits for communication with multiple slave devices.

Slave Device Selection: Functionality and Purpose: Slave devices in an I2C network are components such as sensors, memory chips, displays, or other peripherals that provide specific functionalities or data.

Compatibility: The selected slave devices should be compatible with the master device's communication protocol, addressing scheme, and supported data transfer rates.

Address Assignment: Each slave device connected to the bus should have a unique address to enable the master to access and communicate with the specific slave device.

Integration and Interoperability: Slave devices should seamlessly integrate into the I2C network, offering required functionality and maintaining compatibility with other devices in the system.

Hardware and Software Support: The slave devices should have adequate hardware and/or software support for I2C communication, including proper signal handling and protocol adherence.

Power Requirements: Consideration of power consumption and voltage compatibility between devices is essential to ensure proper functioning and prevent damage.

Selecting appropriate master and slave devices based on compatibility, functionality, addressing, and system integration is fundamental to establish a reliable and efficient I2C communication network tailored to the specific requirements of the application.

1.6 Termination

Pull-Up Resistors: Pull-up resistors are the most common form of termination used in I2C buses. These resistors are connected between the SDA (data line) and SCL (clock line) signals and the positive power supply voltage (Vcc).

Role of Pull-Up Resistors: The primary function of pull-up resistors in I2C buses is to ensure that the SDA and SCL lines are pulled to a logical high level when not actively driven low by the devices on the bus.

Preventing Signal Corruption: Proper termination with pull-up resistors helps prevent signal corruption, reduces the risk of signal reflections, and ensures that the lines return to a stable high state after data transmission.

Standard Values: Pull-up resistors in I2C applications typically have values in the range of $2k\Omega$ to $10k\Omega$, depending on the specific requirements of the bus, including capacitance, speed, and the number of devices connected.

Bus Capacitance Consideration: The choice of pull-up resistor value is influenced by the total bus capacitance, and it is essential to strike a balance to achieve reliable signal transitions without compromising speed.

I2C Standard Compliance: Termination with pull-up resistors is a standard practice recommended by the I2C protocol to ensure proper signal quality and compliance with communication standards.

1.7 Addressing Modes in I2C

- **7-Bit Addressing:** The I2C protocol uses 7-bit addressing for device identification. This allows up to 128 unique addresses for different devices connected to the I2C bus.

- **Device Identification:** Each device on the bus is assigned a 7-bit address, which is used by the master device to identify and communicate with specific slave devices.

- **128 Device Addresses:** The 7-bit addressing scheme provides 128 possible addresses (2^7) for devices, ranging from 0x00 to 0x7F (in hexadecimal notation).
- **Reserved Addresses:** Certain address values are reserved for special purposes. For example, the address 0x00 is reserved for a general call to all devices on the bus.
- **General Call Addressing:** The general call address (0x00) is a special case where a command or data is sent to all devices on the bus simultaneously.
- **10-Bit Addressing:** In addition to 7-bit addressing, the I2C protocol also supports 10-bit addressing for applications requiring a larger address space.
- **Extended Addressing:** 10-bit addressing allows for an extended range of 1024 addresses (2^{10}), providing flexibility for systems with a large number of devices.
- **Backward Compatibility:** Devices supporting 10-bit addressing can coexist with those using 7-bit addressing on the same I2C bus, ensuring backward compatibility.
- **Device Identification:** The additional address bits in 10-bit addressing provide more granular device identification, reducing the likelihood of address conflicts in larger networks.
- **Addressing Mode Selection:** The choice between 7-bit and 10-bit addressing depends on the specific requirements of the I2C application, considering the number of devices and the desired address space.

1.8 Significance of Noise Filters in I2C Communication

Noise in I2C Communication: Noise, interference, or voltage spikes on the I2C bus can potentially corrupt data transmissions, leading to communication errors and system instability.

Signal Integrity: Maintaining signal integrity is crucial for reliable I2C communication, especially in applications where accurate data transfer is essential.

Noise Filters: Noise filters are components or techniques used to filter out unwanted signals or disturbances on the I2C bus, ensuring that the transmitted data remains accurate and free from corruption.

Types of Noise Filters:

- **Capacitive Filters:** Capacitors can be used to filter out high-frequency noise by providing a path for AC signals to ground.
- **Inductive Filters:** Inductors or ferrite beads can attenuate high-frequency noise, preventing it from affecting the I2C signals.
- **RC Filters:** Resistors and capacitors configured in RC filter circuits can be effective in reducing noise on the bus.
- **Ferrite Beads:** Placing ferrite beads on the I2C lines can suppress high-frequency noise and prevent it from propagating along the bus.

