# UNIT - 5
# SERIAL COMMUNICATION

## COMMUNICATION:

Communication between electronic devices is like communication between humans. Both sides need to speak the same language. In electronics, these languages are called *communication protocols*. Luckily for us, there are only a few communication protocols we need to know when building most electronics projects. In this series of articles, we will discuss the basics of the three most common protocols: SPI, I2C and UART.

SPI, I2C, and UART are quite a bit slower than protocols like USB, Ethernet, Bluetooth, and Wi-Fi, but they're a lot simpler and use less hardware and system resources. SPI, I2C, and UART are ideal for communication between microcontrollers and between microcontrollers and sensors where large amounts of high speed data don't need to be transferred.

DATA COMMUNICATION TYPES: (1) PARALLEL
(2) SERIAL: (I) ASYNCHRONOUS (II) SYNCHRONOUS

Parallel Communication:

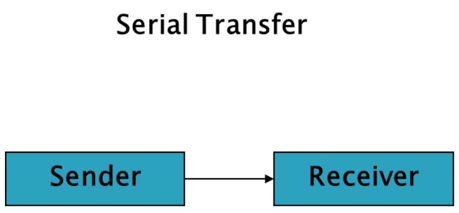
- In parallel communication, all the bits of data are transmitted simultaneously on separate communication lines.
- Used for shorter distance.
- In order to transmit n bit, n wires or lines are used.
- More costly.
- Faster than serial transmission.
- Data can be transmitted in less time.

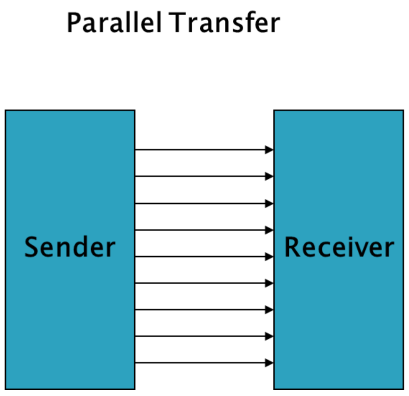
Example: printers and hard disk

**Serial Communication Basics:**

- In serial communication the data bits are transmitted serially one by one i.e. bit by bit on single communication line
- It requires only one communication line rather than n lines to transmit data from sender to receiver.
- Thus all the bits of data are transmitted on single lines in serial fashion.
- Less costly.
- Long distance transmission.

Example: Telephone.


Serial Transfer
Sender
Receiver


Parallel Transfer
Sender
Receiver

Serial communication uses two methods:

- Asynchronous.
- Synchronous.

**Asynchronous:**

- ➪ Transfers single byte at a time.
- ➪ No need of clock signal
    - ❖ Example: UART (universal asynchronous receiver transmitter)

**Synchronous:**

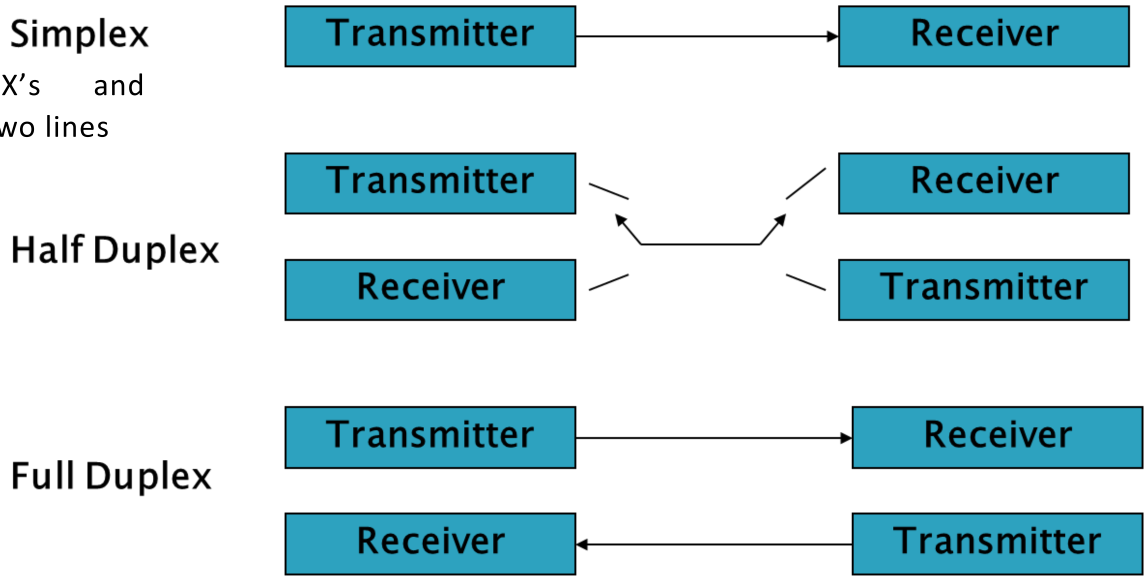
- ➪ Transfers a block of data (characters) at a time.
- ➪ Requires clock signal
    - ❖ Example: SPI (serial peripheral interface), I2C (inter integrated circuit).

**Data Transmission:**In data transmission if the data can be transmitted and received, it is a duplex transmission.

**Simplex:** Data is transmitted in only one direction i.e. from TX to RX only one TX and one RX only

**Half duplex:** Data is transmitted in two directions but only one way at a time i.e. two TX's, two RX's and one line

**Full duplex:** Data is transmitted both ways at the same time i.e. two TX's, two RX's and two lines



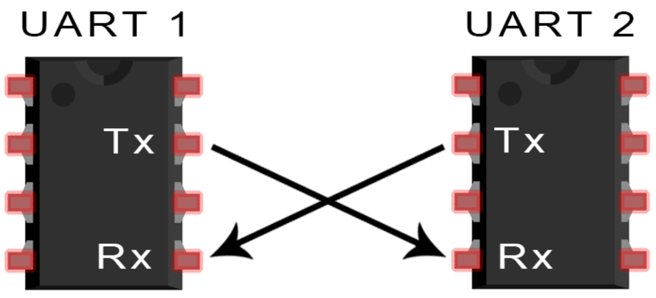
- ➢ A protocol is a set of rules agreed by both the sender and receiver on
    - How the data is packed
    - How many bits constitute a character
    - When the data begins and ends

Various Serial Communication Protocols

| **Serial Protocol** | **Synchronous /Asynchronous** | **Type** | **Duplex** | **Data transfer rate (kbps)** |
|---|---|---|---|---|
| **UART** | Asynchronous | peer-to-peer | Full-duplex | 20 |
| **I2C** | Synchronous | multi-master | Half-duplex | 3400 |
| **SPI** | Synchronous | multi-master | Full-duplex | >1,000 |
| **MICROWIRE** | Synchronous | master/slave | Full-duplex | > 625 |
| **1-WIRE** | Asynchronous | master/slave | Half-duplex | 16 |

## UART COMMUNICATION

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:



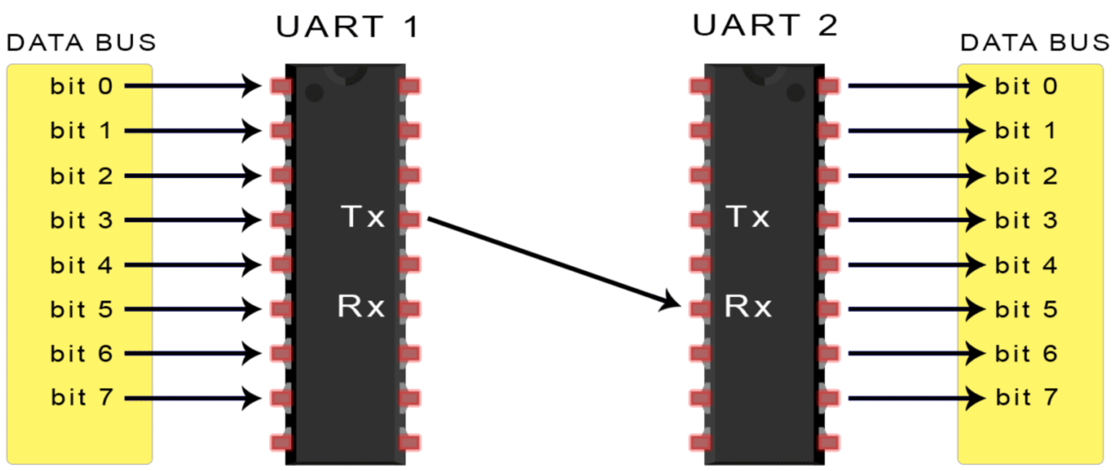
UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.
Both UARTs must be configured to transmit and receive the same data packet structure.

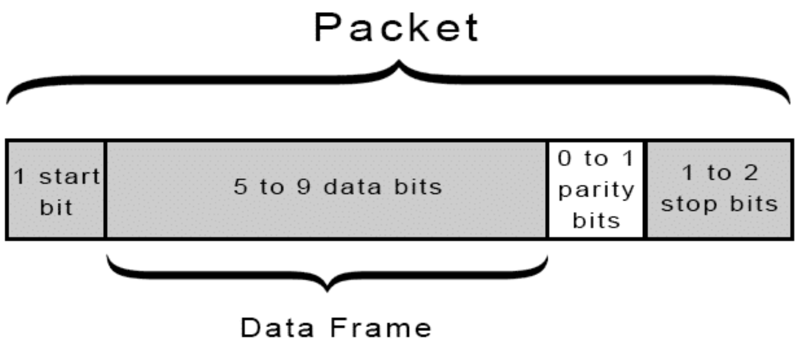
| | |
|---|---|
| **Wires Used** | 2 |
| **Maximum Speed** | Any speed up to 115200 baud, usually 9600 baud |
| **Synchronous or Asynchronous?** | Asynchronous |
| **Serial or Parallel?** | Serial |
| **Max # of Masters** | 1 |
| **Max # of Slaves** | 1 |

## HOW UART WORKS

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end:



UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



## START BIT

The UART data transmission line is normally held at a high voltage level when it’s not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to lowfor one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

## DATA FRAME

The data frame contains the actual data being transferred. It can be 5 bits to 9 bits long if a parity bit is used. If no parity bit is used, the data frame can be 8 bits long. In most cases, the data is sent with the least significant bit first.

## PARITY

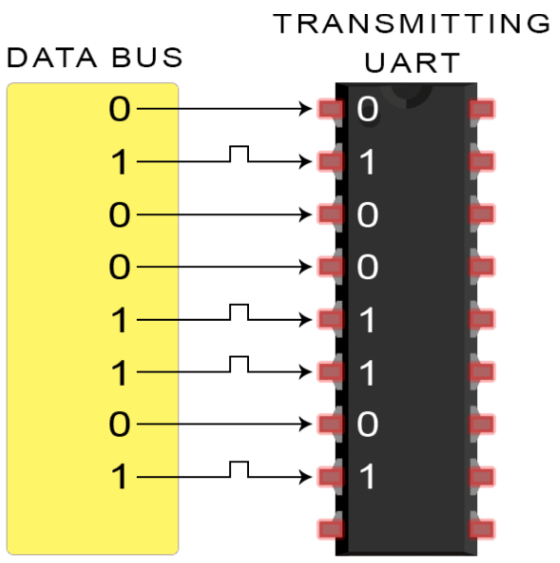
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

## STOP BITS

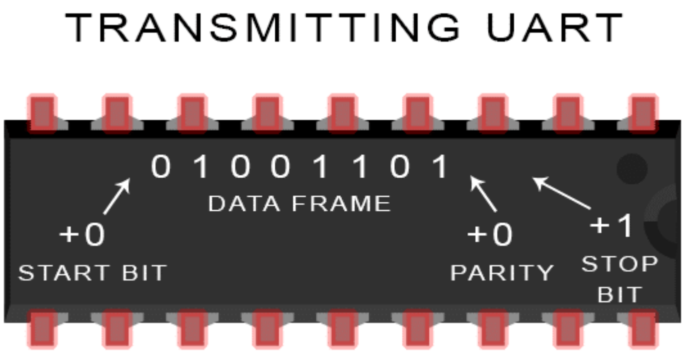
The Stop Bit, as the name suggests, marks the end of the data packet. It is usually two bits long but often only on bit is used. In order to end the transmission, the UART maintains the data line at high voltage (1).

