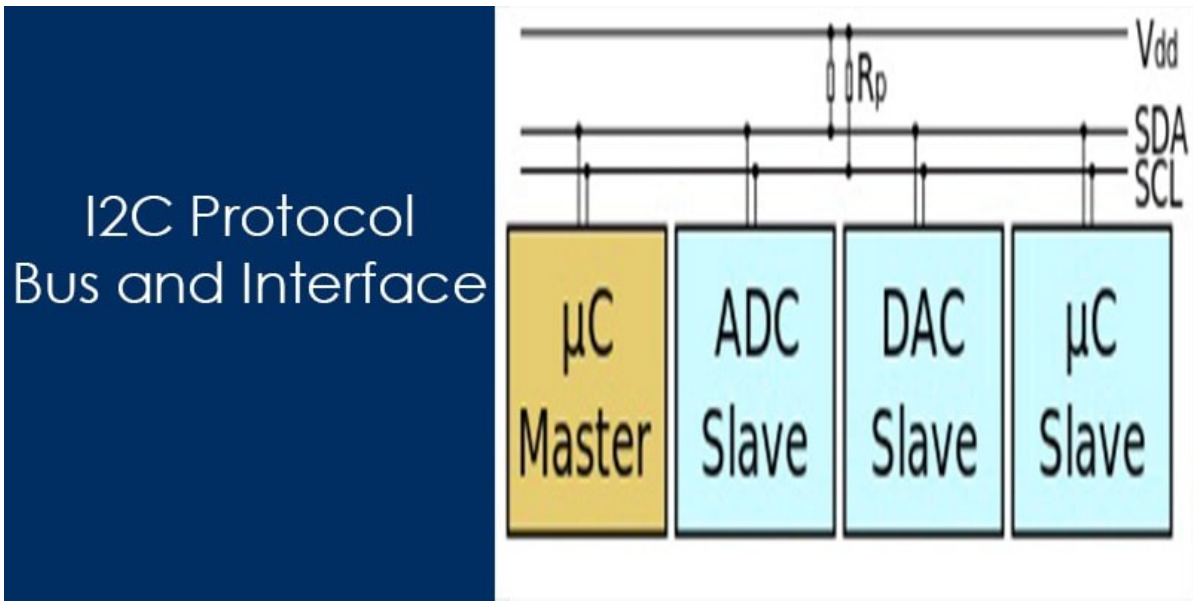


An I2C protocol is one of the serial communication protocol that is used for the chip to chip communication. Similar to the I2C protocol, SPI and UART also used for the chip to chip communication.

The I2C is the short form of Inter-Integrated Circuit, is a type of bus, which designed and developed by Philips in 1980 for inter-chip communication.I2C is adopted by a lot of vendor companies for the chip to chip communication.

It is a multi-master and multi-slave serial communication protocol means that we have the freedom to attach multiple IC at a time with the same bus. In I2C protocol, communication always started by the master and in the case of multi-master, only one master has the ownership of the bus.

In this article, you will learn the I2C protocol and its bus configuration and uses in the chip to chip communication.

What is I2C?

I2C is a serial communication protocol. It provides the good support to the slow devices, for example, EEPROM, ADC, and RTC etc.
I2c is not only used with the single board but also used with the other external components which have connected with boards through the cables.

I2C is basically two-wire communication protocol.It uses only two wire for the communication. In which one wire is used for the data (SDA) and other wire is used for the clock (SCL).

In I2C, both buses are bidirectional, which means master able to send and receive the data from the slave. The clock bus is controlled by the master but in some situations slave is also able to suppress the clock signal, but we will discuss it later.

Additionally, an I2C bus is used in the various control architecture, for example, SMBus (System Management Bus), PMBus (Power Management Bus), IPMI (Intelligent Platform Management Interface) etc.

Why use I2C

Unlike the serial com port I2c is the synchronous communication, in I2C both master and slave use the shared clock which is produced by the master.

In serial port, both the transmitter and receiver device have own clock generator. Hence it is very important to minimize the difference between the clock of the transmitter and slave otherwise data will be corrupt during the communication.

Another disadvantage of asynchronous serial com port is that only two devices transmit and receive the data but beside it, I2C can be multi-master and multi-slave.

Asynchronous serial is used a UART chip for the communication.
There is no specific limit defined for the asynchronous communication but most of the serial devices support up to the maximum baud rate 230400 (bits per second).

SPI is full duplex and faster than I2c Although sometimes I2C is much easier and beneficial. In I2C, we needed only two wire for the communication but in SPI we have needed four wire for the communication.I2c can be multi-master but SPI never can be multi-master.

These are some advantage which forced to us to use the I2C protocol in communication.

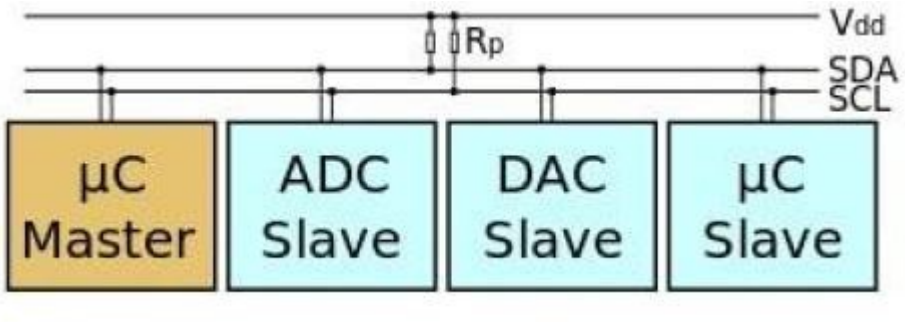
Feature of I2C Bus

- In I2C only two buses are required for the communication, the serial data bus (SDA) and serial clock bus (SCL).
- Each component in I2C bus is software addressable by a unique address, this unique address is used by the master to communicate with a particular slave.
- Always a master and slave relationships exist at all times in I2C.
- In I2C, Alway's communication is started by the master.
- The I2C bus provides the ability of the arbitration and collision detection.
- I2C is the 8-bit oriented serial bidirectional communication, there are following speed mode in I2C

MODE	SPEED
Standard-mode	100 kbit/s
Fast-mode	400 kbit/s
Fast-mode Plus	1 Mbit/s
High-speed mode	3.4 Mbit/s

I2C Physical layer

I2C is pure master and slave communication protocol, it can be the multi-master or multi-slave but we generally see a single master in I2C communication. In I2C only two wire are used for communication, one is data bus (SDA) and the second one is the clock bus (CLK).



All slave and master are connected with same data and clock bus, here important thing is to remember these buses are connected to each other using the WIRE-AND configuration which is done by to putting both pins is open drain. The wire-AND configuration allows in I2C to connect multiple nodes to the bus without any short circuits from signal contention.

The open drain allows the master and slave to drive the line low and release to high impedance state. So In that situation, when master and slave release the bus, need a pull resistor to pull the line high. The value of the pull-up resistor is very important as per the perspective of the design of I2C system because the incorrect value of pull-up resistor can lead to signal loss.