Placement and Configuration: The placement and configuration of noise filters depend on the specific characteristics of the I2C bus, the nature of the noise, and the requirements of the connected devices.

Improved Reliability: Incorporating noise filters in an I2C communication system contributes to improved reliability, reduced data errors, and enhanced overall performance.

1.9 Message Frame

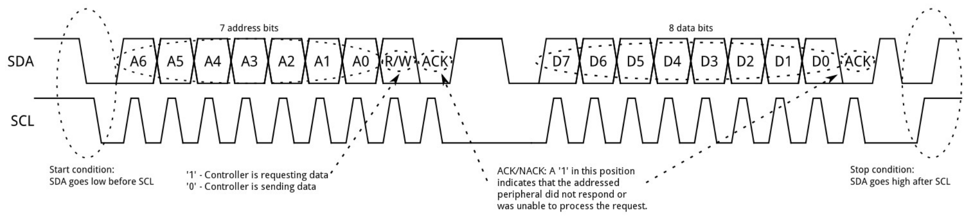
7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition:
SDA goes low before SCL
'1' - Controller is requesting data
'0' - Controller is sending data
ACK/NACK: A '1' in this position
indicates that the addressed
peripheral did not respond or
was unable to process the request.
Stop condition:
SDA goes high after SCL

Figure 2: I2C Basic Address and Data Frames

Message Frame in I2C: In the I2C protocol, data is transferred in messages, each consisting of a specific frame structure. The message frame defines the sequence of events and data transitions during communication between the master and slave devices.

Components of a Message Frame:

1. **Start Condition:** The master initiates communication by generating a start condition on the bus, indicating the beginning of a message.
2. **Address Frame:** The master sends the address of the target slave device, indicating the recipient of the upcoming data transfer.

3. **Read/Write Bit:** The master specifies whether the upcoming operation is a read or write by transmitting a read/write bit.

4. **ACK/NACK Bit:** After receiving data, the slave acknowledges successful reception with an acknowledgment (ACK) or indicates failure with a non-acknowledgment (NACK) bit.

5. **Data Frames:** Data frames contain the actual information being transmitted between the master and slave devices.

6. **Stop Condition:** The master concludes the communication by generating a stop condition on the bus.

Role of Address Frame: The address frame is crucial for identifying the specific slave device intended to receive or transmit data. Each slave device on the bus has a unique address assigned to it.

Read/Write Bit Significance: The read/write bit determines the direction of data flow, indicating whether the master is initiating a write operation to the slave or requesting data from the slave through a read operation.

ACK/NACK Bit Handling: The acknowledgment or non-acknowledgment bit following data transmission allows the recipient (slave) to confirm successful reception or report any errors.

1.10 Header and Payload in Packets

Packet Structure in I2C Communication: I2C communication involves the transmission of data in packets, each comprising a header and a payload. The header contains control information, while the payload carries the actual data.

Header Components:

- **Destination Address:** Specifies the address of the target device (slave) for which the packet is intended.

- **Source Address:** Identifies the address of the transmitting device (

2 Common ICs that Utilize the I2C Protocol

The I2C protocol is widely employed in a variety of integrated circuits (ICs) designed for different applications. Here are some common categories of ICs that utilize the I2C protocol:

2.1 Accelerometers:

- **MPU6050:** A popular motion-tracking device that combines a 3-axis gyroscope and a 3-axis accelerometer.

2.2 Gyroscopes:

- **L3GD20H:** A 3-axis digital gyroscope that provides precise angular rate sensing.
- **MPU6050:** As mentioned earlier, this device includes both accelerometer and gyroscope functions.

2.3 Real-Time Clocks (RTCs):

- **DS1307:** A real-time clock with a built-in I2C interface for precise timekeeping.
- **DS3231:** A highly accurate RTC with an integrated temperature-compensated crystal oscillator and I2C interface.

2.4 I/O Expanders:

- **MCP23017:** A 16-bit I/O expander with serial interface and interrupt capabilities.
- **PCF8574:** An 8-bit I/O expander with I2C interface, commonly used for GPIO expansion.

2.5 Motor Drivers:

- **TB6612FNG:** A dual motor driver with I2C interface for controlling DC motors.