## STEPS OF UART TRANSMISSION

1. The transmitting UART receives data in parallel from the data bus:



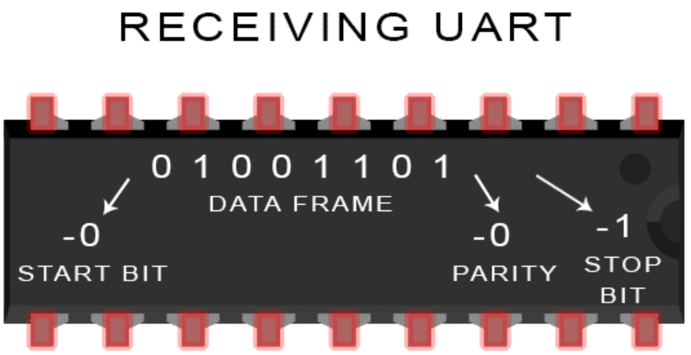
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data

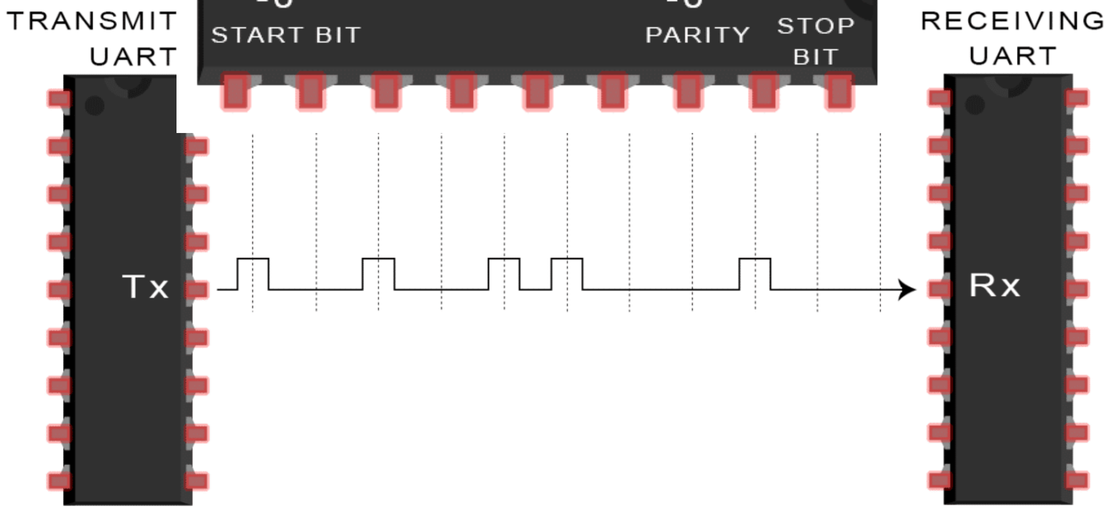


frame:

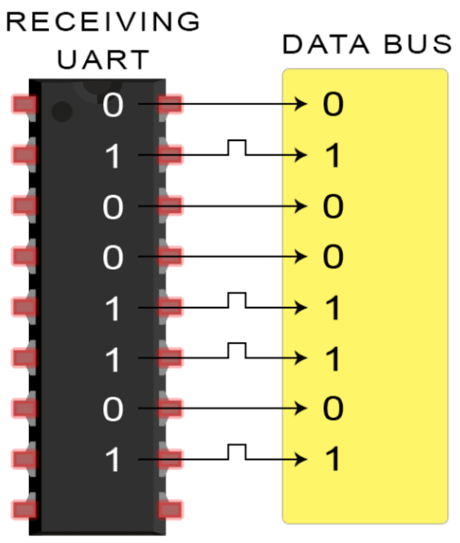
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:

4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:





5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



## ADVANTAGES AND DISADVANTAGES OF UARTS

No communication protocol is perfect, but UARTs are pretty good at what they do. Here are some pros and cons to help you decide whether or not they fit the needs of your project:

### ADVANTAGES

- Only uses two wires
- No clock signal is necessary
- Has a parity bit to allow for error checking
- The structure of the data packet can be changed as long as both sides are set up for it
- Well documented and widely used method

### DISADVANTAGES

- The size of the data frame is limited to a maximum of 9 bits
- Doesn’t support multiple slave or multiple master systems
- The baud rates of each UART must be within 10% of each other

UART or Universal Asynchronous Receiver Transmitter is a dedicated hardware associated with serial communication. The hardware for UART can be a circuit integrated on the microcontroller or a dedicated IC. This is contrast to SPI or I2C, which are just communication protocols.

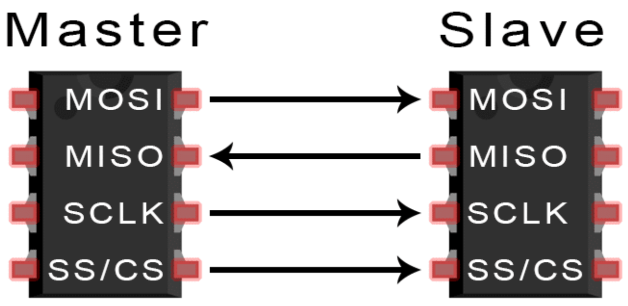
UART is one of the most simple and most commonly used Serial Communication techniques. Today, UART is being used in many applications like GPS Receivers, Bluetooth Modules, GSM and GPRS Modems, Wireless Communication Systems, RFID based applications etc.

## SPI COMMUNICATION PROTOCOL

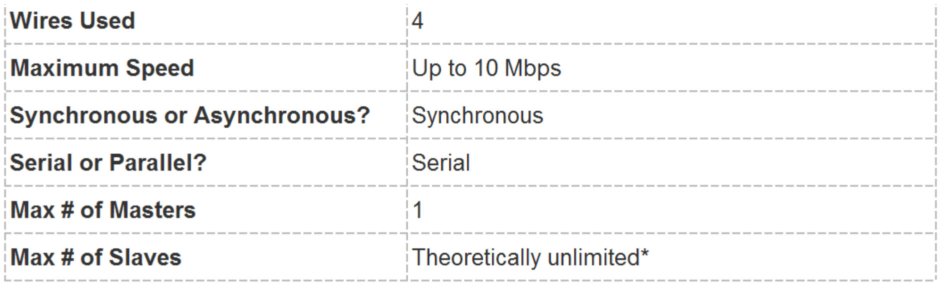
SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each packet, so the data is interrupted during transmission.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave (more on this below).



**MOSI (Master Output/Slave Input)** – Line for the master to send data to the slave.
**MISO (Master Input/Slave Output)** – Line for the slave to send data to the master
**SCLK (Clock)** – Line for the clock signal.
**SS/CS (Slave Select/Chip Select)** – Line for the master to select which slave to send data to.



| | |
|---|---|
| **Wires Used** | 4 |
| **Maximum Speed** | Up to 10 Mbps |
| **Synchronous or Asynchronous?** | Synchronous |
| **Serial or Parallel?** | Serial |
| **Max # of Masters** | 1 |
| **Max # of Slaves** | Theoretically unlimited* |

*In practice, the number of slaves is limited by the load capacitance of the system, which reduces the ability of the master to accurately switch between voltage levels.

## HOW SPI WORKS

### THE CLOCK

The clock signal synchronizes the output of data bits from the master to the sampling of bits by the slave. One bit of data is transferred in each clock cycle, so the speed of data transfer is determined by the frequency of the clock signal. SPI communication is always initiated by the master since the master configures and generates the clock signal.

Any communication protocol where devices share a clock signal is known as *synchronous.* SPI is a synchronous communication protocol. There are also *asynchronous* methods that don't use a clock signal. For example, in UART communication, both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.

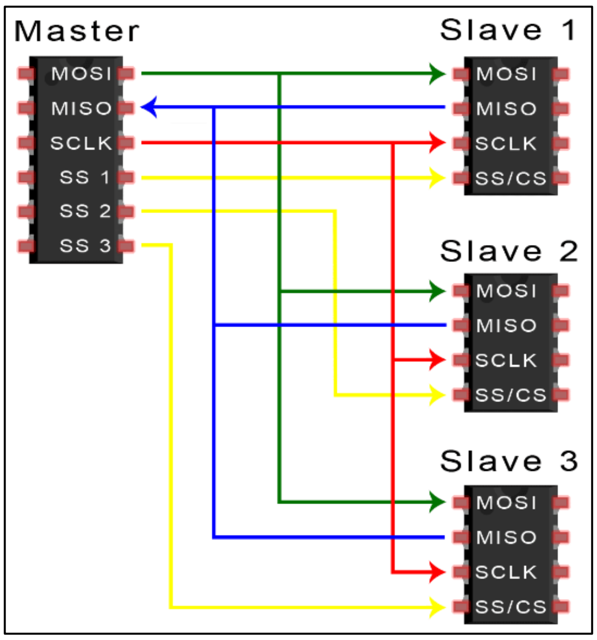
The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.

### SLAVE SELECT

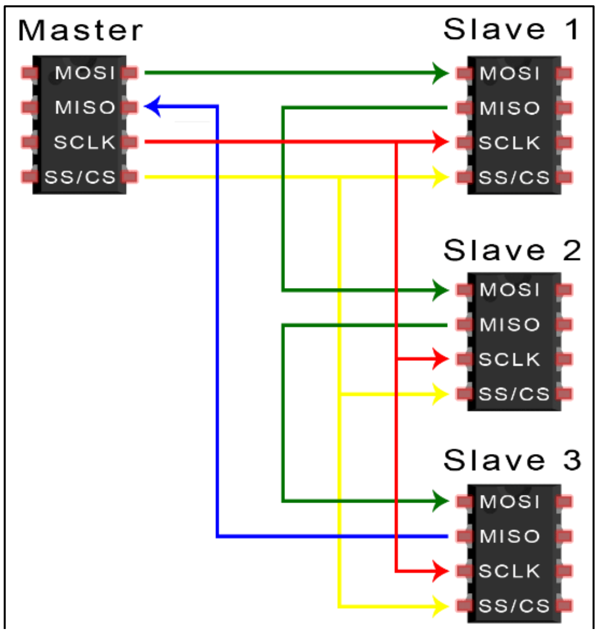
The master can choose which slave it wants to talk to by setting the slave's CS/SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

### MULTIPLE SLAVES

SPI can be set up to operate with a single master and a single slave, and it can be set up with multiple slaves controlled by a single master. There are two ways to connect multiple slaves to the master. If the master has multiple slave select pins, the slaves can be wired in parallel like this:



If only one slave select pin is available, the slaves can be daisy-chained like this:



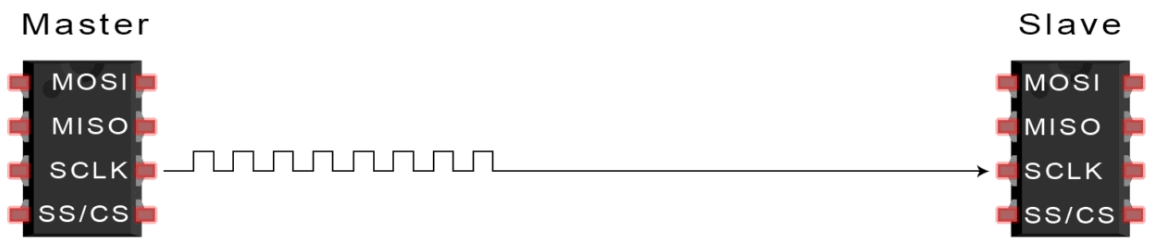
## MOSI AND MISO

The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.

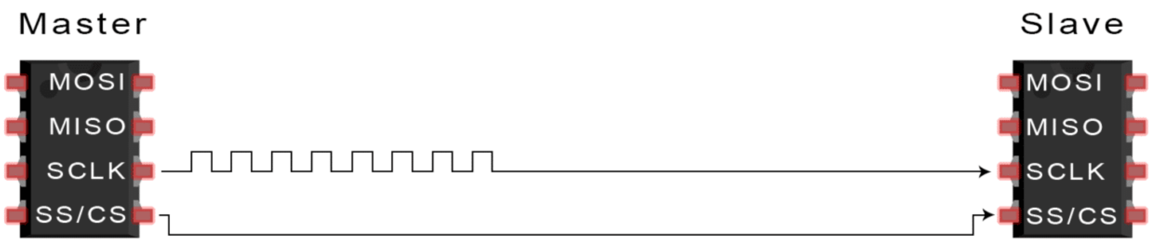
The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

## STEPS OF SPI DATA TRANSMISSION

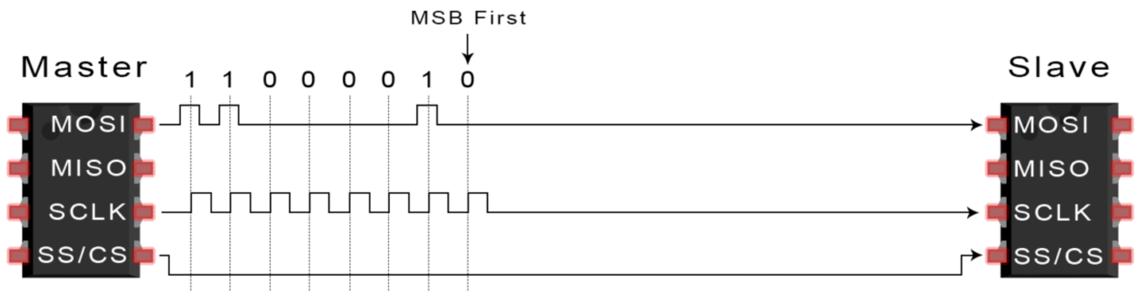
1. The master outputs the clock signal:



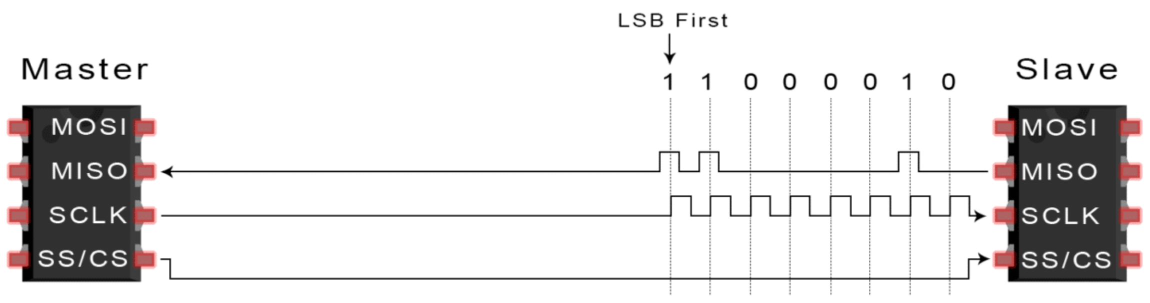
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:



## ADVANTAGES AND DISADVANTAGES OF SPI

There are some advantages and disadvantages to using SPI, and if given the choice between different communication protocols, you should know when to use SPI according to the requirements of your project:

### ADVANTAGES

- No start and stop bits, so the data can be streamed continuously without interruption
- No complicated slave addressing system like I2C
- Higher data transfer rate than I2C (almost twice as fast)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

### DISADVANTAGES

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master

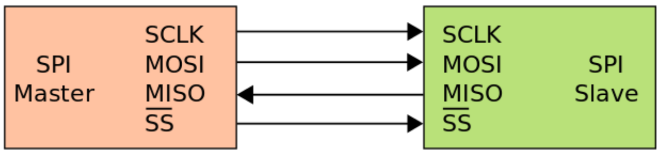
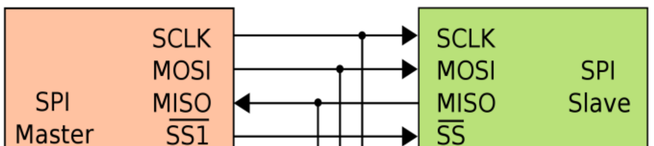


Fig: SPI Master connected to a single slave

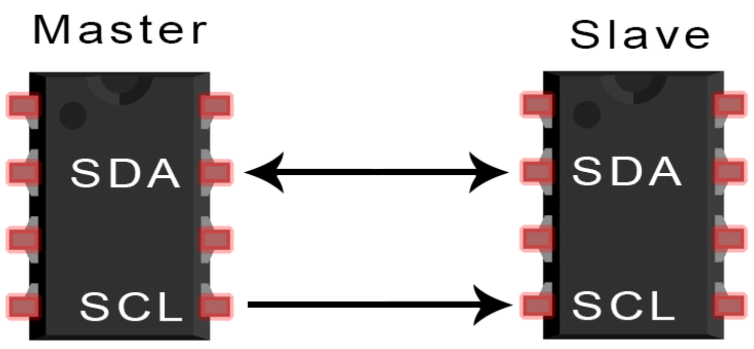


## I2C COMMUNICATION PROTOCOL

### INTRODUCTION TO I2C COMMUNICATION

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:



**SDA (Serial Data)** – The line for the master and slave to send and receive data.
**SCL (Serial Clock)** – The line that carries the clock signal.