Note: *We know that I2c communication protocol supports the multiple masters and multiple slaves, but most system designs include only one master.*

I2C protocol

I2C is the very easy chip to chip communication protocol. In I2C, communication is always started by the master. When the master wants to communicate with slave then he asserts a start bit followed by the slave address with read/write bit.
After the asserting of the start bit, all slave comes in the attentive mode. If the transmitted address match with any of the slave on the bus then an ACKNOWLEDGEMENT (ACK) bit is sent by the slave to the master.

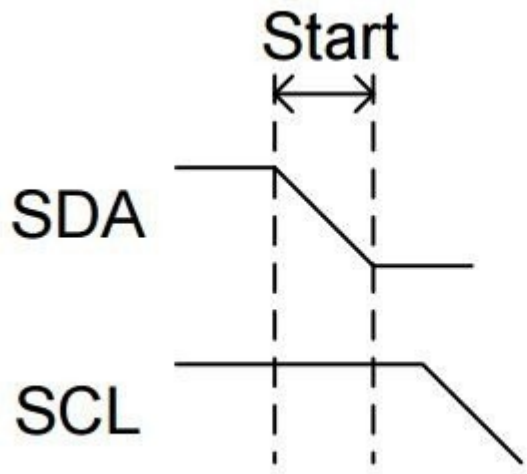
After getting the ACK bit master start the communication. If there is no slave whose address match with the transmitted address then master received a NOT-ACKNOWLEDGEMENT (NACK) bit, in that situation either master assert the stop bit to stop the communication or assert a repeated start bit on the line for new communication.

Data Frame overview of I2C protocol

I2C is an eight-bit communication protocol, in I2C we get ACK (acknowledgment) or NACK (Not Acknowledgment) bits after each byte. Here, I am describing some important terms which related to I2c data frame.

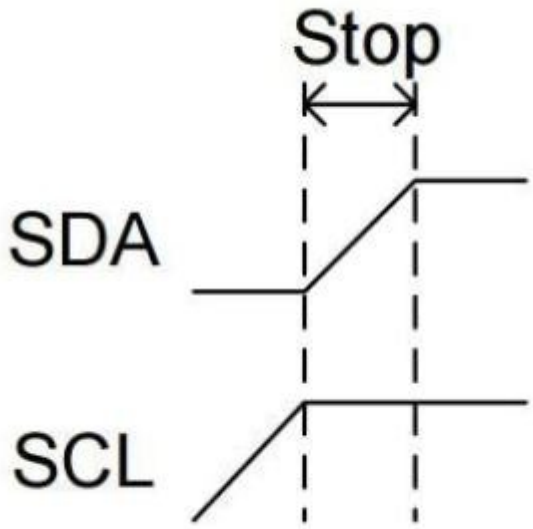
START CONDITION

The default state of SDA and SCL line is high. A master asserts the start condition on the line to start the communication. A high to low transition of SDA line while the SCL line is high called the START condition. The START condition is always asserted by the master. The I2C bus is considered busy after the assertion of the START bit.



STOP CONDITION

The STOP condition is asserted by the master to stop the communication. A Low to high transition of SDA line while the SCL line is high called the STOP condition. The STOP condition is always asserted by the master. The I2C bus is considered free after the assertion of the STOP bit.



Note: A START and STOP condition always asserted by the master.

REPEATED START

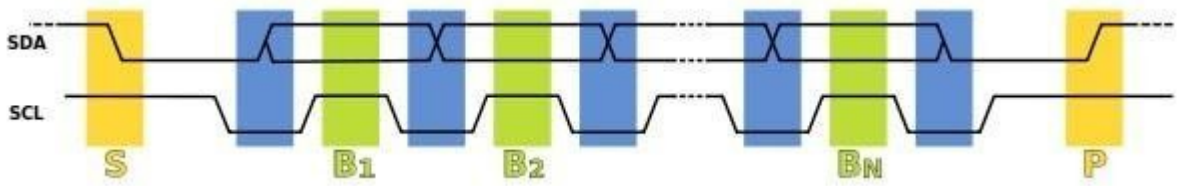
The repeated start condition similar to the START condition but both are different to each other. The repeated start is asserted by the master before the stop condition (When the bus is not in an idle state).

A Repeated Start condition is asserted by the master when he does not want to lose their control from the bus. The repeated start is beneficial for the master when it wants to start a new communication without the asserting the stop condition.

Note: Repeated start is beneficial when more than one master connected with the I2c Bus.

BYTE FORMAT IN I2C PROTOCOL

In I2C, every data which is transmitted over the SDA line must be eight-bit longs. It is very important to remember in I2c that data bit is always transmitted from the MSB and we can send or receive any number of bytes in I2C between the start and stop condition.



When we send or receive the bytes in i2c, we always get a NACK bit or ACK bit after each byte of the data is transferred during the communication.

In I2C, one bit is always transmitted on every clock. A byte which is transmitted in I2C could be an address of the device, the address of register or data which is written to or read from the slave.

In I2C, SDA line is always stable during the high clock phase except for the start condition, stop condition and repeated start condition. The SDA line only changes their state during low clock phase.

Note: SDA can only change their state only SCL is low except the start, repeated start and stop condition.

Handshaking Process in I2C Protocol

In I2C for each byte, an acknowledgment needs to be sent by the receiver, this acknowledgment bit is a proof that data is properly received by the receiver and it wants to continue the communication.

A master starts the communication to assert a start condition on the bus. After the start condition master is transmitted a 7-bit address with associated a read or write bits (here I am discussing 7-bit address).

After the transmission of the address byte, master release the data lines to put the data line (SDA) in high impedance state, which allows the receiver to give the acknowledgment.

If this transmitted address is matched with any receiver then it pulls down the SDA lines low for the acknowledgment and after the acknowledgment, it releases the data lines. The master generates a clock pulse to read this acknowledgment bit and continue the read or write operation.

If this transmitted address is not matched with any receiver then nobody is pull down the data lines low, master understands it is a NACK and in that situation, master asserts a stop bit or repeated start bit for further communication.

Acknowledge (ACK) and Not Acknowledge (NACK)

In above lines, I have already described the ACK (acknowledgment) and NACK (Not Acknowledgement) bits and their importance in I2C protocol.

In below section, I am describing some scenario, where NACK bit is generated.

- When the receiver is unable to receive or transmit the data, in that situation it generates a NACK bit to stop the communication.
- During the communication, if the receiver gets any data or commands which are not understood by the receiver then it generates a NACK bit.
- During the transfer, if the receiver performs any real-time operation and not able to communicate with master then assert a NACK bit.

- When Master is a receiver and reads the data from the slave, then after the reading of whole data it asserts a NACK bit on data lines to stop the communication.
- If there is no device present in the I2c bus of the same address which is transmitted by the master, then the master will not get the acknowledge by any slave and treat this situation as NACK.