3 I^2C Chapter Summary

3.1 I2C Modes:

- Masters control the bus, initiate and terminate communication, and determine the data flow direction.
- Slaves respond to commands from the master, acknowledge data reception, and provide requested data.

3.2 Communication Flow:

- **Initiation and Connection:** Master devices initiate communication by generating a start condition on the bus. It sends the address of the slave device it wants to communicate with. Slaves listen to the bus and respond if their address matches the one sent by the master.
- **Reception:** Master devices can receive data from slave devices by initiating a read operation after sending the slave's address. Slaves respond by sending the requested data to the master.
- **Transmission:** Master devices transmit data to slave devices by initiating a write operation after sending the slave's address. Slaves receive the data and acknowledge the successful reception.

3.3 I2C Transaction Handling Modes:

1. **Polling:**
    - **Pros:** Simplicity in implementation. Easier to manage and understand the flow of communication.
    - **Cons:** Consumes CPU cycles, making the microcontroller less available for other tasks. Inefficient for systems requiring fast response times or handling multiple simultaneous tasks.
2. **Interrupts:**
    - **Pros:** Efficient utilization of CPU resources by allowing the microcontroller to perform other tasks while waiting for I2C events. Suitable for systems with asynchronous or sporadic I2C activities.
    - **Cons:** Requires proper interrupt handling and may introduce complexity in coding and debugging.
3. **DMA (Direct Memory Access):**
    - **Pros:** Offloads CPU involvement, enabling the microcontroller to perform other tasks simultaneously. Efficient for handling large data transfers or continuous data streaming.

- **Cons:** Requires careful configuration and management of DMA controller settings. May introduce complexity in programming and debugging.

3.4 I2C Module Block Diagram

The I2C module operates in two main modes: Master mode and Slave mode

3.4.1 Master Mode Operation

In Master mode, the I2C interface takes the initiative in data transfer and assumes the responsibility of generating the clock signal. The sequence of a serial data transfer in Master mode always begins with a start condition and concludes with a stop condition. Notably, both start and stop conditions are generated by software.

3.4.2 Slave Mode Operation

In Slave mode, the I2C interface recognizes its own addresses (7-bit or 10-bit) and the General Call address. General Call detection is configurable through software.

Data and addresses are sent as 8-bit bytes, with the Most Significant Bit (MSB) first. The initial byte(s) after the start condition hold the address information, with the format varying based on the addressing mode (one byte in 7-bit mode, two bytes in 10-bit mode).

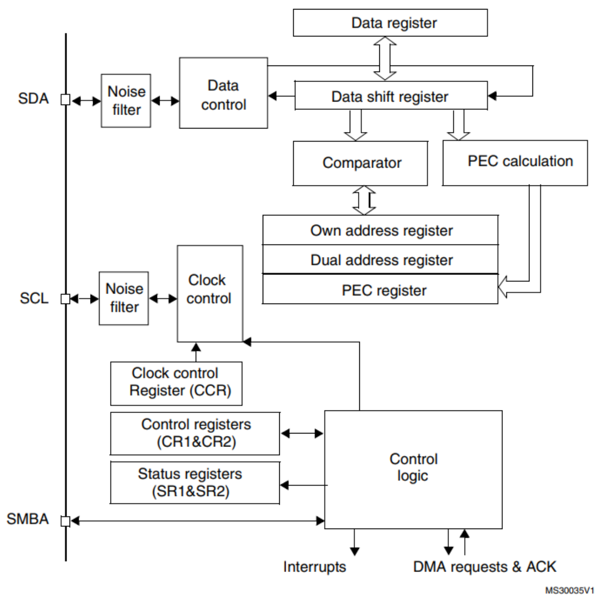

Figure 3: I2C Module Block Diagram

3.5 I2C Register Configuration

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	Set alternate function for pin PB8
GPIOB_MODER BIT 18, 19	Set alternate function for pin PB9
GPIO_OSPEEDR Bit 16, 17	Set Very High Speed for pin PB8
GPIO_OSPEEDR Bit 18, 19	Set Very High Speed for pin PB9
GPIO_PUPDR bit 16, 17	Set Pull-up for pin PB8, PB9
GPIOB_AFRH bit 0, 1, 2, 3	Set AF4 for pin PB8
GPIOB_AFRH bit 4, 5, 6, 7	Set AF4 for pin PB9
I2C1_CR1 bit 15 (SWRST)	Put 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	8-bit data register (DR [7:0])
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)

