International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

Serial Communication Inter-IC Bus, Interface and Protocol Using BeagleBone

Ankur Yadav[a]*, Khushboo Kumari Yadav[b]

[a]*Individual Researcher, 680 Epic Way, San Jose, CA 95134, USA*
[b]*Intel Corporation, 3200 Mission College Blvd, Santa Clara, CA 95054, USA*
[a]*Email:ankurayadav@gmail.com*
[b]*Email:yadavkhushboo1994@gmail.com*

Abstract

To communicate with peripherals the embedded systems mainly use the serial communication. Hence serial communication plays a vital role in designing embedded systems. The protocols used for serial communication are Universal Asynchronous Receiver Transmitter (UART), Serial Peripheral Interface (SPI), Universal Serial Bus (USB), Control Access Network (CAN) and Inter IC Protocol (I2C). The characteristics of serial communication protocol are high speed and low data loss, ensure the data transfer and simplifying the system level design. This paper provides an overview of I2C-Bus, I2C protocol and its interfacing. The paper also demonstrates the I2C protocol using BeagleBone.

Keywords: Inter-IC (I2c); I2C-Bus; SCL; SDA; BeagleBone; ADXL345 3-axix accelerometer; i2cdetect tool.

1. Introduction

I2C stands for Inter-Integrated Circuit. The Inter Integrated Circuits (I2C) is a 2-wired communication bus. It was invented by Phillips Semiconductors (now NXP Semiconductors) to allow communication between integrated circuits (ICs) from different manufacturers and now it is used by almost all major manufacturers. Some of the applications that use the I2C bus are memory devices, PCs, Television, cellphones, microcontrollers, LCD, ADCs, DACs and other devices. It's I²C (Inter-Integrated Circuit, referred to as I-two-C, IIC or I-squared-C) [1]. The I2C protocol allows connection of a wide range of devices without using a separate addressing or chip enable signals Reference [2]. It uses two-wire interface for efficient Inter-IC communication.

Received: 10/9/2025
Accepted: 12/9/2025
Published: 12/19/2025

---

* *Corresponding author.*

One as the serial clock (SCL) line and other as the serial data (SDA) line. Each device linked to the bus has a unique address used to recognize the device in communication and is software addressable by a distinct address and a simple master-slave relationship always exist; masters can operate as master-transmitters or as master-receivers [2].

The protocol comprises a set of conditions to start or end communication between the devices. The I2C is 8-bit oriented in which bidirectional data can be transferred in various modes. The Standard-mode supports data transfers at up to 100 kbit/s, the Fast-mode supports 400 kbit/s, 1 Mbit/s is supported in the Fast-mode Plus (Fm+), and up to 3.4 Mbit/s in the High-speed mode and the Ultra-Fast-mode which is unidirectional mode with data transfer up to 5 Mbit/s [2].

2. Methodology

2.1.1 I2C Bus Specification

The embedded system consists of microcontrollers and peripherals like LCD drivers, sensors, memory, converters, I/O expanders, matrix switches etc [3]. The cost and complexity to connect these devices must be kept minimum and it must be taken care of while designing the system that the slower device should not slow down the faster one to communicate with the system. To satisfy these needs the I2C serial bus came into the picture. This bus specifies rules for the connections, protocol, formats, addresses and procedures for data transfer [3].