I2C write operation

In I2c before the performing, the write operation master has to assert a start condition on I2c bus with the slave's address and write control bit (for write operation control bit will be 0).

If the transmitted address match with any slave which connected to the I2C bus then master receives an acknowledge bit. After getting the ACK bit master send the address of the register, where it wishes to write, the slave will acknowledge again, letting the master know it is ready for the write operation.

After getting this acknowledgment, the master will start sending the data to the slave. Master will get the acknowledgment of each transmitted byte of data.

If the master does not get the acknowledgment from the slave then master assert a stop condition to stop the communication or either assert the repeated start to establish a new communication. There or another option to stop the communication when the master has sent all the data than the master is terminated the transmission with a STOP condition.

I2C Read operation

I2c Read operation same as the I2C write operation, In which master asserts the start condition before the read operation. After the start condition master transmit the slave address with Read control bit (for read operation control bit will be 1), if the transmitted address match with any device in the I2c bus then it acknowledges to the master to pulling down the data bus(SDA).

After getting ACK bit, master release the data bus but continue sending the clock pulse, in that situation master become receiver and slave become the slave transmitter.

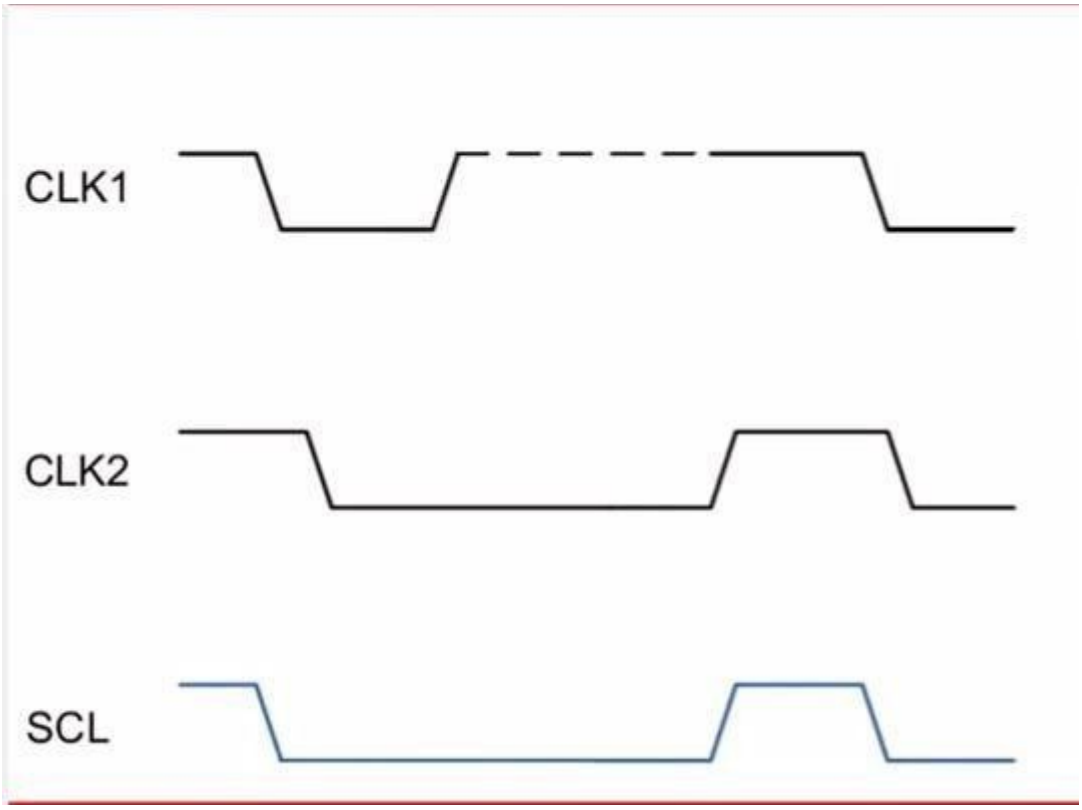
In the read operation, the master gives the acknowledgment to the slave on the receiving of every byte of data to let the slave know that it is ready for more data. Once the master has received the number of bytes which it is expecting, it will send a NACK bit to release the bus and assert the stop bit to halt the communication.

Some Special cases in I2C

Clock synchronization in I2C

Unlike Rs232, I2c is synchronous communication, in which clock is always generated by the master and this clock is shared by both master and slave.
In the case of multi-master, all master generate their own SCL clock, hence it is necessary that clock of all master should be synchronized. In the i2C, this clock synchronization is done by wired and logic.

For the better understanding, I am taking an example, where two masters try to communicate with a slave. In that situation, both masters generate their own clock, master M1 generate clk1 and master M2 generate clk2 and clock which observed on the bus is SCL.



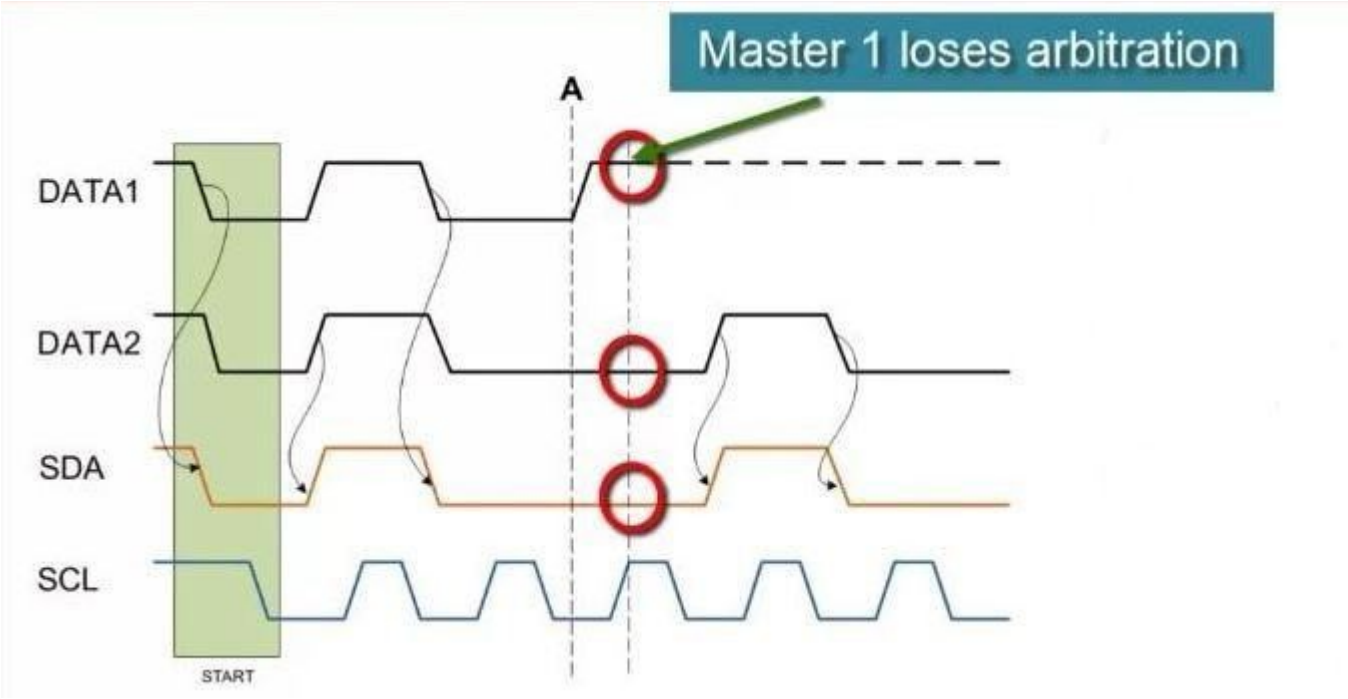
The SCL clock would be the Anding (clk1 & clk2) of clk1 and clk2 and most interesting thing is that highest logic 1 of SCL line defines by the CLK which has lowest logic 1.