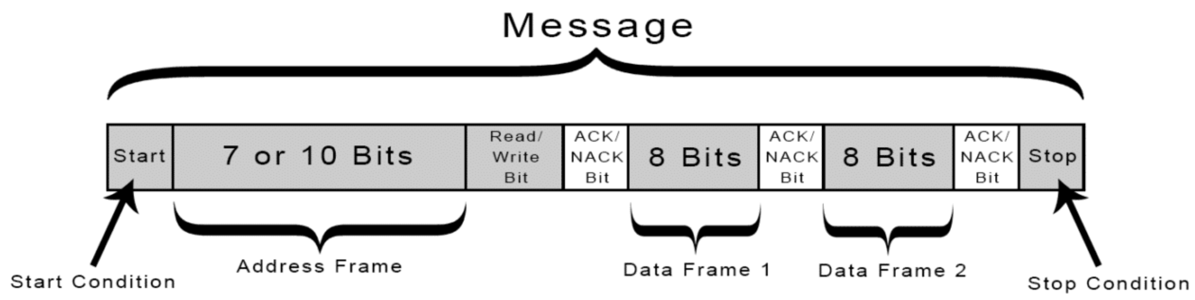
I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line).

Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

| | |
|---|---|
| **Wires Used** | 2 |
| **Maximum Speed** | Standard mode= 100 kbps |
| | Fast mode= 400 kbps |
| | High speed mode= 3.4 Mbps |
| | Ultra fast mode= 5 Mbps |
| **Synchronous or Asynchronous?** | Synchronous |
| **Serial or Parallel?** | Serial |
| **Max # of Masters** | Unlimited |
| **Max # of Slaves** | 1008 |

## HOW I2C WORKS

With I2C, data is transferred in *messages.* Messages are broken up into *frames* of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being



transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

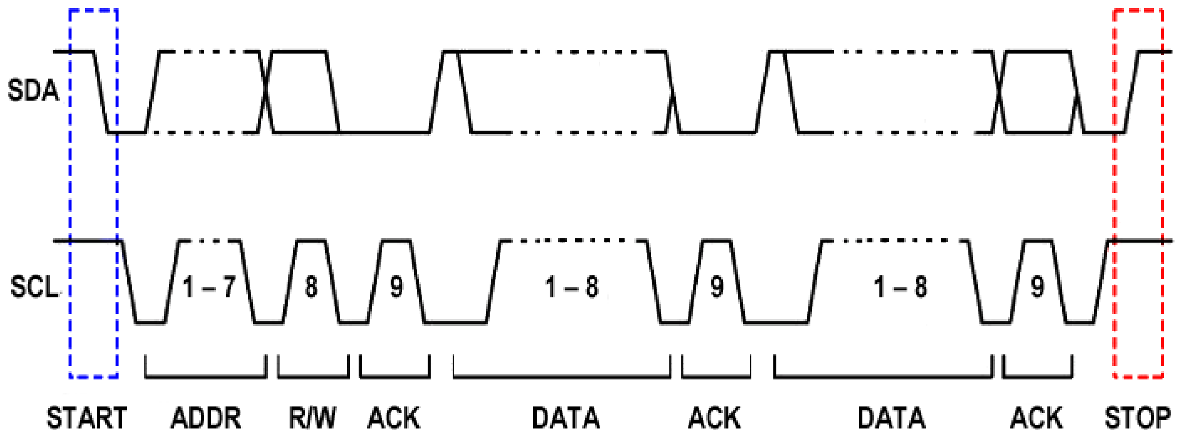
**Start Condition:** The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

**Stop Condition:** The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

**Address Frame:** A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

**Read/Write Bit:** A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

**ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.



## ADDRESSING

I2C doesn't have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by *addressing*. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn't match, the slave does nothing and the SDA line remains high.

## READ/WRITE BIT

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

## THE DATA FRAME

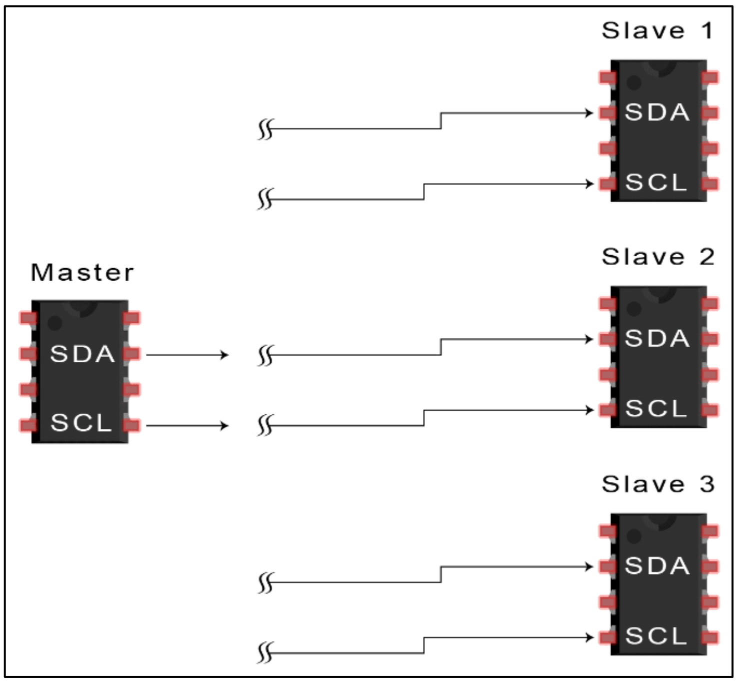
After the master detects the ACK bit from the slave, the first data frame is ready to be sent.

The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully. The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent.

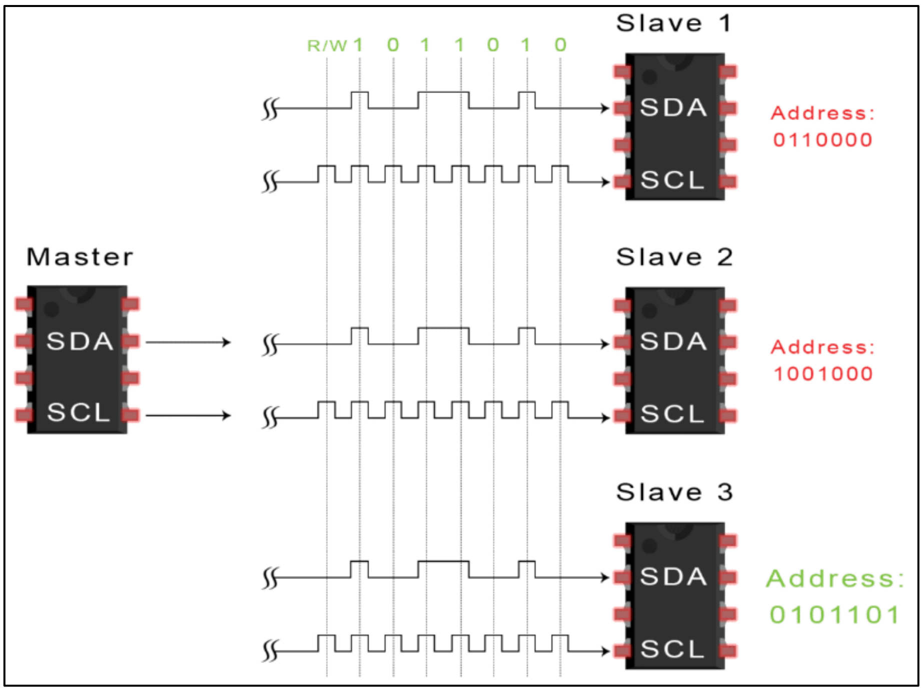
After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

## STEPS OF I2C DATA TRANSMISSION

1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low:

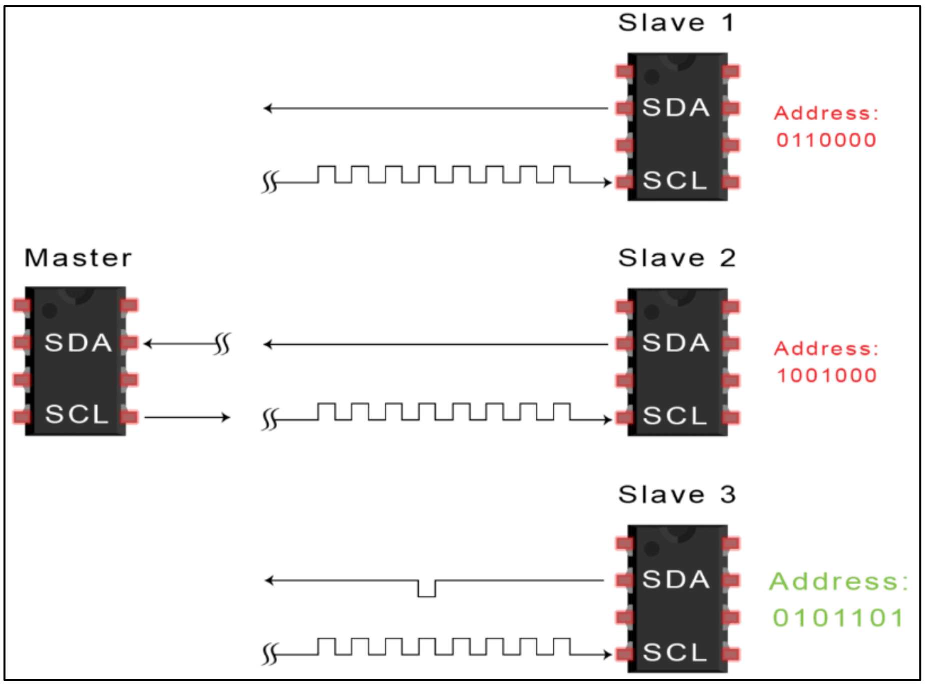


2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:

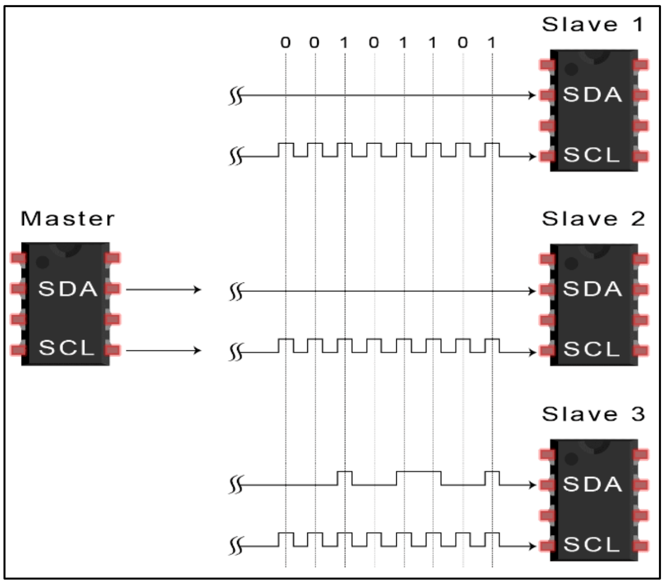


3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit.

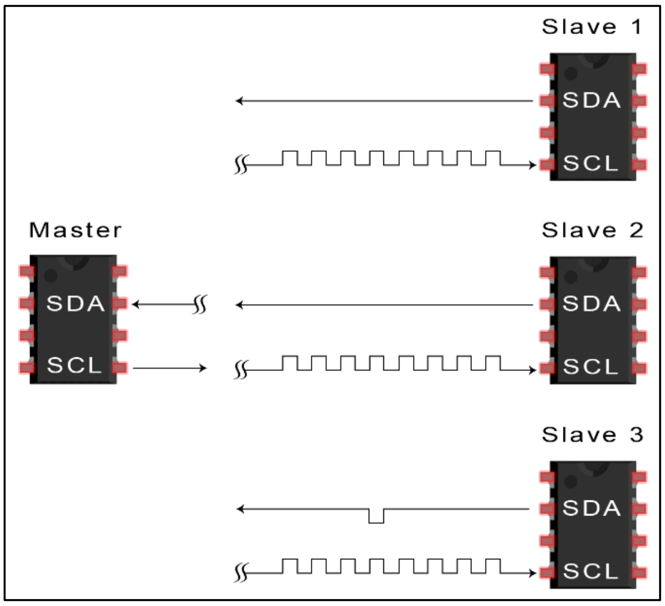
If the address from the master does not match the slave's own address, the slave leaves the SDA line high.