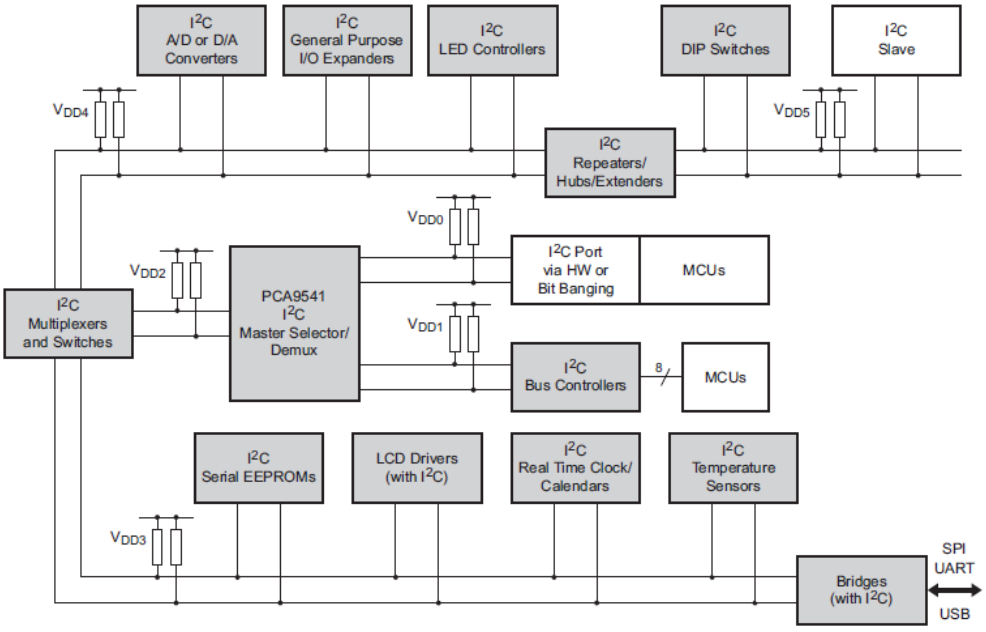

Figure 1: An example of I2C-bus applications [2]

The I2C has two bidirectional buses, those are SDA (serial data) and SCL (serial Clock), through which all the master and slave devices are connected. Each device can transmit, receive or it can be a transceiver. Master devices generate the bus clocks and start communication. Other devise which are the slaves respond to the commands of

the master devices on the bus. Each slave device has a unique address on the bus. Since no slave device sends commands to the masters hence the master devices need not be addressed.

2.1.2 I2C Terminology

Table 1: I2C-bus terminologies [2]

Transmitter	The transmitter sends the data to the bus.
Receiver	The receiver receives the data from the bus.
Master	The master generates the clock signal, initiates and terminates the data transfer.
Slave	Slaves are the devices that are addressed by the master devices.
Multi-master	At the same time, when more than one master device tries to control the bus without corrupting the message.
Arbitration	When multiple masters attempt to control the bus simultaneously, this procedure is done to allow only one master to take control without corrupting the winning message.
Synchronization	Process to match the clock signals of two or more devices

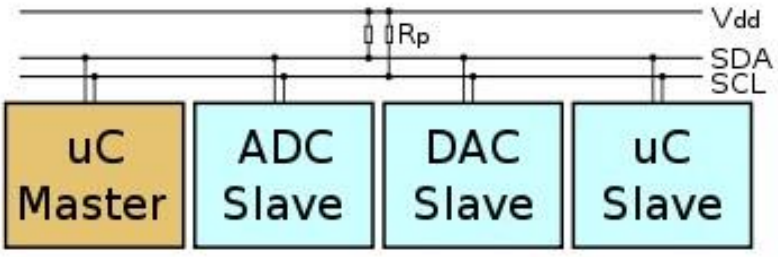

Figure 2: I2C Master-Slave configuration [3]

2.1.3 Bus Signal

The bidirectional SCL and SDA signals are connected to a positive power supply via resistors which means when both the lines are high the bus is free. The devices on the bus must have open-drain pins or open-collector pins to implement the wired-AND function. There is no standard bus voltage, hence the logical 1 depends upon supply voltage. Unlimited number of devices can be connected on a single bus provided the capacitance does not exceed 400pf.

2.1.4 Data Validity

One bit of data is transmitted with every clock pulse. When the SCL signal is low the SDA signal can change else when SCL is high SDA should be stable.