Arbitration in I2C

The arbitration is required in case of multi-master, where more than one master is tried to communicate with a slave simultaneously. In I2C arbitration is achieved by the SDA line.

For Example,

Suppose two masters in the I2C bus is tried to communicate with a slave simultaneously then they will assert a start condition on the bus. The SCL clock of the I2c bus would be already synchronized by the wired and logic.



In the above case, everything will be good till the state of SDA line will same what is the masters driving on the bus. If any master sees that the state of SDA line differs, what is it driving then they will exit from the communication and lose their arbitration.

Note: Master which is losing their arbitration will wait till bus become free.

Clock stretching in I2C

In I2c, communication can be paused by the clock stretching to holding the SCL line low and it cannot continue until the SCL line released high again.

In I2C, slave able to receive a byte of data on the fast rate but sometimes slave takes more time in processing the received bytes in that situation slave pull the SCL line to pause the transaction and after the processing of the received bytes, it again released the SCL line high again to resume the communication.

The clock stretching is the way in which slave drive the SCL line but it is the fact, most of the slave do not drive the SCL line

Note: *In I2c communication protocol, most of the I2C slave devices do not use the clock stretching feature, but every master should support the clock stretching.*

Advantages of I2C communication protocol

There is a lot of advantage of I2C protocol which makes the user helpless to use the I2C protocol in many applications.

- It is the synchronous communication protocol, so no need of precise oscillators for the master and slave.
- It requires only two wire, one wire for the data (SDA) and other wire for the clock (SCL).
- It provides the flexibility to the user to select the transmission rate as per the requirements.
- In I2C Bus, each device on the bus is independently addressable.
- It follows the master and slave relationships.
- It has the capability to handle the multiple masters and multiple slaves on the I2C Bus.
- I2C has some important features like arbitration, clock synchronization, and clock stretching.
- I2C provide ACK/NACK (acknowledgment/ Not-acknowledgement) features which provide the help in error handling.

Some important limitation of I2C communication protocol

An I2C protocol has a lot of advantage but beside it, I2C has a few limitations.

- It consumes more power than other serial communication busses due to open-drain topology.
- It is good only for the short distance.
- I2C protocol has some limitation for the number of slaves, the number of the slave depends on the capacitance of the I2C bus.
- It only provides few limited communication speed like 100 kbit/s,400 kbit/s etc.
- In I2c, devices can set their communication speed, slower operational devices can delay the operation of faster speed devices.

Conclusion

An I2c is the easy and cheap communication protocol, It can be multi-master or multi-slave. In I2c we get the acknowledgment (ACK) and not acknowledgment(NACK) bits after the each transmitted byte. Some disadvantage also attaches with I2C, it is a half-duplex communication and slow as compared to SPI (serial peripheral communication).

What is I2C communication?

I2C is a serial communication protocol. It provides good support to the slow devices, for example, EEPROM, ADC, I2C LCD, and RTC etc. It is not only used with the single board but also used with the other external components which have connected with boards through the cables.

I2C is basically a two-wire communication protocol. It uses only two wire for communication. In which one wire is used for the data (SDA) and other wire is used for the clock (SCL).

In I2C, both buses are bidirectional, which means master able to send and receive the data from the slave. The clock bus is controlled by the master but in some situations slave is also able to suppress the clock signal, but we will discuss it later.

Additionally, an I2C bus is used in the various control architecture, for example, SMBus (System Management Bus), PMBus (Power Management Bus), IPMI (Intelligent Platform Management Interface) etc.

What does I2C stand for?

Inter-Integrated Circuit

How many wires are required for I2C communication?

In I2C only two buses are required for the communication, the serial data bus (SDA) and serial clock bus (SCL).

I2C is half duplex or full duplex?

half duplex

I2C is Synchronous or Asynchronous Communication?

I2C is Synchronous Communication

Explain the physical layer of the I2C protocol

I2C is pure master and slave communication protocol, it can be the multi-master or multi-slave but we generally see a single master in I2C communication. In I2C only two wire are used for communication, one is data bus (SDA) and the second one is the clock bus (CLK).

All slave and master are connected with same data and clock bus, here important thing is to remember these buses are connected to each other using the WIRE-AND configuration which is done by to putting both pins is open drain. The wire-AND configuration allows in I2C to connect multiple nodes to the bus without any short circuits from signal contention.

The open drain allows the master and slave to drive the line low and release to high impedance state. So In that situation, when master and slave release the bus, need a pull resistor to pull the line high. The value of the pull-up resistor is very important as per the perspective of the design of the I2C system because the incorrect value of the pull-up resistor can lead to signal loss.

Note: *We know that I2c communication protocol supports the multiple masters and multiple slaves, but most system designs include only one master.*

Explain the operation and frame of I2C protocol

I2C is a chip to chip communication protocol. In I2C, communication is always started by the master. When the master wants to communicate with slave then he asserts a start bit followed by the slave address with read/write bit.

After asserting the start bit, all slave comes in the attentive mode. If the transmitted address match with any of the slave on the bus then an ACKNOWLEDGEMENT (ACK) bit is sent by the slave to the master.