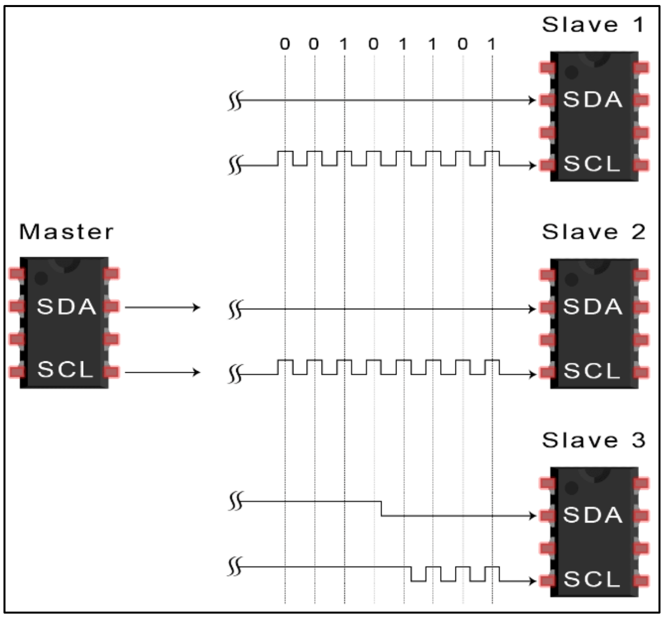
4. The master sends or receives the data frame:



5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

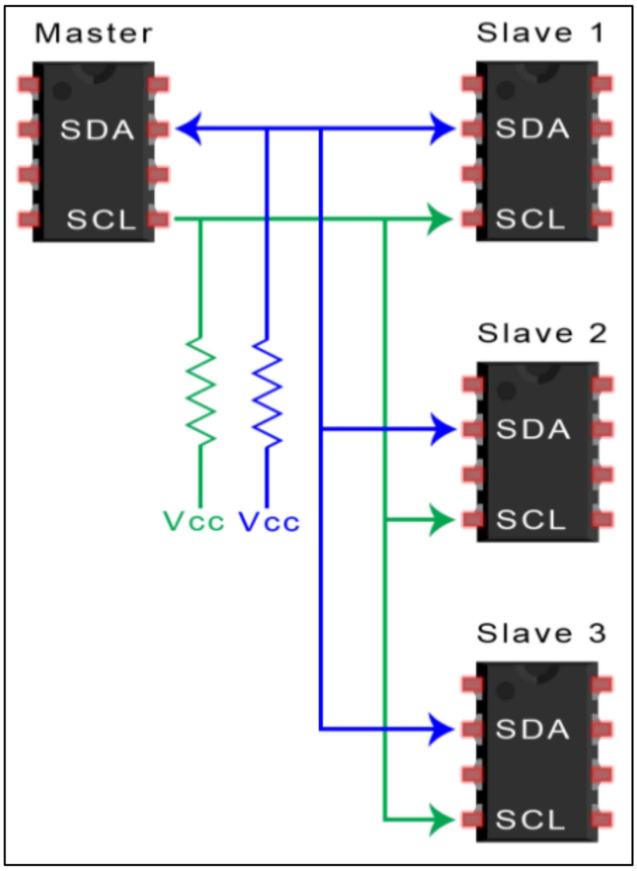


6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:



## SINGLE MASTER WITH MULTIPLE SLAVES

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7 bit address, 128 ($2^7$) unique address are available. Using 10 bit addresses is uncommon, but provides 1,024 ($2^{10}$) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K/10K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



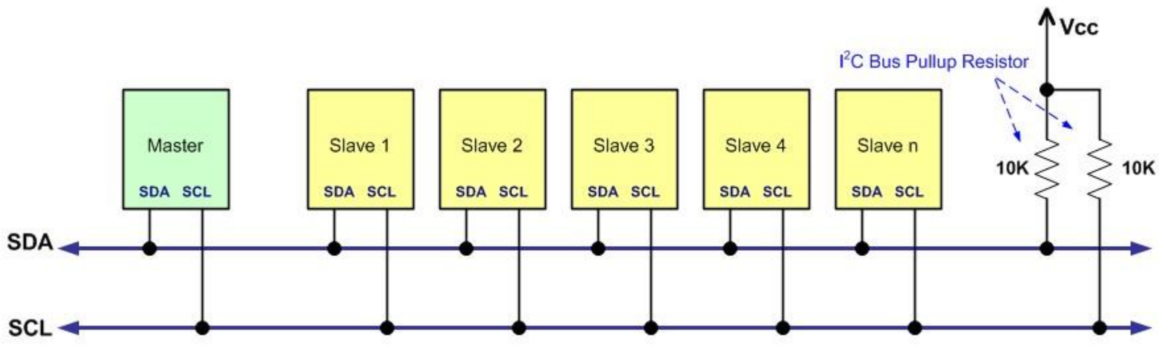
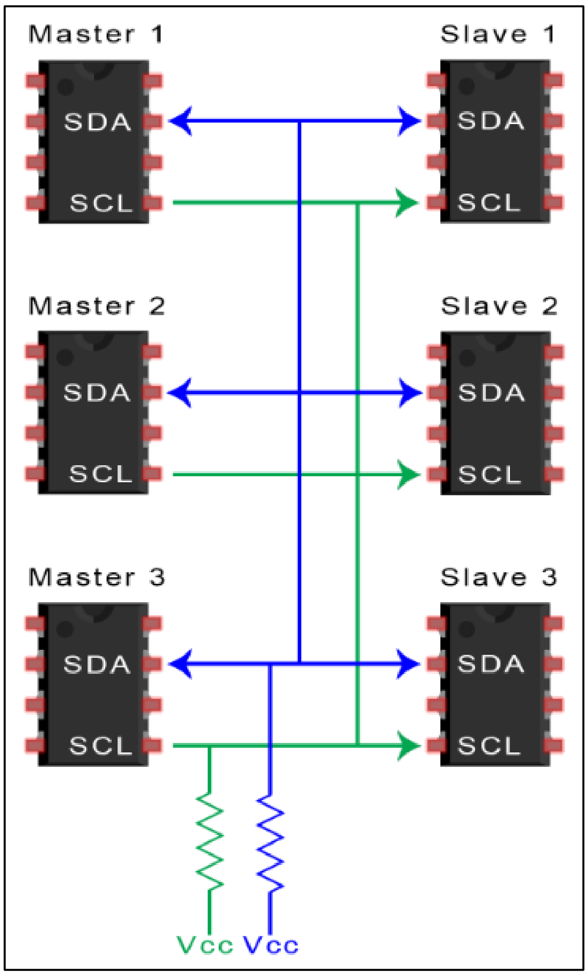


Figure: I2C Bus Connection

## MULTIPLE MASTERS WITH MULTIPLE SLAVES

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



## ADVANTAGES AND DISADVANTAGES OF I2C

There is a lot to I2C that might make it sound complicated compared to other protocols, but there are some good reasons why you may or may not want to use I2C to connect to a particular device:

## ADVANTAGES

- Only uses two wires
- Supports multiple masters and multiple slaves
- ACK/NACK bit gives confirmation that each frame is transferred successfully
- Hardware is less complicated than with UARTs
- Well known and widely used protocol

## DISADVANTAGES

- Slower data transfer rate than SPI
- The size of the data frame is limited to 8 bits
- More complicated hardware needed to implement than SPI

## UNIVERSAL SERIAL BUS (USB)

Universal Serial Bus (USB) is a set of interface specifications for high speed wired communication between electronics systems peripherals and devices with or without PC/computer. The USB was originally developed in 1995 by many of the industry leading companies like Intel, Compaq, Microsoft, Digital, IBM, and Northern Telecom.

The major goal of USB was to define an external expansion bus to add peripherals to a PC in easy and simple manner.

USB offers users simple connectivity. It eliminates the mix of different connectors for different devices like printers, keyboards, mice, and other peripherals. That means USB-bus allows many peripherals to be connected using a single standardized interface socket.It supports all kinds of data, from slow mouse inputs to digitized audio and compressed video.

USB also allows hot swapping. The "hot-swapping" means that the devices can be plugged and unplugged without rebooting the computer or turning off the device. That means, when plugged in, everything configures automatically.Once the user is finished, they can simply unplug the cable out; the host will detect its absence and automatically unload the driver. This makes the USB a plug-and-play interface between a computer and add-on devices.

USB is now the most used interface to connect devices like mouse, keyboards, PDAs, game-pads and joysticks, scanners, digital cameras, printers, personal media players, and flash drives to personal computers.

USB sends data in serial mode i.e. the parallel data is serialized before sends and de-serialized after receiving.

The benefits of USB are low cost, expandability, auto-configuration, hot-plugging and outstanding performance. It also provides power to the bus, enabling many peripherals to operate without the added need for an AC power adapter.

**Various versions USB:**

**USB1.0:** USB 1.0 is the original release of USB having the capability of transferring 12Mbps, supporting up to 127 devices. This USB 1.0 specification model was introduced in January 1996.

**USB1.1:** USB 1.1 came out in September 1998. USB 1.1 is also known as full-speed USB. This version is similar to the original release of USB; however, there are minor modifications for the hardware and the specifications. USB version 1.1 supported two speeds, a full speed mode of 12Mbits/s and a low speed mode of 1.5Mbits/s.

**USB2.0:** Hewlett-Packard, Intel, LSI Corporation, Microsoft, NEC, and Philips jointly led the initiative to develop a higher data transfer rate than the 1.1 specifications. The USB 2.0 specification was released in April 2000 and was standardized at the end of 2001.

Supporting three speed modes (1.5, 12 and 480 Mbps), USB 2.0 supports low-bandwidth devices such as keyboards and mice, as well as high-bandwidth ones like high-resolution Web-cams, scanners, printers and high-capacity storage systems.

USB 2.0, also known as hi-speed USB. This hi-speed USB is capable of supporting a transfer rate of up to 480 Mbps, compared to 12 Mbps of USB 1.1. That's about 40 times as fast! Wow!

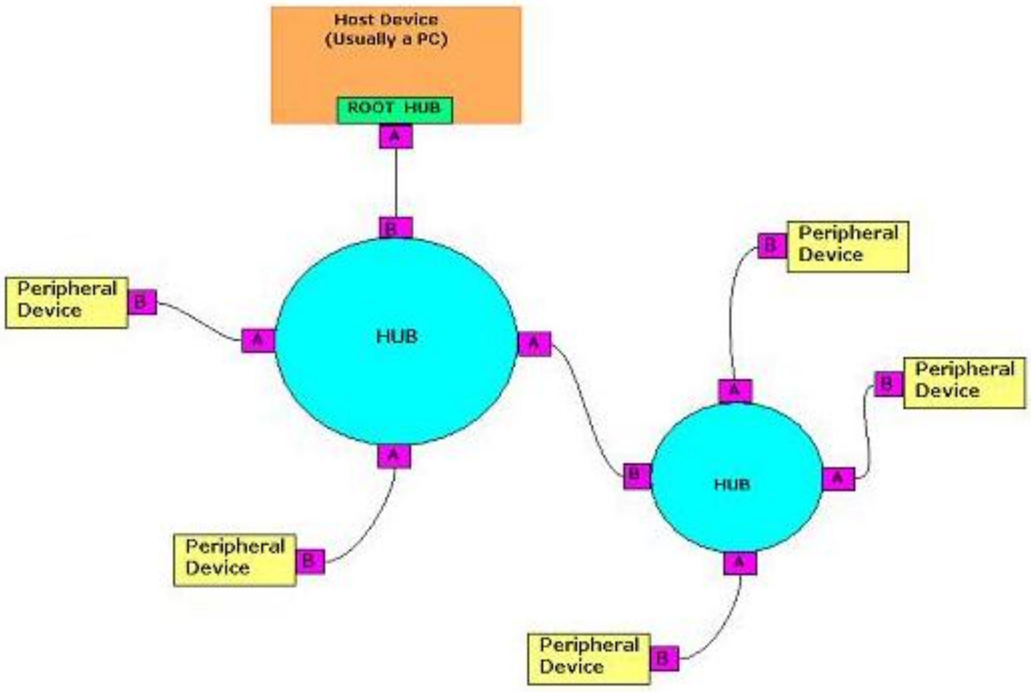
**USB3.0:** USB 3.0 is the latest version of USB release. It is also called as Super-Speed USB having a data transfer rate of 4.8Gbps (600 MB/s). That means it can deliver over 10x the speed of today's Hi-Speed USB connections.

The USB 3.0 specification was released by Intel and its partners in August 2008. Products using the 3.0 specifications are come out in 2010.

**The USB "tiered star" topology:**

The USB system is made up of a host, multiple numbers of USB ports, and multiple peripheral devices connected in a tiered-star topology.

The host is the USB system's master, and as such, controls and schedules all communications activities. Peripherals, the devices controlled by USB, are slaves responding to commands from the host. USB devices are linked in series through hubs.



There always exists one hub known as the root hub, which is built in to the host controller.

**Fig:** The USB "tiered star" topology

**USB connectors:**

Connecting a USB device to a computer is very simple -- you find the USB connector on the back of your machine and plug the USB connector into it. If it is a new device, the operating system auto-detects it and asks for the driver disk. If the device has already been installed, the computer activates it and starts talking to it.

The USB standard specifies two kinds of cables and connectors.

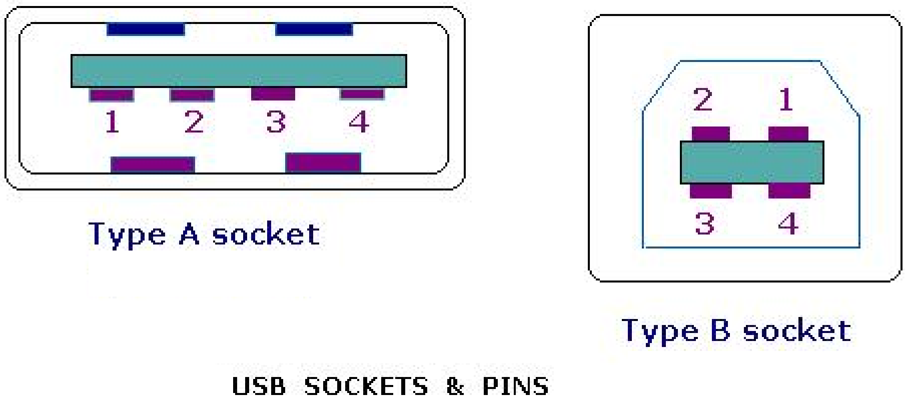


**Fig:** USB Type A & B Connectors

The USB standard uses "A" and "B" connectors mainly to avoid confusion:

1. "A" connectors head "upstream" toward the computer.