4 Driver Code

I2C Initialization

`void I2C1_init_begin()` Enables I2C1 clock and GPIOB for I2C pins. Configures pins as an alternate function for I2C. Sets pull-up resistors and open-drain mode. Initializes I2C1 for master or slave mode.

```
void I2C1_init_begin() {
 // Enable I2C1 clock and GPIOB for I2C pins
 RCC_EN_I2C1_PERIPHRAL();
 RCC_EN_I2C1_PORT();

 // Configure pins as an alternate function for I2C
 I2C1_GPIO_ALTERNATE();
 I2C1_GPIO_ALTERNATE_FUNCTION();

 // Set pull-up resistors and open-drain mode
 GPIOB_PUPDR |= ( (1 << 12) | (1 << 14) );
 GPIOB_OTYPER |= ( (1 << 6) | (1 << 7) );

 // Disable I2C Peripheral (Bit 0 PE)
 SET_BIT(I2C1_CR1, 0);

 // Addressing 7 Bit mode (BIT 15 ADDMODE)
 CLR_BIT(I2C1_OAR1, 15);

 // Master Configuration
 if (ADDRESS_CONFIG == -1) {
  I2C1_Master_Init(OWN_ADDRESS);
 }
 // Slave Configuration
 else {
  I2C1_Slave_Init((uint8_t)OWN_ADDRESS);
 }
}
```

I2C1 Master Initialization

`void I2C1_Master_Init(uint8_t Address)`

Description: Configures I2C1 as a master. Sets clock frequency to 400 kHz in Fast Mode. Generates a start condition. Sends the slave address to the data register. Enables the I2C peripheral.

Parameters:

- `uint8_t Address` - The slave address.

Return value: It will return the slave address.

Source:

```c
uint32_t ccr_value;
uint8_t read_sr = 0;
/* Set Peripheral Clock frequency (APB1 = 45 MHz) in CR2 register */
I2C1_CR2 |= (1 << 0) | (1 << 2) | (1 << 3) | (1 << 5);
/* Fast Mode Select <Bit 15 F/S> */
SET_BIT(I2C1_CCR, 15);
/* 1: Fm mode Tlow/Thigh = 16/9 (see CCR) <Bit 14 DUTY> */
SET_BIT(I2C1_CCR, 14);
ccr_value = (uint16_t)5u;
I2C1_CCR |= ccr_value;
/* Set the Rise Time register */
I2C1_TRISE |= (uint8_t)14u;
/* Enable the ACK */
SET_BIT(I2C1_CR1, 10);
/* Generate START */
SET_BIT(I2C1_CR1, 8);
/* Polling until SB bit to be set */
while (!(I2C1_SR1 & (1 << 0)));
/* Send the Slave Address to the DR Register */
I2C1_DR = Address;
while (!(I2C1_SR1 & (1 << 1)));
/* Read SR1 and SR2 to clear the ADDR bit */
read_sr = I2C1_SR1 | I2C1_SR2;
/* Enable I2C Peripheral <Bit 0 PE> */
SET_BIT(I2C1_CR1, 0);
```

I2C1 Slave Initialization

`void I2C1_Slave_Init(uint8_t ownAddress)`

Description: Configures I2C1 as a slave. Sets the slave address. Enables the I2C peripheral and acknowledgment.

Parameters:

- `uint8_t ownAddress` - The slave address.

Return value: The slave address.

Source:

```
/* Set the slave address in OAR1 register */
I2C1_OAR1 = (ownAddress << 1) | (1 << 14); // 7-bit addressing

/* Set Peripheral Clock frequency (APB1 = 45 MHz) in CR2 register */
I2C1_CR2 |= (1 << 0) | (1 << 2) | (1 << 3) | (1 << 5);
/* Enable I2C Peripheral <Bit 0 PE> */
SET_BIT(I2C1_CR1, 0);
/* Enable Acknowledgment <Bit 10 ACK> */
SET_BIT(I2C1_CR1, 10);
```

I2C1 Write

`void I2C1_Write(uint8_t Data)`

Description: Writes a byte of data to the I2C bus. Waits for the transmit buffer to be empty. Writes data to the data register. Waits for byte transmission to complete.

Parameters:

- `uint8_t Data` - The data byte to be written.

Return value: No explicit return value. This function writes data to the I2C bus.