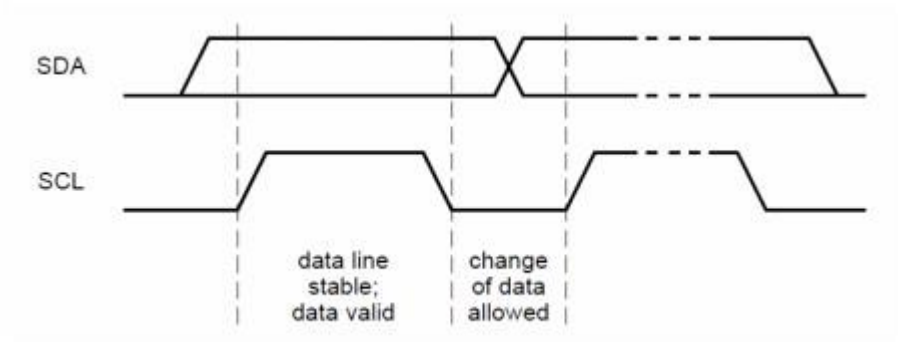

Figure 3: Bit transfer on I2C-bus [2]

2.1.5 Start and Stop Condition

The data transfer is initiated by sending a START(S) bit and is terminated by sending a STOP (P) bit. When during HIGH SCL the SDA line does a transition from HIGH to LOW it defines the START condition while the LOW to HIGH transition defines a STOP condition. Both this START and STOP conditions are initiated and controlled by the master device. After the START condition the BUS is BUSY and after the STOP condition it is again considered to be free after a certain amount of time. Instead of Stop condition if the repeated STOP (Sr) is sent then the BUS remains BUSY.

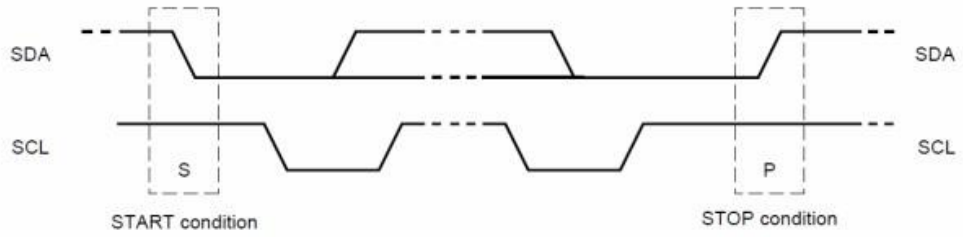

Figure 4: START and STOP conditions [2]

2.1.6 Byte Format

Unlimited number of bytes can be transmitted on the SDA line per transfer with each byte having 8-bit length, followed by an acknowledgement bit. The first bit transferred is the Most Significant Bit (MSB). In the interrupt conditions, to force the master into a wait state the slave can hold the clock line to SCL LOW. It releases the line SCL when it gets ready for another byte of data.

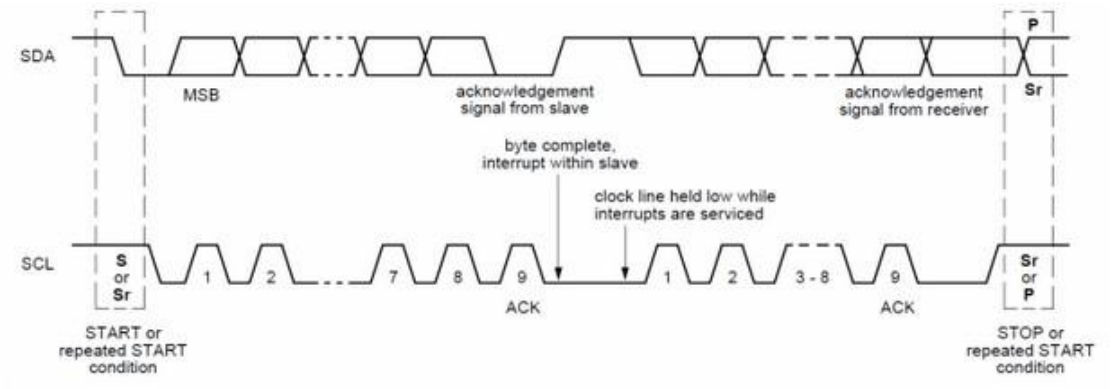

Figure 5: Data transfer on the I2C-bus [2]

The acknowledgement (ACK) or Not-Acknowledgement (NACK) is sent after every byte transferred depending on whether the data has been received successfully or not.