2. "B" connectors head "downstream" and connect to individual devices.

By using different connectors on the upstream and downstream end, it is impossible to install a cable incorrectly, because the two types are physically different.

| Pin No | Signal | Color of the cable |
|---|---|---|
| 1 | +5V power | Red |
| 2 | - Data | White / Yellow |
| 3 | +Data | Green / Blue |
| 4 | Ground | Black/Brown |

**Table:**USB pin connections

USB can support 4 data transfer types or transfer modes.

1. Control
2. Isochronous
3. Bulk
4. Interrupt

**Control transfers** exchange configuration, setup and command information between the device and host. The host can also send commands or query parameters with control packets.

**Isochronous transfer** is used by time critical, streaming device such as speakers and video cameras. It is time sensitive information so, within limitations, it has guaranteed access to the USB bus.

**Bulk transfer** is used by devices like printers & scanners, which receives data in one big packet.

**Interrupt transfer** is used by peripherals exchanging small amounts of data that need immediate attention.

All USB data is sent serially. USB data transfer is essentially in the form of packets of data, sent back and forth between the host and peripheral devices. Initially all packets are sent from the host, via the root hub and possibly more hubs, to devices.

**Each USB data transfer consists of a…**

1. Token packet (Header defining what it expects to follow)
2. Optional Data Packet (Containing the payload)
3. Status Packet (Used to acknowledge transactions and to provide a means of error correction).

***<u>Implementation of I2C Protocol Using MSP430 & EEPROM:</u>***

The schematic in Figure 1 shows how an EEPROM device can be connected to a MSP430 that has a hardware I2C module. On the F16x devices, it uses a USART module. On the F2xx devices, it uses an USCI module.

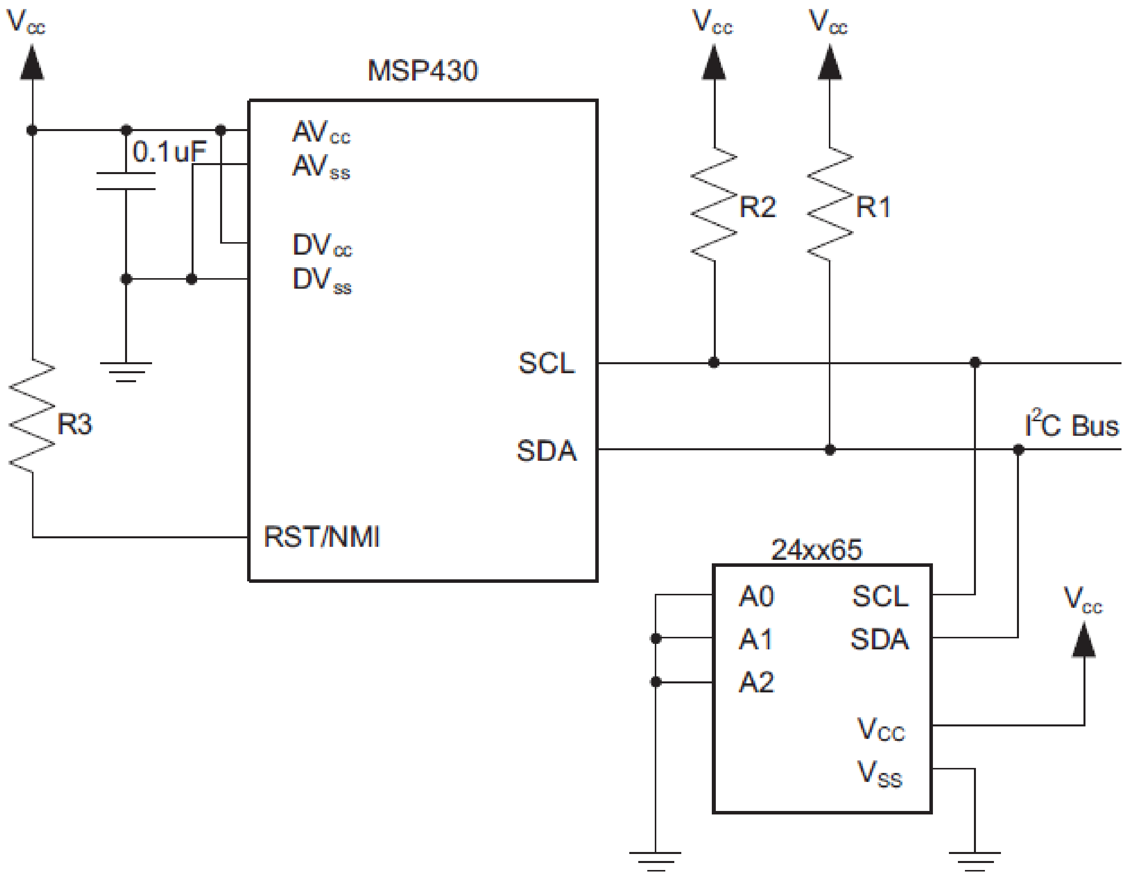


**Figure 1. Interfacing the 24xx128 EEPROM to the MSP430 via I$^2$C Bus**

On the 24xx128, the user configurable pins A0, A1, and A2 define the I2C device addresses of the connected EEPROMs. The inputs are used to allow multiple devices to operate on the same bus. The logic levels applied to these pins define the address block occupied by the device in the address map. A particular device is selected by transmitting the corresponding bits (A0, A1, and A2) in the control byte.

**MSP430 Source Code**

The software example shows how to use the MSP430 USCI module for communication with EEPROM via I2C bus. Depending on the memory size of the EEPROM the addressing scheme may look different. There are EEPROM versions that only need one byte for addressing (memory size of 256 bytes or less) and there are EEPROM versions that need two bytes. The example code uses two bytes for addressing.

**Table 1: Project Filenames and Subscriptions**

| Filename | Description |
|---|---|
| Main.c | Contains the main function that demonstrates various examples on how to use the available EEPROM functions |
| I2Croutines.c | EEPROM function library source file |
| I2Croutines.h | EEPROM function library header definitions file |

The $I^2C$ routines have the functions described in the following sections and can be used as a library.

### *1) InitI2C*

Declaration: void InitI2C (unsigned char eeprom_i2c_address);
Description: Initializes the MSP430 for I2C communication
Arguments: eeprom_i2c_address: Target EEPROM address to be initialized
Return: none

### *2) EEPROM_ByteWrite*

Declaration: void EEPROM_ByteWrite (unsigned int Address, unsigned char Data);
Description: A byte write command that writes a byte of data into the specified address that is provided.
Arguments: Address to write in the EEPROM Data Data to be written
Return: none

Figure 2 shows the Byte Write protocol.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| S | Contol Byte | R/W | ACK | Word Address 1 | ACK | Word Address 2 | ACK | Data | ACK | P |

**Figure 2. EEPROM "Byte Write" Command**

When using a 24xx128 EEPROM, the upper seven bits of the control byte is always structured in the following way: The upper 4 bits are a fixed number (1010) and the lower 3 bits are defined by the logical level connected to the pins A2, A1, and A0 of the EEPROM. For storing one byte of data in the EEPROM, four bytes have to be sent via I2C. The first byte is the control byte, which is followed by the EEPROM address. This is the address a byte to be stored in the EEPROM. Large EEPROM memory uses two bytes for defining the EEPROM address. The fourth and last byte is the actual data that is stored in the accessed EEPROM. The data is written into the EEPROM address that is transmitted as a part of the command format (see Figure 2).

### *3) EEPROM_PageWrite*

Declaration: void EEPROM_PageWrite(unsigned int StartAddress , unsigned char * Data , unsigned char Size);
Description: A Page Write command that writes a specified size of data from the data pointer into the specified address.
Arguments: StartAddress: Starting point address to start writing in the EEPROM
Data Pointer to data array to be written
Size: Size of data to be written
Return: none

***4) EEPROM_AckPolling***

Declaration: void EEPROM_AckPolling(void);

Description: The Acknowledge Polling is used to check if the write cycle of the EEPROM is complete. After writing data into theEEPROM, the Acknowledge Polling function should be called.

Arguments: none

Return: none

The EEPROM requires a finite time to complete a write cycle. This is typically defined in the EEPROM datasheet however it is possible that the write cycles may complete faster than the specified time. The function EEPROM_AckPolling() takes advantage of the fact that the EEPROM will not acknowledge its own address as long as it is busy finishing the write cycle. This function continues to send a control byte until it is acknowledged, meaning that the function will only return after the write cycle is complete.

***5) EEPROM_RandomRead***

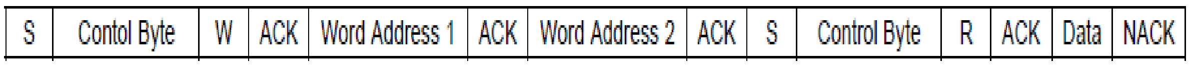
Declaration: unsigned char EEPROM_RandomRead(unsigned int Address);

Description: The Random Read command of the EEPROM allows reading the contents of a specified memory location.

Arguments: Address: Address to be read from the EEPROM

Return: Data byte from EEPROM

Figure 4 shows the Random Address read protocol. It uses master-transmit and master-receive operationwithout releasing the bus in between. This is achieved by using a repeated START condition.



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | Contol Byte | W | ACK | Word Address 1 | ACK | Word Address 2 | ACK | S | Control Byte | R | ACK | Data | NACK |

**Figure 4. EEPROM "Random Access Read" Command**

First the address counter of the EEPROM has to be set. This is done using a master-transmit operation.After sending the address, the MSP430 is configured for master-receive operation and initiates data readby sending a repeated START condition.

***6) EEPROM_CurrentAddressRead:***

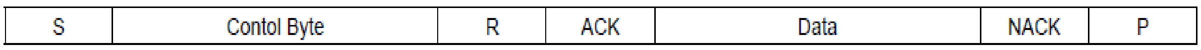
Declaration: unsigned char EEPROM_CurrentAddressRead(void);

Description: The EEPROM internal address pointer is used. After execution of a write or read operation the internal addresspointer is automatically incremented.

Arguments: none

Return: Data byte from EEPROM

Figure 5 shows the current address read operation. The MSP430 is configured as a master –receivebefore executing this command. Before the communication is started by the MSP430 the I2C module isconfigured to master-receive mode.



| | | | | | | |
|---|---|---|---|---|---|---|
| S | Contol Byte | R | ACK | Data | NACK | P |

**Figure 5. EEPROM "Current Address Read" Command**

After the STOP condition has occurred the received data is returned to the caller of the function.

***7) EEPROM_SequentialRead***

Declaration: void EEPROM_SequentialRead(unsigned int Address , unsigned char * Data , unsigned int Size);

Description: The sequential read is used to read a sequence of data specified by the size and starting from the knownaddress. The data pointer points to where the data is to be stored.

Arguments: StartAddress: Starting point address to start reading from the EEPROM

Data Pointer to data array to be stored

Size: Size of data to be read

Return: none

Figure 6 shows the sequential read protocol implementation. The MSP430 is configured as a mastertransmitter and sends out the control byte followed by the address. After that, a re-start is issued with theMSP430 configured as a master receiver. When the last character is read, a stop command is issued.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | Control Byte | W | ACK | Word Address 1 | ACK | Word Address 2 | ACK | S | Control Byte | R | ACK | Data Byte 0 | ACK | ... | Data Byte 63 | ACK | P |

**Figure 6. EEPROM "Sequential Read" Command**

**<u>I2C Interface Program:</u>**

The following example shows how to use the functions from the files "I2Croutines.c":

```
#include "msp430.h"
#include "I2Croutines.h"
unsigned char read_val[150];
unsigned char write_val[150];
unsigned int address;
int main(void)
{
unsigned int i;
WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
InitI2C();                // Initialize I2C module
EEPROM_ByteWrite(0x0000,0x12);
EEPROM_AckPolling(); // Wait for EEPROM write cycle
// completion
EEPROM_ByteWrite(0x0001,0x34);
EEPROM_AckPolling(); // Wait for EEPROM write cycle
// completion
EEPROM_ByteWrite(0x0002,0x56);
EEPROM_AckPolling(); // Wait for EEPROM write cycle
// completion
EEPROM_ByteWrite(0x0003,0x78);
EEPROM_AckPolling(); // Wait for EEPROM write cycle
// completion
EEPROM_ByteWrite(0x0004,0x9A);
EEPROM_AckPolling(); // Wait for EEPROM write cycle
// completion
EEPROM_ByteWrite(0x0005,0xBC);
EEPROM_AckPolling(); // Wait for EEPROM write cycle
// completion
read_val[0] = EEPROM_RandomRead(0x0000); // Read from address//0x0000
read_val[1] = EEPROM_CurrentAddressRead();// Read from address //0x0001
read_val[2] = EEPROM_CurrentAddressRead();// Read from address //0x0002
read_val[3] = EEPROM_CurrentAddressRead();// Read from address //0x0003
read_val[4] = EEPROM_CurrentAddressRead();// Read from address //0x0004
read_val[5] = EEPROM_CurrentAddressRead();// Read from address //0x0005
// Fill write_val array with counter values
for(i = 0 ; i <= sizeof(write_val) ; i++)
{
write_val[i] = i;
}
address = 0x0000; // Set starting address at 0
// Write a sequence of data array
EEPROM_PageWrite(address , write_val , sizeof(write_val));
// Read out a sequence of data from EEPROM
EEPROM_SequentialRead(address, read_val , sizeof(read_val));
__bis_SR_register(LPM4);
__no_operation();
}
```

## Implementation of UART Using MSP430 & PC:

Serial communication can be divided into two categories:

- Synchronous serial communication
- Asynchronous serial communication

Synchronous serial communication uses a dedicated communication channel to send a clock signal which provides the timing between the transmitter and receiver. While in asynchronous communication, the timing is encoded within the signal. There are several versions of both synchronous and asynchronous serial communication. The MSP430G2553 supports 3 different types of communications using its USCI peripheral: Serial Peripheral Interface (SPI), Inter-Integrated Circuit Interface (I2C) and UART.