Source:

```
void I2C1_Write(uint8_t Data)
{
    /* Polling for TXE bit to set */
    while (!(I2C1_SR1 & (1 << 7)));
    CLR_BIT(I2C1_CR1, 8);
    /* Write Data to be sent in Data Register */
    I2C1_DR = (uint8_t)Data;
    /* Polling while BTF=0 but when BTF=1; Data byte transfer succeeded
    while (!(I2C1_SR1 & (1 << 2)));
}
```

I2C1 Read

`uint8_t I2C1_Read()`

Description: Reads a byte of data from the I2C bus. Clears the ACK bit to signal the last byte. Generates a stop condition. Waits for data to be received. Reads data from the data register.

Return value: The data byte received.

Source:

```
uint8_t I2C1_Read()
{
    uint8_t read_sr = 0, DataReceived = 0;

    /* Clear the ACK bit */
    CLR_BIT(I2C1_CR1, 10);

    /* Read SR1 and SR2 to clear the ADDR bit */
    read_sr = I2C1_SR1 | (I2C1_SR2 << 8);

    /* Stop I2C */
    SET_BIT(I2C1_CR1, 9);

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

    /* Read the data from the DATA REGISTER */
    DataReceived = I2C1_DR;

    return DataReceived;
}
```

Request From Slave

`void requestFrom(uint8_t address, uint8_t quantity, uint8_t array[], uint8_t size)`

Description: Requests and reads multiple bytes from a slave device. Generates a start condition and sends the slave address with the read bit. Reads data bytes into the provided array. Generates a stop condition after reading the last byte.

Parameters:

- `uint8_t address` - The slave address.
- `uint8_t quantity` - The quantity of bytes to read.
- `uint8_t array[]` - The array to store the received data.
- `uint8_t size` - The size of the data array.

Return value: It will return the address, quantity, and size.

Source:

```
void requestFrom(uint8_t address, uint8_t quantity, uint8_t array[], ui
    uint32_t temp;
    uint8_t i = 0;

    /* Generate START condition */
    SET_BIT(I2C1_CR1, 8); // Start bit
    /* Wait for the START bit to be set */
    while (!(I2C1_SR1 & (1 << 0)));
    /* Send slave address with read bit (1) */
    I2C1_DR = (address << 1) | 1; // Address with read bit
    while (!(I2C1_SR1 & (1 << 1))); // Wait for ADDR bit
    /* Clear ADDR by reading SR1 */
    temp = I2C1_SR1;
    /* Disable ACK after the first byte to be received */
    if (quantity == 1) {
        CLR_BIT(I2C1_CR1, 10); // Disable ACK
    }
    /* Receive data bytes */
    for (i = 0; i < size - 1; i++) {
        while (!(I2C1_SR1 & (1 << 2))); // Wait for BTF
        CLR_BIT(I2C1_CR1, 10); // Disable ACK
        array[i] = I2C1_DR; // Read data byte
    }
    /* Wait for the last byte */
    while (!(I2C1_SR1 & (1 << 2))); // Wait for BTF
    SET_BIT(I2C1_CR1, 9); // Generate STOP condition
    array[size - 1] = I2C1_DR; // Read the last byte
}
```

Begin Transmission

`void beginTransmission(int address)`

Description: Initiates a transmission to a slave device. Generates a start condition and sends the slave address.

Parameters:

- `int address` - The slave address.

Return value: Address will transmission to a slave.

Source:

```
void beginTransmission(int address) {
    uint32_t temp;

    /* Start condition */
    I2C1_CR1 |= (1 << 8); // Start condition
    /* Wait till SB == 1 */
    while (!(I2C1_SR1 & (1 << 0)));
    /* Read SR1 */
    temp = I2C1_SR1;
    /* Send slave address */
    I2C1_DR |= (address << 0);
    /* Wait till ADDR == 1 */
    while (!(I2C1_SR1 & (1 << 1)));
    /* Read SR1 & SR2 to clear ADDR */
    temp = I2C1_SR1;
    temp = I2C1_SR2;
}
```

End Transmission

`void I2C1_endTransmission(void)`

Description: Generates a stop condition to end a transmission.

Parameters: void

Return value: Non-return.