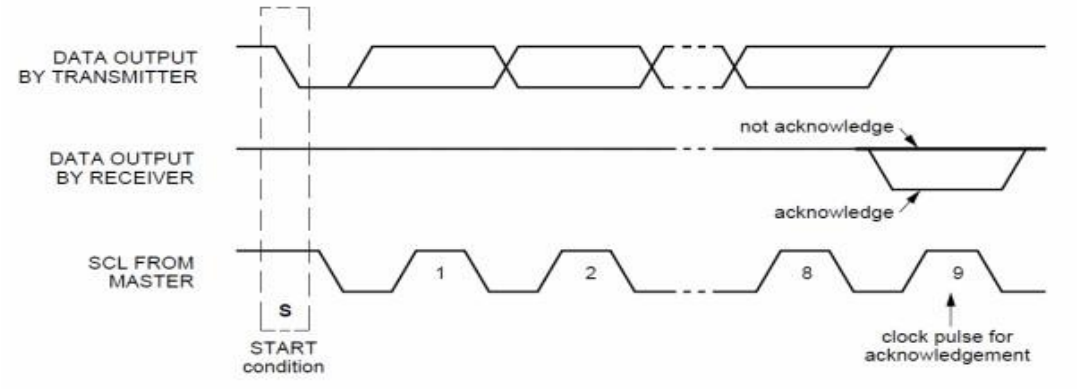

Figure 6: Acknowledgment and Not-Acknowledgment of data on the I2C Bus [3]

2.1.7 Clock Synchronization and Arbitration

In case of two masters transmitting on the bust at the same time, clock synchronization and arbitration is done to decide which master takes control of the bus. The wired-AND connection of I2c interface of the SCL line is used to perform clock synchronization. When the SCL transits HIGH to LOW, masters concerned start counting their LOW periods. The master clocks hold SCL line in the LOW state until the clock (CLK) reaches HIGH. If more than one clock is LOW, then the state of SCL line may not be changed by the LOW to HIGH transition of the clock. Hence the master holds the SCL line to LOW with the longest LOW period and during this time masters with a shorter LOW period enters a HIGH wait-state.

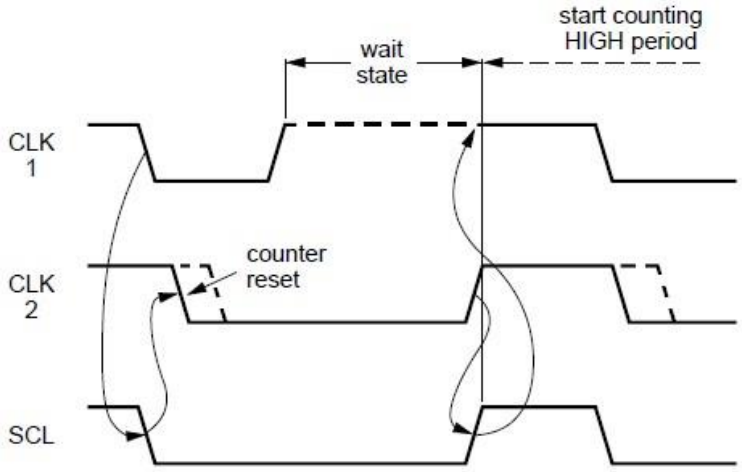

Figure 7: Synchronization of Clock during the arbitration procedure [2]

The clock goes HIGH when all the masters come into HIGH state. The SCL line is pulled LOW again by the first master which completes its HIGH period.

In case of two masters transmitting at the same time, the arbitration process is used to determine which master can transmit on the bus. When two or more masters transmit simultaneously, when the SCL is HIGH, each master checks if SDA matches the data, it has transmitted or not. The arbitration is performed bit by bit. The data can be transmitted by two masters simultaneously without any error, provided the transactions are identical. The moment the sync between the data of any master and the SDA goes off, the arbitration is lost, and the master turns off the SDA output drivers. But the other masters are not interrupted, and they complete their transactions. During the arbitration process the information is not lost.