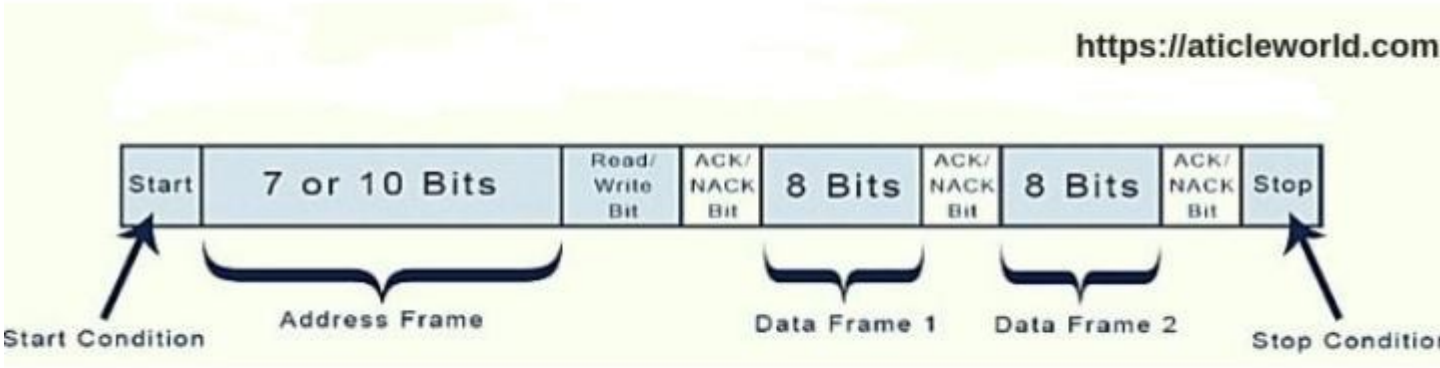
After getting the ACK bit, master starts the communication. If there is no slave whose address match with the transmitted address then master received a NOT-ACKNOWLEDGEMENT (NACK) bit, in that situation either master assert the stop bit to stop the communication or assert a repeated start bit on the line for new communication.

When we send or receive the bytes in i2c, we always get a NACK bit or ACK bit after each byte of the data is transferred during the communication.

In I2C, one bit is always transmitted on every clock. A byte which is transmitted in I2C could be an address of the device, the address of register or data which is written to or read from the slave.

In I2C, SDA line is always stable during the high clock phase except for the start condition, stop condition and repeated start condition. The SDA line only changes their state during the low clock phase.

See the below image,



I2C FRAME

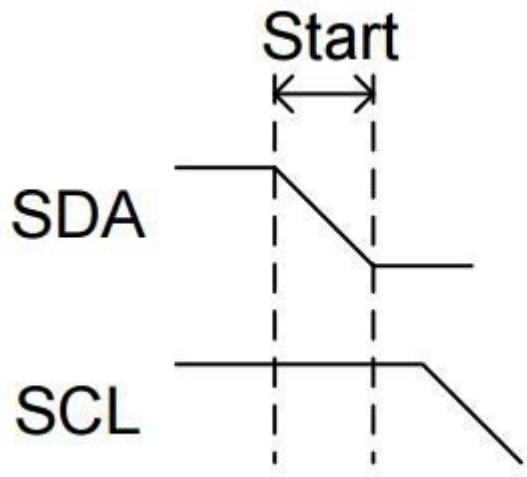
Start Bit: Start the communication

Stop bit: Stop communication.

What is START bit and STOP bit?

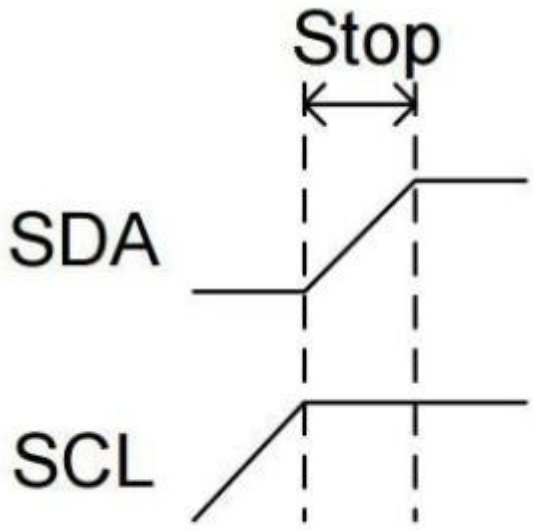
Start Condition:

The default state of SDA and SCL line is high. A master asserts the start condition on the line to start the communication. A high to low transition of the SDA line while the SCL line is high called the START condition. The START condition is always asserted by the master. The I2C bus is considered busy after the assertion of the START bit.



Stop Condition:

The STOP condition is asserted by the master to stop the communication. A Low to high transition of SDA line while the SCL line is high called the STOP condition. The STOP condition is always asserted by the master. The I2C bus is considered free after the assertion of the STOP bit.



Note: A START and STOP condition always asserted by the master.

Note: You can also see, Embedded c interview questions

What is the repeated start condition?

The repeated start condition similar to the START condition but both are different from each other. The repeated start is asserted by the master before the stop condition (When the bus is not in an idle state).

A Repeated Start condition is asserted by the master when he does not want to lose their control from the bus. The repeated start is beneficial for the master when it wants to start a new communication without the asserting the stop condition.

Note: Repeated start is beneficial when more than one master connected with the I2c Bus.

What is the standard bus speed in I2C?

There are following speed mode in I2C

MODE	SPEED
Standard-mode	100 kbit/s
Fast-mode	400 kbit/s
Fast-mode Plus	1 Mbit/s

High-speed mode	3.4 Mbit/s

What is the limiting factor as to how many devices can go on the I²C bus?

It depends on the total capacitance.

Is it possible to have multiple masters in I2C?

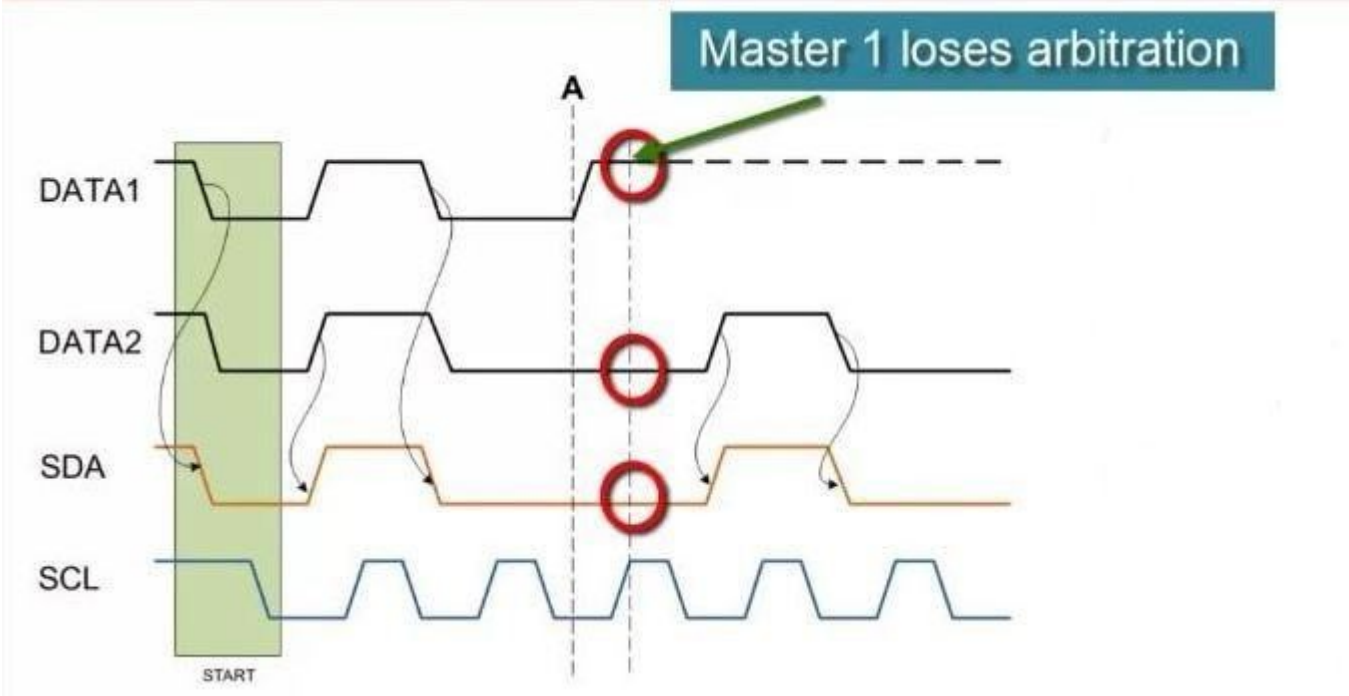
Yes I2C support multiple master and multiple slaves.