The MSP430G2553 is an ultra-low power device and has one active mode and five software selectable low-power modes of operation. An interrupt event can wake up the device from any of the low-power modes, service the request, and restore back to the low-power mode on return from the interrupt program.

In UART mode, the USCI transmits and receives characters at a bit rate asynchronous to the paired device. Timing for the transmission of each character is based on the selected baud rate of the USCI. The transmit and receive functions use the same baud rate frequency.

The block diagram for the experiment is as shown in **Figure.** The UART baud rate is set to 9600. The UART specifies two pins, a TX pin (located at P1.2) and an RX pin (P1.1) for communication. The USCI-UART module is configured to transmit a preset message on an interrupt. An interrupt provided within the program code transmits the message to be displayed on the UART terminal of the computer.

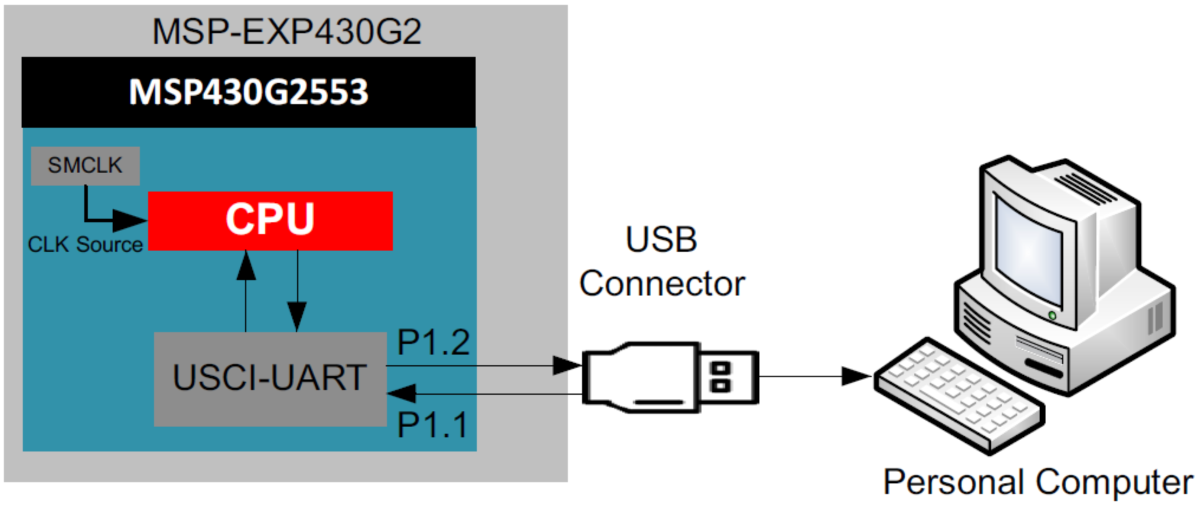


**Figure: Functional Block Diagram (UART Communication b/w PC and MSP430)**

In our program, we configure our device to stay in Low power mode 1 (LPM1) for power optimiza- tion until it is woken up by the interrupt. The features of Low power mode 1 (LPM1):

- CPU is disabled
- ACLK and SMCLK remain active, MCLK is disabled
- DCO's DC generator is disabled if DCO not used in active mode

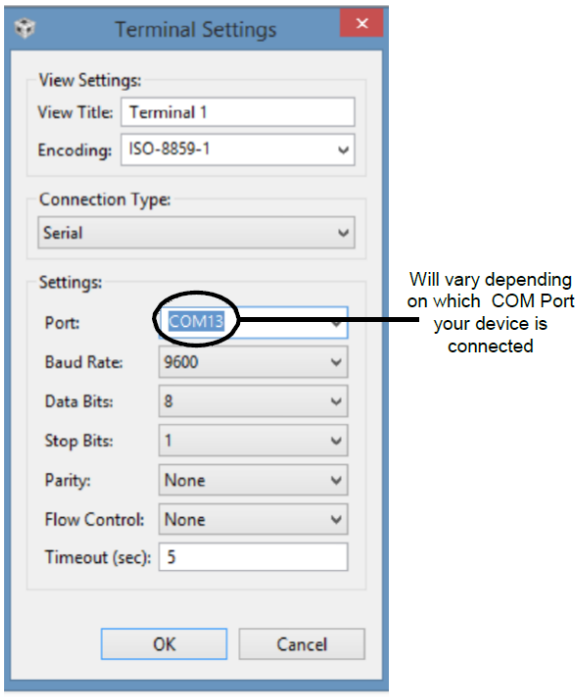
***Hardware and Software Requirement***

1. MSP430G2 Launchpad
2. USB cable
3. Personal Computer
4. CCS (Code Composer Studio) Software

***UART Terminal Setup***

**1.** In CCS window select **Viewother**and from the options given select **Terminal.**

**2.** When the terminal window opens, click on the **Settings** icon and set the values as shown in**Figure 9-2**.



Terminal Settings

View Settings:
View Title: Terminal 1
Encoding: ISO-8859-1

Connection Type:
Serial

Settings:
Port: COM13
Baud Rate: 9600
Data Bits: 8
Stop Bits: 1
Parity: None
Flow Control: None
Timeout (sec): 5

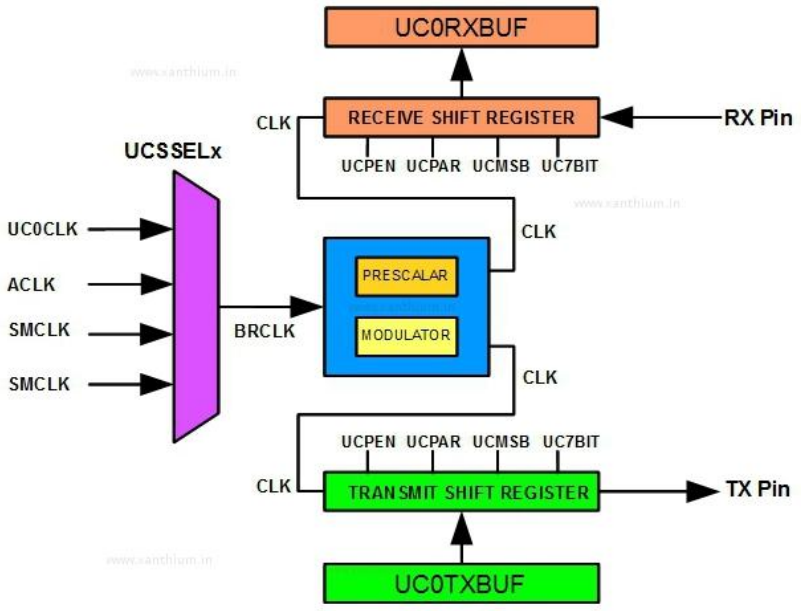
OK Cancel

Will vary depending on which COM Port your device is connected

Figure 9-2 Terminal Window Settings

This Terminal will allow you to view the serial information received by the computer and will also allow you to type in the information that you wish to transmit to the MSP430G2553 via the UART.

**UART Block Diagram**



***Registers Used:***
The registers used are:

• **WDTCTL: 16-bit Watchdog Timer Control Register** - The Watchdog Timer (WDT) is typically used to trigger a system reset after a certain amount of time. In this program it is stopped.

• **IE2: Interrupt Enable Register 2 -** This register enables the USCI transmit and receive interrupts.Here, the USCI interrupts are enabled.

• **P1SEL and P1SEL2: Function Select Register -** Each bit in the function select register is used to select the pin function - I/O port or peripheral module function. Here, the UART function on Pins P1.1 and P1.2 are enabled.

• **UCA0CTL0: USCI_A0 Control Register 0 -** This register configures the USCI module. Here the configuration is: No parity, LSB first, 8-bit data, 1 stop bit, UART, Asynchronous.

• **UCA0CTL1: USCI_A0 Control Register 1** - This register is used to configure clock source and enable software interrupts. Here, SMCLK is used as clock source.

• **UCA0BR0 & UCA0BR1: USCI_A0 Baud Rate Control Register 0 and 1** - These 16-bit registers are configured the clock prescale setting of the baud rate generator. Here, the registers are configured to have 1MHz frequency and 9600 baud rate.

• **UCA0MCTL: USCI_A0 Modulation Control Register** - This register selects the modulation stages which determine the modulation pattern for BITCLK. Here, the 2nd Stage modulation = 1, and oversampling is turned off.

• **IFG2: Interrupt Flag Register 2** - This register consists of transmit and receive interrupt flags. Here, the flags are cleared.

***USCI Control registers for UART configuration:***

| | | | | | | |
|---|---|---|---|---|---|---|
| ***UCPEN*** | ***UCPAR*** | ***UCMSB*** | ***UC7BIT*** | ***UCSPB*** | ***UCMODEx*** | ***UCSYNC*** |

**UCAxCTL0 Register abbreviated as** USCI_Ax Control Register 0

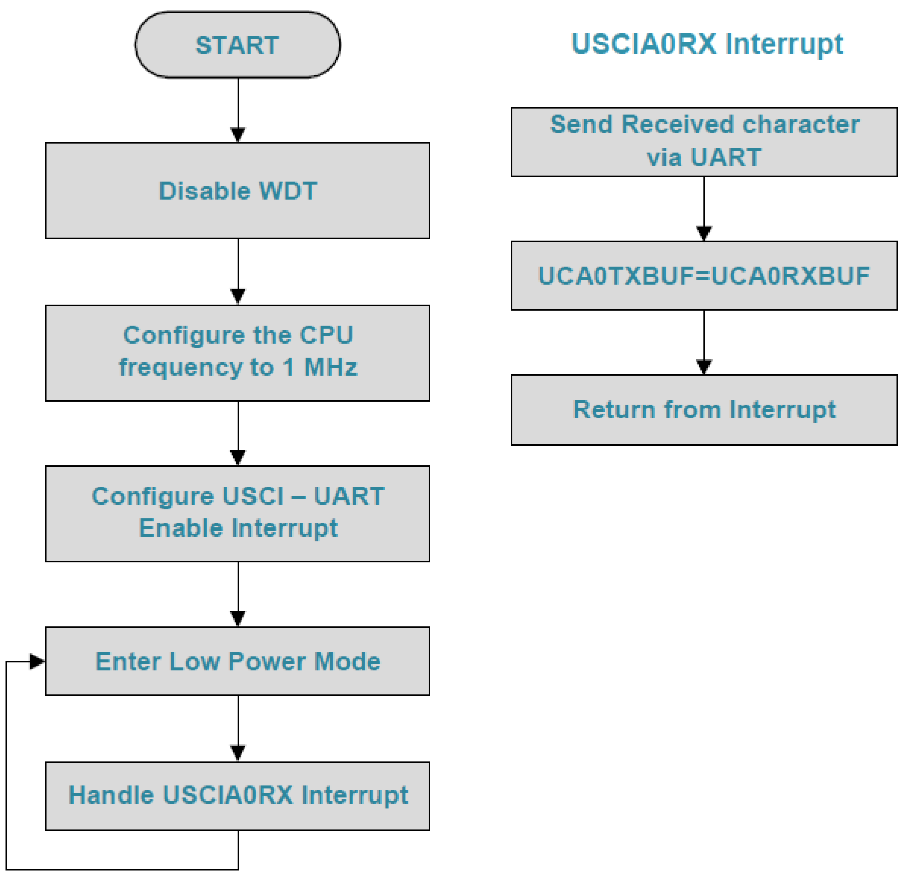
| | | | | | | |
|---|---|---|---|---|---|---|
| ***UCSSELx*** | ***UCRXEIE*** | ***UCBRKIE*** | ***UCDORM*** | ***UCTXADDR*** | ***UCTXBRK*** | ***UCSWRST*** |

**UCAxCTL1 Register abbreviated as** USCI_Ax Control Register 1

| **Bit** | **Name** | **Type** | **Reset** | **Description (UCAxCTL0 Register )** |
|---|---|---|---|---|
| 7 | UCPEN | RW | 0h | Parity enable<br>0b = Parity disabled<br>1b = Parity enabled |
| 6 | UCPAR | RW | 0h | Parity select. UCPAR is not used when parity is disabled.<br>0b = Odd parity<br>1b = Even parity |
| 5 | UCMSB | RW | 0h | MSB first select. Controls the direction of receive and transmit shift register.<br>0b = LSB first; 1b = MSB first |
| 4 | UC7BIT | RW | 0h | Character length. Selects 7-bit or 8-bit character length.<br>0b = 8-bit data; 1b = 7-bit data; |
| 3 | UCSPB | RW | 0h | Stop bit select. Number of stop bits.<br>0b = One stop bit; 1b = Two stop bits |
| 2-1 | UCMODEx | RW | 0h | USCI mode. The UCMODEx bits select the asynchronous mode when UCSYNC = 0.<br>00b = UART mode<br>01b = Idle-line multiprocessor mode<br>10b = Address-bit multiprocessor mode<br>11b = UART mode with automatic baud-rate detection |
| 0 | UCSYNC | RW | 0h | Synchronous mode enable<br>0b = Asynchronous mode<br>1b = Synchronous mode |

| Bit | Field | Type | Reset | Description (UCAxCTL1 Register) |
|---|---|---|---|---|
| 7-6 | UCSSELx | RW | 0h | USCI clock source select. These bits select the BRCLK source clock.<br>00b = UCAxCLK (external USCI clock)<br>01b = ACLK<br>10b = SMCLK<br>11b = SMCLK |
| 5 | UCRXEIE | RW | 0h | Receive erroneous-character interrupt enable<br>0b = Erroneous characters rejected and UCRXIFG is not set.<br>1b = Erroneous characters received set UCRXIFG. |
| 4 | UCBRKIE | RW | 0h | Receive break character interrupt enable<br>0b = Received break characters do not set UCRXIFG.<br>1b = Received break characters set UCRXIFG. |
| 3 | UCDORM | RW | 0h | Dormant. Puts USCI into sleep mode.<br>0b = Not dormant. All received characters set UCRXIFG.<br>1b = Dormant. |
| 2 | UCTXADDR | RW | 0h | Transmit address. Next frame to be transmitted is marked as address, depending on the selected multiprocessor mode.<br>0b = Next frame transmitted is data.<br>1b = Next frame transmitted is an address. |
| 1 | UCTXBRK | RW | 0h | Transmit break. Transmits a break with the next write to the transmit buffer.<br>0b = Next frame transmitted is not a break.<br>1b = Next frame transmitted is a break or a break/synch. |
| 0 | UCSWRST | RW | 1h | Software reset enable<br>0b = Disabled. USCI reset released for operation.<br>1b = Enabled. USCI logic held in reset state. |