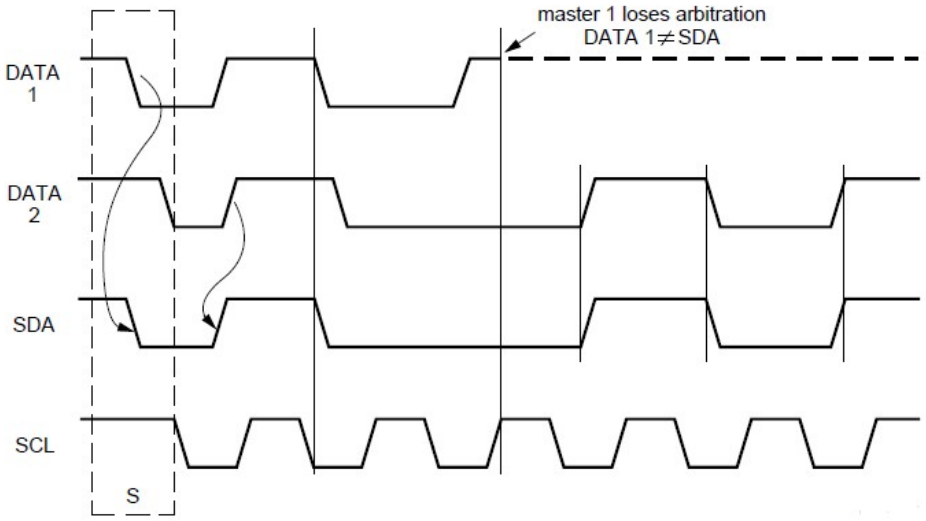

Figure 8: Arbitration procedure of two masters [2]

3. Results and Discussion

3.1 A Complete Data Transfer Process to 7-bit I2C Addresses

The start condition (S) is initiated by the master. After that the 7-bit address of the slave is sent. The 8th bit is sent after that which indicates the data transfer direction that is READ/WRITE (R/W) bit in which 'ONE' is sent for data READ or request for data and a 'ZERO' is sent for data WRITE or transmission of data. To terminate the communication a STOP bit is sent by the master device. However, the master sends a repeated START condition (Sr) if it wants to carry on the communication.

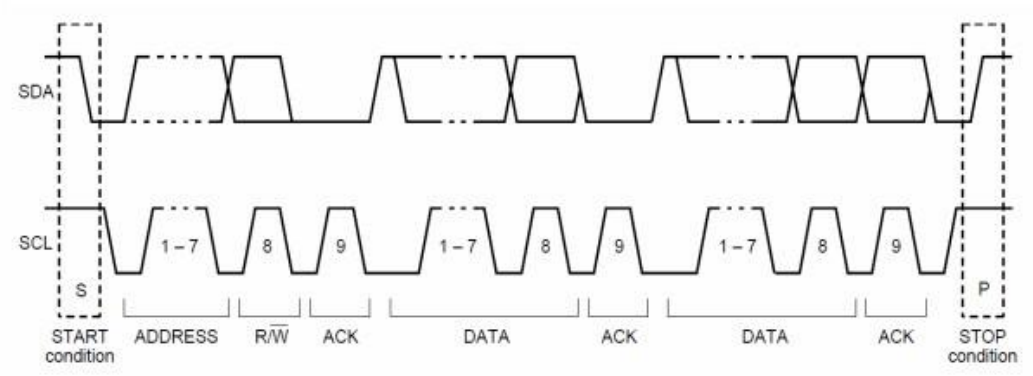

Figure 9: A complete data transfer [2]

3.1.1 Possible formats for communication between master and slave devices

- When the master sends data to the slave or writes, the slave sends the acknowledgement (A).