What is a bus arbitration?

The arbitration is required in case of multi-master, where more than one master is tried to communicate with a slave simultaneously. In I2C arbitration is achieved by the SDA line.

For Example,

Suppose two masters in the I2C bus is tried to communicate with a slave simultaneously then they will assert a start condition on the bus. The SCL clock of the I2c bus would be already synchronized by the wired and logic.



In the above case, everything will be good till the state of SDA line will same what is the masters driving on the bus. If any master sees that the state of SDA line differs, what is it driving then they will exit from the communication and lose their arbitration.

Note: Master which is losing their arbitration will wait till the bus become free.

What is I2C clock stretching?

In I2c, communication can be paused by the clock stretching to holding the SCL line low and it cannot continue until the SCL line released high again.

In I2C, slave able to receive a byte of data on the fast rate but sometimes slave takes more time in processing the received bytes in that situation slave pull the SCL line to pause the transaction and after the processing of the received bytes, it again released the SCL line high again to resume the communication.

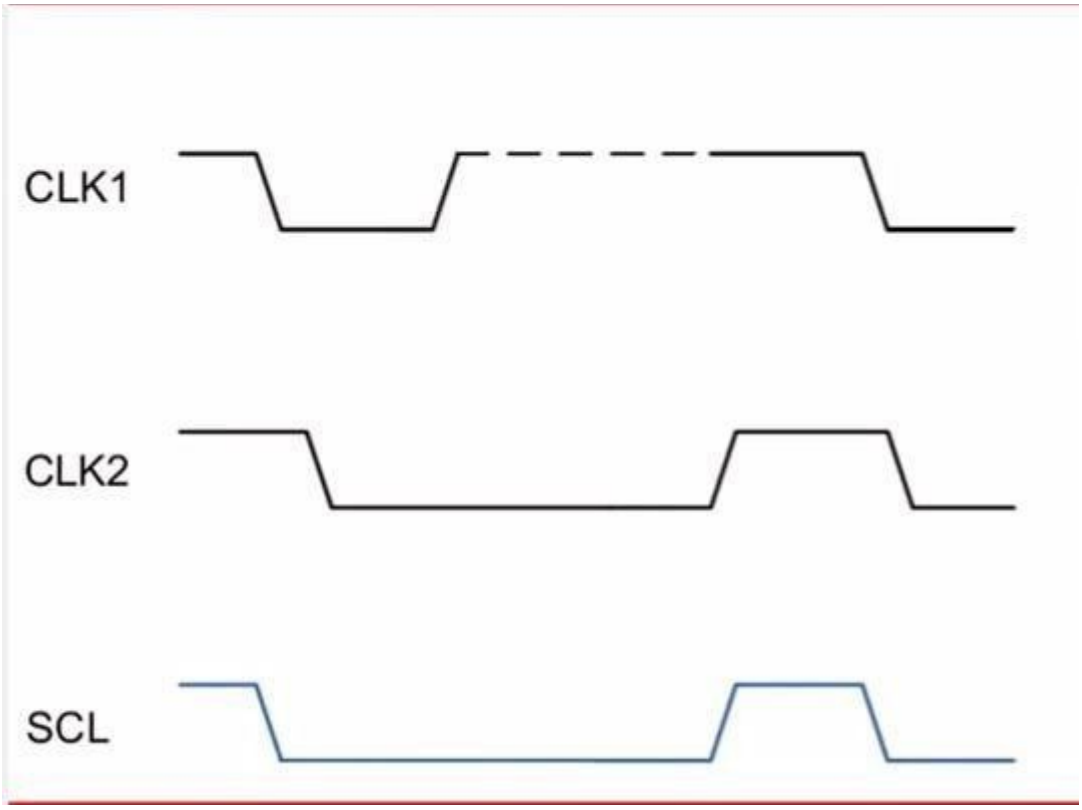
The clock stretching is the way in which slave drive the SCL line but it is the fact, most of the slave does not drive the SCL line

Note: *In I2c communication protocol, most of the I2C slave devices do not use the clock stretching feature, but every master should support the clock stretching.*

What is I2C clock synchronization?

Unlike Rs232, I2c is synchronous communication, in which clock is always generated by the master and this clock is shared by both master and slave. In the case of multi-master, all master generate their own SCL clock, hence it is necessary that clock of all master should be synchronized. In the i2C, this clock synchronization is done by wired and logic.

For a better understanding, I am taking an example, where two masters try to communicate with a slave. In that situation, both masters generate their own clock, master M1 generate clk1 and master M2 generate clk2 and clock which observed on the bus is SCL.



The SCL clock would be the Anding (clk1 & clk2) of clk1 and clk2 and most interesting thing is that highest logic 1 of SCL line defines by the CLK which has lowest logic 1.

When must data be stable for a correct I²C bus transaction?

When the clock is high

Is Hot swapping possible in I2C protocol?

Yes, hot swapping is possible in I2C.

If a slave is servicing an internal interrupt, what will it do to avoid losing data?

The slave will stretch the clock until the interrupt servicing is complete.

Advantages of I2C communication?

There is a lot of advantage of I2C protocol which makes the user helpless to use the I2C protocol in many applications.

- It is the synchronous communication protocol, so there is no need of a precise oscillator for the master and slave.
- It requires only two wire, one wire for the data (SDA) and other wire for the clock (SCL).

- It provides the flexibility to the user to select the transmission rate as per the requirements.
- In I2C Bus, each device on the bus is independently addressable.
- It follows the master and slave relationships.
- It has the capability to handle the multiple masters and multiple slaves on the I2C Bus.
- I2C has some important features like arbitration, clock synchronization, and clock stretching.
- I2C provide ACK/NACK (acknowledgment/ Not-acknowledgement) features which provide help in error handling.

What are the limitations of I2C interface?