Source:

```
void I2C1_endTransmission(void) {
    /* Stop generation */
    I2C1_CR1 |= (1 << 9);

    /* Polling until the STOP condition to be sent as
     * It is cleared by hardware after detecting a Stop condition on th
    while (I2C1_SR2 & (1 << 0));

    /* Clear the STOP bit */
    CLR_BIT(I2C1_CR1, 9);
}
```

End I2C1

`void I2C1_end(void)`

Description: It's important to call this function when I2C1 is not in use to reduce power consumption and avoid potential conflicts with other peripherals that share the same clock line. To re-enable I2C1, you would need to set the I2C1 clock enable bit (bit 21) in the APB1ENR register again.

Parameters: Void

Return value: Void

Source:

```
void I2C1_end(void) {
    /* Disable I2C Peripheral clk */
    CLR_BIT(RCC_APB1ENR, 21);
}
```

On Receive

`void onReceive(void (*function)(void))`

Description: Sets a handler function to be called when data is received. Waits for start condition and data to be available. Calls the handler function. Reads the received data. Generates a stop condition.

Parameters: `void (*function)(void)`

Return value: The function doesn't return any value.

Source:

```
void onReceive(void (*function)(void)) {
    uint8_t receivedData;

    /* Set the handler pointer to the given function and execute it */
    void (*handler)(void) = function;

    /* Wait until ADDR == 1 and RxNE == 1 (Start condition detected) */
    while (!(I2C1_SR1 & (1 << 1)) & !(I2C1_SR1 & (1 << 6)));

    handler();
    /* Read data from the Data Register (DR) */
    receivedData = (uint8_t)I2C1_DR;

    /* Generate STOP condition */
    I2C1_CR1 |= (1 << 9);
}
```

On Request

`void onRequest(void (*function)(void))`

Description: Sets a handler function to be called when the master requests data from the slave. Waits for address match and transmitter mode. Calls the handler function. Reads the data register (unclear purpose). Generates a stop condition.

Parameters: `void (*function)(void)`

Return value: The function doesn't return any value.

Source:

```
void onRequest(void (*function)(void)) {
    uint16_t temp;

    /* Set the handler pointer to the given function and execute it */
    void (*handler)(void) = function;

    /* Wait for ADDR bit (Address matched) and Tx bit (Transmitter mode
    while (!(I2C1_SR1 & (1 << 1)) && !(I2C1_SR1 & (1 << 7)));

    /* Set the function pointer to the provided function */
    handler = function;
    /* Execute the provided function */
    handler();
    /* Read DR register */
    temp = I2C1_DR;
    /* Generate STOP condition */
    I2C1_CR1 |= (1 << 9);
}
```

Available

`uint8_t available(void)`

Description: Checks if there's any data available to be read from the I2C1 receive buffer. This is useful for determining when to initiate a read operation and avoid attempting to read from an empty buffer.

Parameters: void

Return value: If the RxNE bit is 1, indicating data is available, the function returns 1 (true). If the RxNE bit is 0, indicating no data available, the function returns 0 (false).

Source:

```
uint8_t available(void) {
    /* Check if RxNE (Receive buffer not empty) bit is set */
    if (I2C1_SR1 & (1 << 6)) {
        return 1; // Data is available
    } else {
        return 0; // No data available
    }
}
```

Set Clock

`void setClock(uint8_t freq)`

Description: Configures the clock frequency used for I2C1 communication. This adjusts the speed at which data is transferred on the I2C bus, allowing you to match the requirements of different devices on the bus.

Parameters: `uint8_t freq`

Return value: The `setClock(uint8_t freq)` function does not explicitly return a value. Its return type is void, indicating that it directly performs its actions without providing a return value.

Source:

```
void setClock(uint8_t freq) {
    /* Set the frequency in CR2 register */
    I2C1_CR2 |= (freq << 0);
}
```

Enable Noise Filter

`void I2C1_enableNoiseFilter(uint8_t DNF)`

Description: Enables and configures noise filtering on the I2C1 bus to improve signal integrity and reduce the impact of electrical interference.

Parameters: `uint8_t DNF`

Return value: The `I2C1_enableNoiseFilter(uint8_t DNF)` function does not have a return value. Its return type is void, meaning it performs its actions without providing any explicit output value.

Source:

```
void I2C1_enableNoiseFilter(uint8_t DNF) {
    /* Set 4 ANOFF: Analog noise filter OFF
     * 0: Analog noise filter enable */
    CLR_BIT(I2C1_FLTR, 4);

    /* Bits 3:0 DNF[3:0]: Digital noise filter */
    I2C1_FLTR |= DNF;
}
```