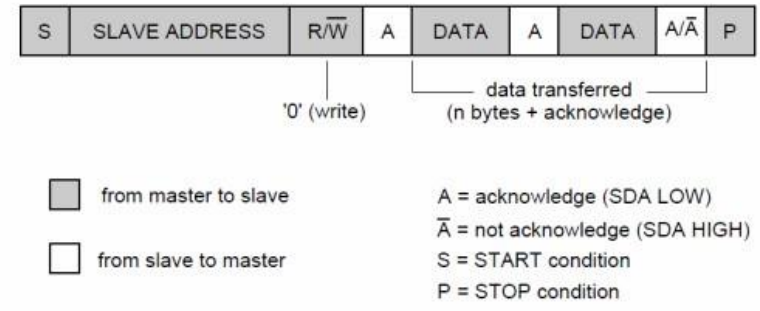

Figure 10: A slave addressed with a 7-bit address by a master [2]

- When the data transfer direction is inverted, i.e. the master reads, and the slave transmits, then the master sends the acknowledgement (A) to the slave device to inform the slave that it has received the data. The master transmits a Not-Acknowledge bit and after that a STOP bit to terminate the data transfer.

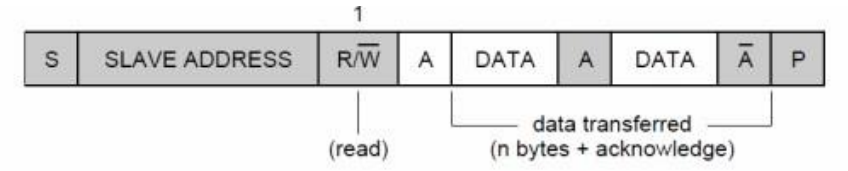

Figure 11: A master reads a slave (inverted data transfer direction) [2]

- Sometimes it may happen that the master wants to write as well as read data from the slave devices. In this condition the master first writes the data to the slave device and then changes the data transfer direction to read the data from the slave device. The above two formats are combined in this condition.

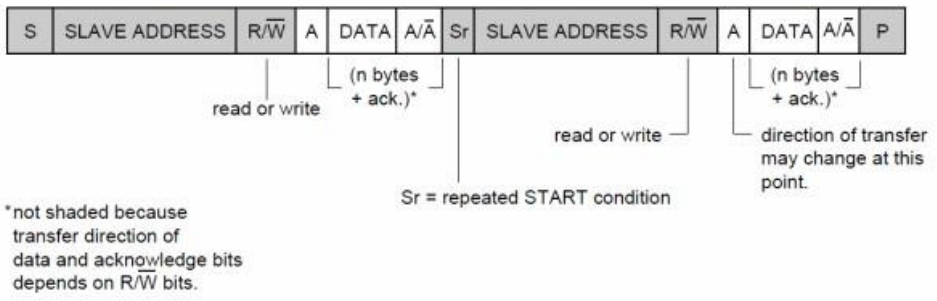

Figure 12: Combined format [2]

3.2 Demonstration of I2C Using Beaglebone Black and ADXL345 3Aaxix Accelerometer

3.2.1 BeagleBone Black

The BeagleBone Black is a credit card sized open-source computer which can run embedded Linux. It runs on the ARM Cortex A8 processor and has 512MB of ram and 4GB eMMC on board for flash storage. It has a large user community. It has been used in a wide variety of applications such as Internet of Things, 3d printing, robotics and advanced vehicle system etc. We will use this for our demonstration. It supports three i2c buses with frequencies up to 400kHz. To use the I2C bus in BeagleBone black we have to install a device tree overlay because its pins are multiplexed and we need to inform the OS about the pin configuration we want to use.

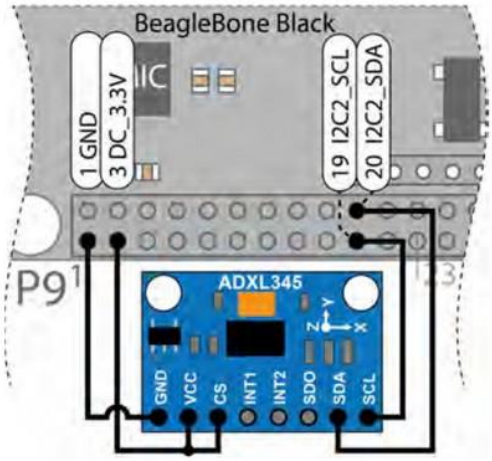

Figure 13: Picture taken from book: EXPLORING BEAGLEBONE: by Derek Molloy[5]

3.2.2 I2C Detect Tool

It is a handy tool when we want to access a sensor using Linux. We can easily see sensors connected to BeagleBone using this tool. We can also read and modify values in the sensor. So, using this tool we can easily configure and read the sensor before programming the sensor. After we successfully configure and read the sensor using i2cdetect then we can make program for it using C [5].