- Half duplex communication, so data is transmitted only in one direction (because of the single data bus) at a time.
- Since the bus is shared by many devices, debugging an I2C bus (detecting which device is misbehaving) for issues is pretty difficult.
- The I2C bus is shared by multiple slave devices if anyone of these slaves misbehaves (pull either SCL or SDA low for an indefinite time) the bus will be stalled. No further communication will take place.
- I2C uses resistive pull-up for its bus. Limiting the bus speed.
- Bus speed is directly dependent on the bus capacitance, meaning longer I2C bus traces will limit the bus speed.

What is the difference between SPI and I2C (I2C vs SPI)?

You can see this article, Difference between I2c and SPI

Questions for you:

- What is locking(or waiting) and unlocking I2c protocol? How you could design the unlocking I2c protocol for your system.
- Which is better to use I2C or SPI?
- I2C is Edge Triggering or Level Triggering?
- Is in I2c two slaves have the same address?
- How will the master indicate that it is either address/data? How will it intimate to the slave that it is going to either read/write?
- What is the voltage level for 0 and 1 in I2C?
- How could a slave send the data to the Master in I2C while the master is communicating with another slave?

I2C	SPI
I2C can be multi-master and multi-slave, which means there can be more than one master and slave attached to the I2C bus	SPI can be multi-save but does not a multi-master serial protocol, that means there can be only one master attached to SPI bus.
I2C is half-duplex communication protocol.	SPI is a full duplex commination protocol.
I2C has the feature of clock stretching, that means if the slave cannot able to send fast data as fast enough then it suppresses the clock to stop the communication.	Clock stretching is not the feature of SPI.
I2C is used only two wire for the communication, one wire is used for the data and the second wire is used for the clock.	SPI needs three or four wire for communication ((depends on requirement), MOSI, MISO, SCL and Chip-select pin.
I2C is slower than SPI.	In comparison to I2C, SPI is faster.
I2C draws more power than SPI.	Draws less power as compared to I2C.

I2C is less susceptible to noise than SPI	SPI is more susceptible to noise than I2C.
I2C is cheaper to implement than the SPI communication protocol.	Costly as compare to I2C.
I2C work on wire and logic and it has a pull-up resistor.	There is no requirement of pull-up resistor in case of the SPI.
In I2C communication we get the acknowledgment bit after each byte.	Acknowledgment bit is not supported by the SPI communication protocol.
I2C ensures that data sent is received by the slave device.	SPI does not verify that data is received correctly or not.
I2C support the multi-master communication.	SPI does not support multi -master communication.
I2C is a multi-master communication protocol that’s why it has the feature of arbitration.	SPI is not a multi-master communication protocol, so it does not consist the properties of arbitration.

I2C is the address base bus protocol, you have to send the address of the slave for the communication.	In case of the SPI, you have to select the slave using the slave select pin for the communication.
I2C has some extra overhead due to start and stop bits.	SPI does not have a start and stop bits.
I2C supports multiple devices on the same bus without any additional select lines (work on the basis of device address).	SPI requires additional signal (slave select lines) lines to manage multiple devices on the same bus.
I2C is better for long distance.	SPI is better for the short distance.
I2C is developed by NXP.	SPI is developed by Motorola.

I2C is a protocol for communication between devices. In this column, the author takes the reader through the process of writing I2C clients in Linux.

I2C is a multi-master synchronous serial communication protocol for devices. All devices have addresses through which they communicate with each other. The I2C protocol has three versions with different communication speeds 100kHz, 400kHz and 3.4MHz. The I2C protocol has a bus arbitration procedure through which the master is decided on the bus, and then the master supplies the clock for the system and reads and writes data on the bus. The device that is communicating with the master is the slave device.

The Linux I2C subsystem

The Linux I2C subsystem is the interface through which the system running Linux can interact with devices connected on the system's I2C bus. It is designed in such a manner that the system running Linux is always the I2C master. It consists of the following subsections.

I2C adapter: There can be multiple I2C buses on the board, so each bus on the system is represented in Linux using the *struct i2c_adapter (defined in include/linux/i2c.h)*. The following are the important fields present in this structure.
bus number: Each bus in the system is assigned a number that is present in the I2C adapter structure which represents it.

I2C algorithm: Each I2C bus operates with a certain protocol for communicating between devices. The algorithm that the bus uses is defined by this field. There are currently three algorithms for the I2C bus, which are pca, pcf and bitbanging. These algorithms are used to communicate with devices when the driver requests to write or read data from the device.

I2C client: Each device that is connected to the I2C bus on the system is represented using the *struct i2c_client (defined in include/linux/i2c.h)*. The following are the important fields present in this structure.
Address: This field consists of the address of the device on the bus. This address is used by the driver to communicate with the device.

Name: This field is the name of the device which is used to match the driver with the device.
Interrupt number: This is the number of the interrupt line of the device.
I2C adapter: This is the *struct i2c_adapter* which represents the bus on which this device is connected. Whenever the driver makes requests to write or read from the bus, this field is used to identify the bus on which this transaction is to be done and also which algorithm should be used to communicate with the device.

I2C driver: For each device on the system, there should be a driver that controls it. For the I2C device, the corresponding driver is represented by *struct i2c_driver (defined in include/linux/i2c.h).* The following are the important fields defined in this structure.

Driver.name: This is the name of the driver that is used to match the I2C device on the system with the driver.

Probe: This is the function pointer to the driver's probe routine, which is called when the device and driver are both found on the system by the Linux device driver subsystem.
To understand how to write I2C device information and the I2C driver, let's consider an example of a system in which there are two devices connected on the I2C bus. A description of these devices is given below.
Device 1
Device type: EEPROM
Device name: eeprom_xyz
Device I2C address: 0x30
Device interrupt number: 4
Device bus number: 2

Device 2
Device type: Analogue to digital converter
Device name: adc_xyz
Device I2C address: 0x31
Device interrupt number: Not available
Device bus number: 2

Writing the I2C device file

I2C devices connected on the system are represented by *struct i2c_client.* This structure is not directly defined but, instead, struct *i2c_board_info* is defined in the board file. *struct i2c_client* is defined using *struct i2c_board_info* structure by the Linux I2C subsystem, the fields of the *i2c_board_info* object is copied to *i2c_client* object created.