***Flowchart:***


START
Disable WDT
Configure the CPU frequency to 1 MHz
Configure USCI – UART Enable Interrupt
Enter Low Power Mode
Handle USCIA0RX Interrupt
USCIA0RX Interrupt
Send Received character via UART
UCA0TXBUF=UCA0RXBUF
Return from Interrupt

**C**

Figure 9-3 Flowchart for Serial Communication Using UART

**Program Code for Serial Communication Using UART**

```
#include <msp430.h>
int main(void)
{
   WDTCTL = WDTPW + WDTHOLD; // Stop WDT
/* Use Calibration values for 1MHz Clock DCO*/
   DCOCTL = 0;
   BCSCTL1 = CALBC1_1MHZ;
   DCOCTL = CALDCO_1MHZ;
/* Configure Pin P1.1 = RXD and P1.2 = TXD */
   P1SEL = BIT1 | BIT2 ;
   P1SEL2 = BIT1 | BIT2;
/* Place UCA0 in Reset to be configured */
   UCA0CTL1 = UCSWRST;
   /* Configure */
   UCA0CTL1 |= UCSSEL_2; // SMCLK
   UCA0BR0 = 104; // 1MHz 9600
   UCA0BR1 = 0; // 1MHz 9600
   UCA0MCTL = UCBRS0; // Modulation UCBRSx = 1
/* Take UCA0 out of reset */
```

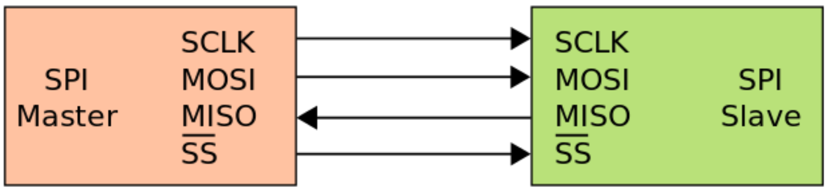
```
  UCA0CTL1 &= ~UCSWRST;
/* Enable USCI_A0 RX interrupt */
  IE2 |= UCA0RXIE;
  __bis_SR_register(LPM0_bits+GIE);//Enter LPM0 with interrupts
}
/*Echo back RXed character, confirm TX buffer is ready first*/
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
  while (!(IFG2&UCA0TXIFG)          //USCI_A0 TX buffer ready?
  UCA0TXBUF = UCA0RXBUF;            // TX RXed character
}
```

**Observation:**

After compiling and running the program the Serial Terminal displays - EchoOutput: As the characters are typed in, they will be displayed on the terminal. The input entered from the keyboard is echoed back and shown on the terminal window. The keyboard entries in Tera-Term are generally streamed only to the serial port and not to the display. Only the characters received via the serial port are displayed on this window.

## Implementation of SPI Using MSP430

A SPI connection between a Master and a Slave device is shown below:



SPI uses the concept of a master device and multiple slave devices. Let’s look at some of the signals:

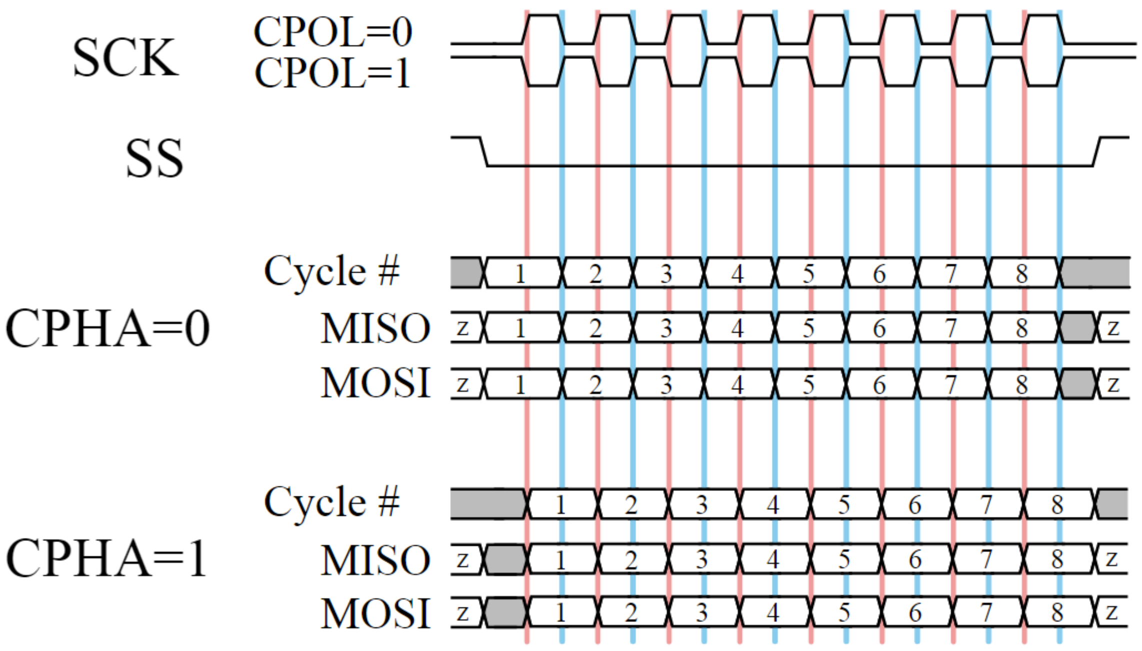
- SCLK – Main clock synchronizing SPI on both devices. Generated by the Master to all the slaves
- MOSI – Master Output and Slave Input in which data is sent from the master to the slave on each clock edge
- MISO – Master Input and Slave Output in which data is sent from the slave to the master on each clock edge
- SS – Slave select, often called CS (Chip Select) or CSn (Chip Select Active Low). This line selects the current active slave

SPI is a full duplex protocol in which the slave and the master exchange data at each transition of the clock by setting MOSI and MISO to a bit value. Note that MOSI and MISO are descriptive name for the connection. Often times they are called SIMO and SOMI or Slave Output (SO) and Slave Input (SI). It’s very important to avoid mixing the two signals by crossing them. MOSI should connect to MOSI, MISO to MISO.

Because it uses a bus topology, the master can talk to multiple devices, although one at a time. The multiplexing of selecting the different devices is accomplished by using a separate Slave Select line for each device. In some cases where only one slave is present, using a Slave Select might not be required by tying SS on the slave to ground. Some slaves, however, need a falling edge transition to operate properly. You should read the datasheet of the slave device carefully to ensure the waveform generated by the SPI master complies with the requirements of the slave. Slave Select line is typically active low which means it remains high when not sending data to the slave. The SS line is then pulled low for the duration of the transaction, framing it.

| Mode | CPOL | CPHA |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

We mentioned above that the master and slave exchange data bits which are captured at a clock transition, but we didn't specify whether it was on the rising or falling edge of the clock. We also didn't mention whether the clock is active low or active high. This is because these are configurable and a slave can use any of the 4 SPI modes. The figure below shows the configuration of the Polarity (POL) and Phase (PH) bits in the MSP430 SPI modules.



It's important that the right phase and polarity are selected so that the slave and master both capture the bit at the correct clock transition. If this configuration is not correct, you will see garbage coming out, and the slave will not properly receive the data sent by the MSP430.

## Configuring the MSP430 for SPI

The MSP430G2553 has two SPI interfaces, mapped with the following pins:

**UCA0**

P1.1 – UCA0SOMI

P1.2 – UCA0SIMO

P1.4 – UCA0CLK

**UCB0**

P1.6 – UCB0SOMI

P1.7 – UCB0SIMO

P1.5 – UCB0CLK

The code above also uses P1.5 as CSn. Why haven't we used STE? Because STE is not slave select. The Slave Select operation on the MSP430 must be performed using GPIOs. The STE pin, on the other hand, is used when the MSP430 operates as a slave and allows master arbitration. It will not act as a slave select. To enable SPI we need to make the pins for MISO, MOSI and CLK in peripheral mode so USCI will control them. We then configure a custom pin such as P1.5 above, to be the SS or CSn. Looking at the port schematics in the G2553 datasheet we can see that to make the pins work with USCI we need to set Both P1SEL and P1SEL2 to '1'.

You might have noticed that we change P1OUT before PDIR, which might seem a bit unnecessary. After all, the output doesn't take effect unless the direction is Output. The reason for this is that if we were to first set the direction, the output register might have been set to '0'. In the time between the output register being 0 and being set to 1, we will cause an undesirable and momentary glitch on the line. The line might go From High Impedance (since by default the pins are in high impedance), to '0' and then to '1'. The change from '0' to '1' is mostly harmless, but can sometimes cause issues with devices.

Note that for the cases when we want to use both SPI interfaces, we would need to change CSn from P1.5 to another pin so that P1.5 can be used for UCB0CLK.

With the pin out configured, it's time to configure the USCI module for operation. As when we configured USCI for UART, we first place the module in reset. We then configure several bits of the UCA0CTL0 register. The code sets the SPI controller to be a Master sending MSB first, which is set by UCMSB and UCMST. UCSYNC selects SPI mode for the module since it supports various protocols and we want to select SPI specifically. UCCKPH enables the right SPI mode for the CC110x transceivers and we use it only as an example. You will need to adjust this for your own device.

As with UART, SPI mode also needs an input clock. The code above selects SMCLK as the source clock for the module with the UCSSEL_2 bit. Just after selecting the clock source, the clock divider is set to device the input clock (SMCLK in this case) by 2. Note that the USCI module must have an input clock divided by at least 2. So, if SMCLK is running at 16MHz, our SPI will run at 8MHz. With the clocks and module configured, we can take the module out of reset and it will be ready to run. Because feeds the master clock to the SPI slave, both master and slave are synchronized which means clock accuracy is not critical. Because of this we don't have to worry about accurate clocks when using SPI. SPI can theoretically operate at any frequency, but the MSP430 and slave limit the maximum frequency that can be used.

Once out of reset, the SPI peripheral is ready. Let's see how we transmit a byte. Note that this is a blocking implementation. In practice you will want to use interrupts which will enable the MSP430 to do other things while the SPI module is operating.

The first thing to do is to select the SPI slave by putting P1.5 low using the P1OUT register. After this we need to check whether the TXIFG flag is set. As with UART, this flag is set when the TX buffer is ready to be filled with data. In our case we know that no other operation has been performed on the SPI module, so there is no reason to check and this is a formality. But it is important to build code that works well in all cases. Note that the while loop waits while the flag is 0 and will exit immediately once the flag is set. As soon as it is, we can send a byte by filling the TXBUF. As soon as TXBUF is filled, the SPI will generate SCLK and transfer the bits. But, how do we know when the transmission is done? We should never interrupt a slave by raising the slave select during a transmission since most devices will interpret that as an aborted transmission and will discard the bits. If you remember, SPI is a duplex protocol which means that the slave also sends back data. In most cases this data is 0x00 or 0xFF (garbage), but we can use the RXIFG flag to wait until the transmission and the reception on the master side is over. After we receive the data, we store the byte received in a variable and then terminate communications by raising P1.5 High again.

**SPI Interfacing Program**

```
#include <msp430.h>
volatile char received_ch = 0;
int main(void)
{
   WDTCTL = WDTPW + WDTHOLD; // Stop WDT

   P1OUT |= BIT5;
   P1DIR |= BIT5;
   P1SEL = BIT1 | BIT2 | BIT4;
   P1SEL2 = BIT1 | BIT2 | BIT4;
```

```
    UCA0CTL1 = UCSWRST
    UCA0CTL0 |= UCCKPH + UCMSB + UCMST + UCSYNC; //3-pin, 8-bit SPI master
    UCA0CTL1 |= UCSSEL_2; // SMCLK
    UCA0BR0 |= 0x02; // /2 (Division factor)
    UCA0BR1 = 0;
            UCA0MCTL = 0; // No modulation
    UCA0CTL1 &= ~UCSWRST; //Initialize USCI state machine
    P1OUT &= (~BIT5); // Select Device
while (!(IFG2 & UCA0TXIFG)); //USCI_A0 TX buffer ready?
    UCA0TXBUF = 0xAA; // Send 0xAA over SPI to Slave
    while (!(IFG2 & UCA0RXIFG)); // USCI_A0 RX Received?
    received_ch = UCA0RXBUF; // Store received data
    P1OUT |= (BIT5); // Unselect Device
}
```

**Case Study: "A Low-Power Battery less Wireless Temperature and Humidity Sensor with Passive Low Frequency RFID"**

Several applications require hermetically sealed environments, where physical parameter measurements such as temperature, humidity, or pressure are measured and, for several reasons, a battery-less operation is required. In such applications, a wireless data and power transfer is necessary. This application report shows how to implement an easy-to-use low-power wireless humidity and temperature sensor comprising a SHT21 from Sensrion, a MSP430F2274 microcontroller, and a TMS37157 PaLFI (passive low-frequency interface). The complete power for the wireless sensor and the MSP430F2274 is provided by the RFID base station (ADR2) reader included in the eZ430-TMS37157 demo kit.