3.2.3 Preparing the Setup and Running the Code

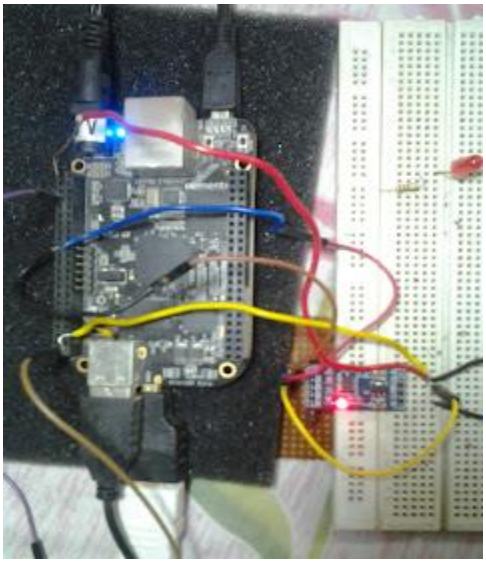

Figure 14: Connecting BeagleBone Black with ADXL345

- **Configuration for Accessing ADXL345 Using I2C Bus**

1)Connect ADXL345 to I2C2 of BeagleBone black as shown in the figure.

2)Login into BeagleBone shell.

3)Install proper device tree overlay for accessing I2C2.

- **Brief description about I2C function used in this code**

- **i2cOpenAdaptor(uint8_t adaptorNumber):** This function I2C adapter number as input and returns a file descriptor for accessing that I2C adapter.

- **i2cSetSlave(int i2cFD, uint8_t address):** This takes I2C adapter file descriptor and address at which device is attached as input and sets slave for file descriptor using ioctl call.

- **i2cSetAddress(int i2cFD, unsigned char add):** This is used by some other functions internally for random access of sensor registers. It takes file descriptor and random address to which file descriptor should point.

- **i2cWriteByte(int i2cFD, unsigned char add, unsigned char byte):** This function is used to write byte provided in the argument at the address provided.

- **i2cWriteBytes(int i2cFD, unsigned char add, int length, uint8_t *bytes):** This function is used to write bytes provided in argument from the address provided to address+length continuously.

- **i2cReadByte(int i2cFD, unsigned char add):** It reads a byte from the address provided and returns the byte.

- **i2cReadBytes(int i2cFD, unsigned char add, int length, uint8_t *buff):** It is used to get 'length' number of bytes in 'buff' array from 'add' starting address.

- **i2cClose(int i2cFD):** For closing file descriptor.

- **The detailed program and its output is provided in Appendix**

3.2.3 Making Program for Reading ADXL345 Values

- Open "/dev/i2c-1" file in read write mode. This file descriptor will be used to access the adaptor.
- Use file descriptor of adaptor from previous step and set our sensor address as slave to the adaptor.
- Now we have to enable measurement mode by setting bit 3 in POWER_CTL register.
- set value 0x08 at address 0x2d.
- Then we have to set proper data format for measurement using DATA_FORMAT register.
- set value 0x0b at address 0x31.
- Now we can read values x, y and z acceleration values from the sensor.

- for x acceleration read data from 0x32 and 0x33.
- for y acceleration read data from 0x34 and 0x35
- for z acceleration read data from 0x36 and 0x37.

- Screenshot of output of the code after reading ADXL345 sensor values when button is clicked.