Note: Board files reside in *arch/ folder* in Linux. For example, the board file for the ATSTK1000 board of the AVR32 architecture is *arch/avr32/boards/atstk1000.c* and the board file for Beagle Board of ARM OMAP3 architecture is *arch/arm/mach-omap2/board-omap3beagle.c.*

struct i2c_board_info (defined in include/linux/i2c.h) has the following important fields.
type: This is the name of the I2C device for which this structure is defined. This will be copied to the name field of *i2c_client object* created by the I2C subsystem.
addr: This is the address of the I2C device. This field will be copied to address the field of *i2c_client object* created by the I2C subsystem.
irq: This is the interrupt number of the I2C device. This field will be copied to the irq field of the *i2c_client object* created by the I2C subsystem.
An array of *struct i2c_board_info* object is created, where each object represents the I2C device connected on the bus. For our example system, the *i2c_board_info* object is written as follows:

```
static struct i2c_board_info xyz_devices[] = {
{
.type = eeprom_xyz,
.addr = 0x30,
.irq = 4,
},
{
.type = adc_xyz,
.addr = 0x31,
},
};
```

I2C device registration

I2C device registration is a process with which the kernel is informed about the device present on the I2C bus. The I2C device is registered using the *struct i2c_board_info* object defined. The kernel gets information about the device's address, bus number and name of the device being registered. Once the kernel gets this information, it stores this information

in its global linked list *__i2c_board_list*, and when the *i2c_adapter* which represents this bus is registered, the kernel creates the *i2c_client object* from this *i2c_board_info* object.

I2C device registration is done in the board init code present in the board file. I2C devices are registered in the Linux kernel using the following two methods.

Case 1: In most cases, the bus number on which the device is connected is known; in this case the device is registered using the bus number. When the bus number is known, I2C devices are registered using the following API:

int i2c_register_board_info(int busnum, struct i2c_board_info *info, unsigned len);

where,

busnum = the number of the bus on which the device is connected. This will be used to identify the *i2c_adapter* object for the device.

info = array of struct i2c_board_info object, which consists of information of all the devices present in the bus.

len = number of elements in the info array.

For our example system, I2C devices are registered as follows:

i2c_register_board_info(2, xyz_devices, ARRAY_SIZE(xyz_devices));

What the *i2c_register_board_info* does is link the *struct i2c_board_info* object in *__i2c_board_list*, which is the global linked list.

Now, when the I2C adapter is registered using *i2c_register_adapter* API (defined in drivers/i2c/i2c-core.c), it will search for devices that have the same bus number as the adapter, through the *i2c_scan_static_board_info* API. When the *i2c_board_info* object is found with the bus number which is the same as that of the adapter being registered, a new *i2c_client* object is created using the i2c_new_device API.

The *i2c_new_device* API creates a new struct i2c_client object and the fields of the *i2c_client* object are initialised with the fields of the *i2c_board_info* object. The *new i2c_client* object is then registered with the I2C subsystem. During registration, the kernel matches the name of all the I2C drivers with the name of the I2C client created. If any I2C driver's name matches with the I2C client, then the probe routine of the I2C driver will be called.

Case 2: In some cases, instead of the bus number, the *i2c_adapter* on which the device is connected is known; in this case, the device is registered using the *struct i2c_adapter* object. When the *i2c_adapter* is known instead of the bus number, the I2C device is registered using the following API:

struct i2c_client *
i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info);

where,

adap = i2c_adapter representing the bus on which the device is connected.

info = i2c_board_info object for each device.

In our example, the device is registered as follows.
For device 1:

```
i2c_new_device(adap, &xyz_devices[0]);
```

For device 2:

```
i2c_new_device(adap, &xyz_devices[1]);
```

Writing the I2C driver

As mentioned earlier, generally, the device files are present in the *arch/xyz_arch/boards* folder and similarly, the driver files reside in their respective driver folders. For example, typically, all the RTC drivers reside in the *drivers/rtc* folder and all the keyboard drivers reside in the *drivers/input/keyboard* folder.

Writing the I2C driver involves specifying the details of the *struct i2c_driver*. The following are the required fields for *struct i2c_driver* that need to be filled:

driver.name = name of the driver that will be used to match the driver with the device.

driver.owner = owner of the module. This is generally the *THIS_MODULE* macro.

probe = the probe routine for the driver, which will be called when any I2C device's name in the system matches with this driver's name.

Note: It's not just the names of the device and driver that are used to match the two. There are other methods to match them such as *id_table* but, for now, let's consider their names as the main parameter for matching. To understand the way in which the ID table is used, refer to the Linux source code.

For our example of the EEPROM driver, the driver file will reside in the *drivers/misc/eeprom* folder and we will give it a *name 'eeprom_xyz.c*. The *struct i2c_driver* will be written as follows:

```
static struct i2c_driver eeprom_driver = {
.driver = {
.name = "eeprom_xyz",
.owner = THIS_MODULE,
},
.probe = eeprom_probe,
};
```

For our example of an *adc* driver, the driver file will reside in the *drivers/iio/adc* folder, which we will name as *adc_xyz.c*, and the *struct i2c_driver* will be written as follows:

```
static struct i2c_driver adc_driver = {
.driver = {
.name = "adc_xyz",
.owner = THIS_MODULE,
},
.probe = adc_probe,
};
```

The *struct i2c_driver* now has to be registered with the I2C subsystem. This is done in the *module_init* routine using the following API:

i2c_add_driver(struct i2c_driver *drv);
- where *drv i*s the *i2c_driver* structure written for the device.

For our example of an EEPROM system, the driver will be registered as:

i2c_add_driver(&eeprom_driver);
- and the adc driver will be registered as:

i2c_add_driver(&adc_driver);

What this *i2c_add_driver* does is register the passed driver with the I2C subsystem and match the name of the driver with all the *i2c_client names*. If any of the names match, then the probe routine of the driver will be called and the *struct i2c_client* will be passed as the parameter to the probe routine. During the probe routine, it is verified that the device represented by the *i2c_client* passed to the driver is the actual device that the driver supports. This is done by trying to communicate with the device represented by *i2c_client* using the address present in the *i2c_client* structure. If this fails, it returns an error from the probe routine informing the Linux device driver subsystem that the device and driver are not compatible; or else it continues with creating device files, registering interrupts and registering with the application subsystem.

The probe skeleton for our example EEPROM system will be as follows:

```
static int eeprom_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
check if device exists;
if device error
{
return error;
}
else
{
do basic configuration of eeprom using
client->addr;

register with eeprom subsystem;

register the interrupt using client->irq;
}
return 0;
}

static int adc_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
check if device exists;
if device error
{
return error;
}
else
```

```
{
do basic configuration of adc using
client->addr;

register with adc subsystem;
}
return 0;
}
```

After the probe routine is called and all the required configuration is done, the device is active and the user space can read and write the device using system calls. For a very clear understanding of how to write *i2c_driver*, refer to the drivers present in the Linux source code for example, the RTC driver on *i2c* bus, the *drivers/rtc/rtc-ds1307.c* file and other driver files.

For reading and writing data on the I2C bus, use the following API.

Reading bytes from the I2C bus:

i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.

Reading words from the I2C bus:

i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.

Writing bytes on an I2C bus:

i2c_smbus_write_byte_data(struct i2c_client *client, u8 command, u8 data);
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.
data: the data that is to be written to the device.

Writing words on an I2C bus:

i2c_smbus_write_word_data(struct i2c_client *client, u8 command, u16 data);
client: i2c_client object received from driver probe routine.
command: the command that is to be transferred on the bus.
data: the data that is to be written to the device

When the read or write command is issued, the request is completed using the adapters algorithm, which has the routines to read and write on the bus.