3.2.3 Output from ADXL345 I2C Read (also shown in Figure 15):

- Value of x_data: 11
- Value of y_data: 9
- Value of z_data: 243

Figure 15: Screenshot of Output of the Program

3.2.5 Conclusion

The I2C bus is simple to implement and hence it is used by many integrated circuits. In embedded systems, this serial communication protocol, I2C is used to reduce the wires. The microcontrollers or the embedded devices do not need any special I2C interfaces. It is advantageous since it provides easier controlling of the bus and reduces the number of pin counts. The I2C specification shows flexibility, and it can communicate with slower devices.

For large data transfer it has various high-speed modes. To connect the integrated circuits on the board, the I2C is and will remain one of the most popular interfaces.

References

[1] https://en.wikipedia.org/wiki/I%C2%B2C, Inter-Integrated Circuits and applications using I2C, 3rd August 2025, 2:30 p.m.

[2] UM10204 I2C- Bus Specification and User Manual, Rev.6- 4 April 2014.

[3] http://i2c.info/, I2C Info, I2C Bus, Interface, protocol, and bus specifications, 2nd August 2025, 8:45 a.m.

[4] http://i2c.info/i2c-bus-specification, The I2C bus specifications, 2nd August 2025, 8:45 a.m.

[5] D. Molloy, Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux, 2nd ed. New York: Wiley, 2019. pp. 745-766.

[6] A. Henderson and A. Prakash, Android for the BeagleBone Black. Birmingham, UK: Packt Publishing Ltd, 2015.

A. Appendix

A.1 Program

```
#include<sys/ioctl.h>

#include<fcntl.h>

#include<stdio.h>

#include<linux/i2c-dev.h>

#define PATH 30

int main()

{

/****************Opening i2c bus adaptor**************/

    char Buf[PATH] ;

    int fd;
```

```
        snprintf(Buf, sizeof(Buf), "/dev/i2c-%d", 1);
        fd = open(Buf, O_RDWR);
        if (fd < 0)
        {
           printf("Cannot open device");
           return 0;
        }
        printf("File descriptor to access i2c2 is:%d", fd);
/****************Set slave for i2c bus******************/
        if (ioctl(fd, I2C_SLAVE, 0x53) < 0)
        {
           printf("\nProblem in setting slave");
           return 0;
        }
        printf("\nSetting slave device 0x53 for i2c2");
/*****************Reading Device ID******************/
        int byte;
        i2c_smbus_write_byte(fd, 0x00);
        if ((byte = i2c_smbus_read_byte(fd)) < 0)
        {
           printf("\nCannot read device id");
           return 0;
```

```
        }
        printf("\nDevice Id of sendor is:%d", byte);
/********************Configure Sensor*****************/
unsigned char buff[2];
        buff[0] = 0x2d;
        buff[1] = 0x08;
        write(fd, buff, 2);
        buff[0] = 0x31;
        buff[1] = 0x0b;
        write(fd, buff, 2);
/***************Reading Sensor Values******************/
int x_data0, x_data1, y_data0, y_data1, z_data0, z_data1;
i2c_smbus_write_byte(fd, 0x32);
        x_data0 = i2c_smbus_read_byte(fd);
        i2c_smbus_write_byte(fd, 0x33);
        x_data1 = i2c_smbus_read_byte(fd);
        i2c_smbus_write_byte(fd, 0x34);
        y_data0 = i2c_smbus_read_byte(fd);
        i2c_smbus_write_byte(fd, 0x35);
        y_data1 = i2c_smbus_read_byte(fd);
        i2c_smbus_write_byte(fd, 0x36);
        z_data0 = i2c_smbus_read_byte(fd);
```

```
        i2c_smbus_write_byte(fd, 0x37);

        z_data1 = i2c_smbus_read_byte(fd);

        printf("\n\nX Values: %d %d", x_data0, x_data1);

        printf("\nY Values: %d %d", y_data0, y_data1);

        printf("\nZ Values: %d %d", z_data0, z_data1);

        return 0;

}
```

A.2 Output of the Program

File descriptor to access i2c2 is: xxxx

Setting slave device 0x53 for i2c2

Device Id of sender is:229

X Values: xx xx

Y Values: yy yy

Z Values: zz zz