**The application is divided in four steps:**

• **Charge phase:** Generate an RF field of 134.2 kHz from the ADR2 reader to the wireless sensor module to charge the power capacitor.

• **Downlink phase:** Send command or instruction to wireless sensor to start measurement.

• **Measurement and recharge phase:** Trigger measurement of temperature, recharge the power capacitor on the sensor device, and trigger humidity measurement.

• **Uplink phase:** Send measurement results via RF interface (134.2 kHz) back to ADR2 reader.

***<u>Device Specifications</u>***:

**<u>MSP430F2274:</u>**

- The MSP430F2274 is a 16-bit microcontroller from the 2xx family of the ultra-low-power MSP430™ family
- The supply voltage for this microcontroller ranges from 1.8 V to 3.6V. The MCU is capable of operating at frequencies up to 16 MHz.
- It has internal VLO that operates at 12 kHz at room temperature
- It has two timers (Timer_A and Timer_B), each with three capture/compare registers.
- An integrated 10-bit analog-to-digital converter (ADC10) supports conversion rates of up to 200 kbps.
- The current consumption of 0.7 mA during standby mode (LPM3) and 250 mA during active mode makes it an excellent choice for battery-powered applications.

**<u>TMS37157 PaLFI:</u>**

- The TMS37157 PaLFI is a dual interface passive RFID product from Texas Instruments. The device can communicate via the RF and the SPI (wired) interfaces.
- It offers 121 bytes of programmable EEPROM memory.
- The complete memory can be altered through the wireless interface, if the communication/read distances between the reader antenna and the PaLFI antenna are less than 10 cm to 30 cm.

- A microcontroller with a SPI interface has access to the entire memory through the 3-wire SPI interface of the TMS37157.
- If the TMS37157 is connected to a battery, it offers a battery charge function and a battery check function without waking the microcontroller, and consumes less power of about 60 nA in standby mode and about 70 µA in active mode.
- The PaLFI can completely switch off the microcontroller, resulting in an ultralow power consumption of the complete system.

**SHT21 Humidity and Temperature Sensor:**

- The extremely small SHT21 digital humidity and temperature sensor integrates sensors, calibration memory, and digital interface on 3x3 mm footprint.
- This results in cost savings, because no additional components are need and no investments in calibration equipment or process are necessary.
- One-chip integration allows for lowest power consumption, thus enabling energy harvesting and passive RFID solutions

**Interface Between MSP430F2274 and TMS37157 PaLFI**

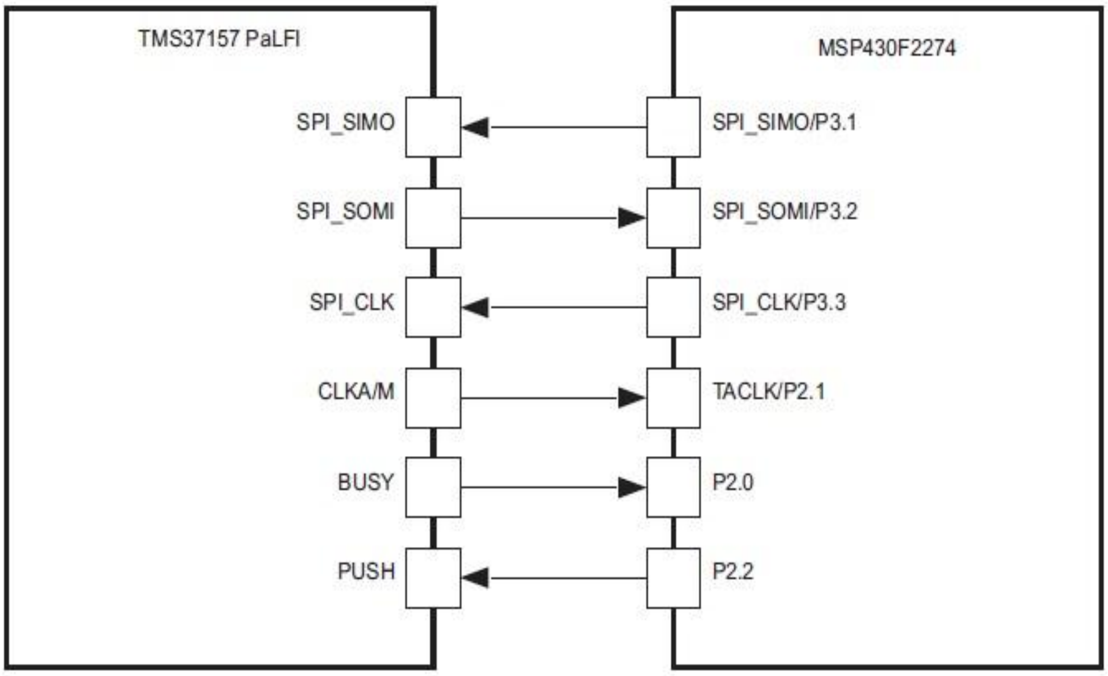


**Figure 1. Block Diagram of Interface Between MSP430F2274 and TMS37157**

- The TMS37157 is connected to the MSP430F2274 through a 3-wire SPI interface.
- The BUSY pin indicates the readiness of the TMS37157 to receive the next data byte from the MSP430F2274.

- The PUSH pin is used to wake up the PaLFI from standby mode so that the MSP430F2274 can access the EEPROM of the PaLFI.
- CLKAM is used for the antenna automatic tune feature of the PaLFI target board.

**Interface Between MSP430F2274 and SHT21:**

- I2C is used to connect both devices.
- The MSP430F2274 contains two communication modules. One is used as UART connection to a host PC, the other one is used to communicate to the TMS37157

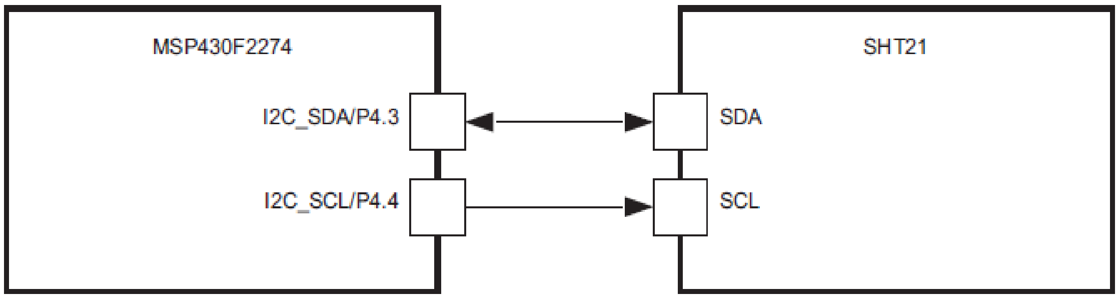


**Figure 2. Block Diagram of Interface Between MSP430F2274 and SHT21**

***Hardware Changes to Original PaLFI Board***

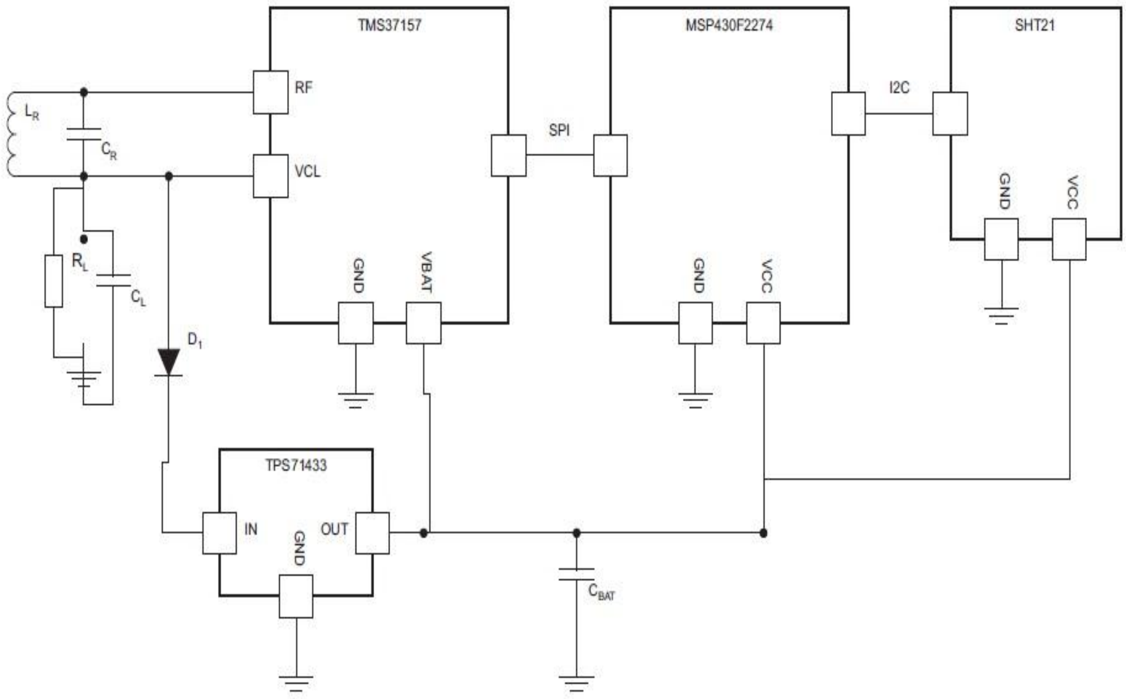


Figure 3. Principle Schematic of the Wireless Sensor

**Program Flow**

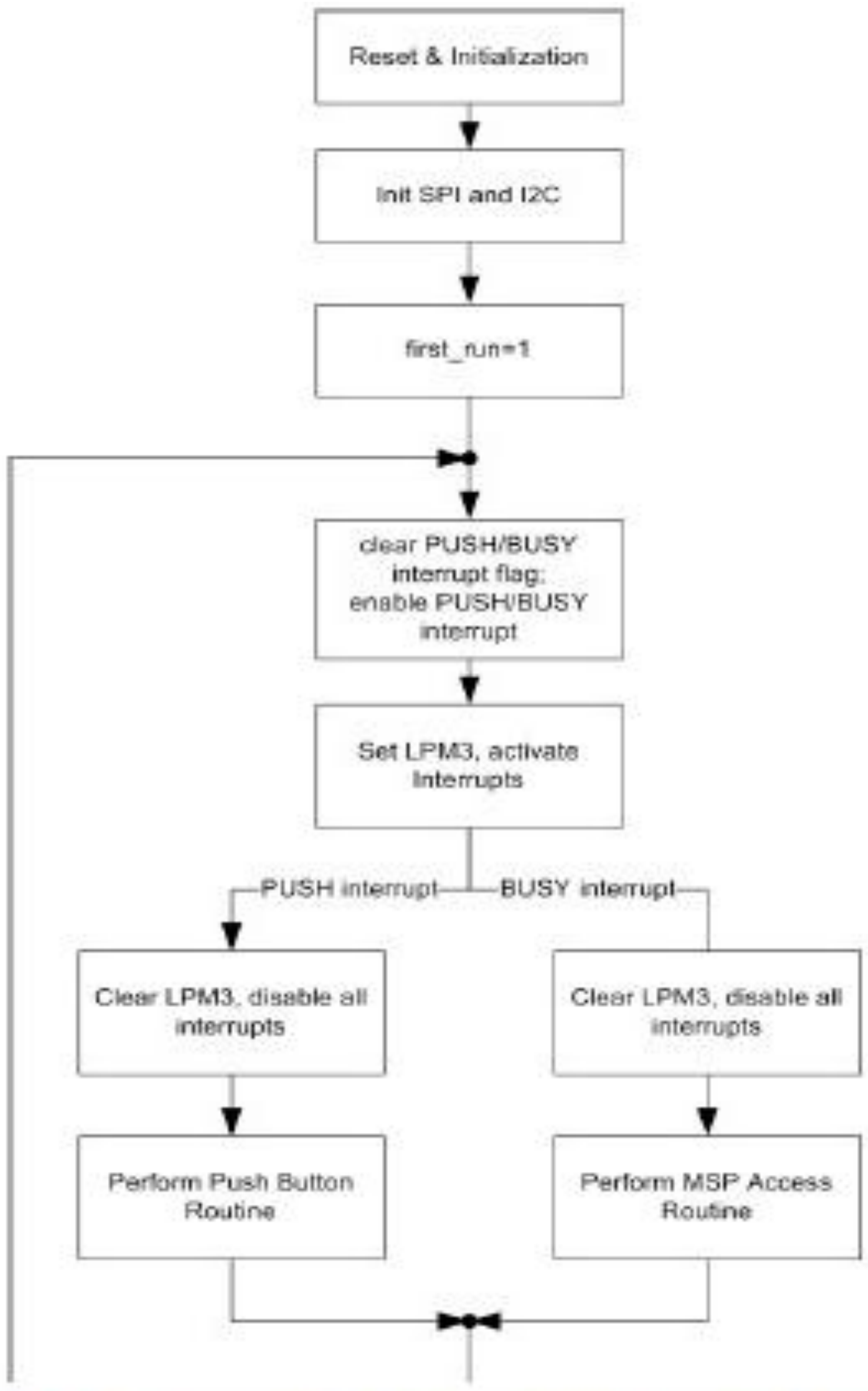

Reset & Initialization
Init SPI and I2C
first_run=1
clear PUSH/BUSY interrupt flag; enable PUSH/BUSY interrupt
Set LPM3, activate Interrupts
PUSH interrupt
BUSY interrupt
Clear LPM3, disable all interrupts
Clear LPM3, disable all interrupts
Perform Push Button Routine
Perform MSP Access Routine

Figure 8. Main Program Flow of MSP430F2274 Software

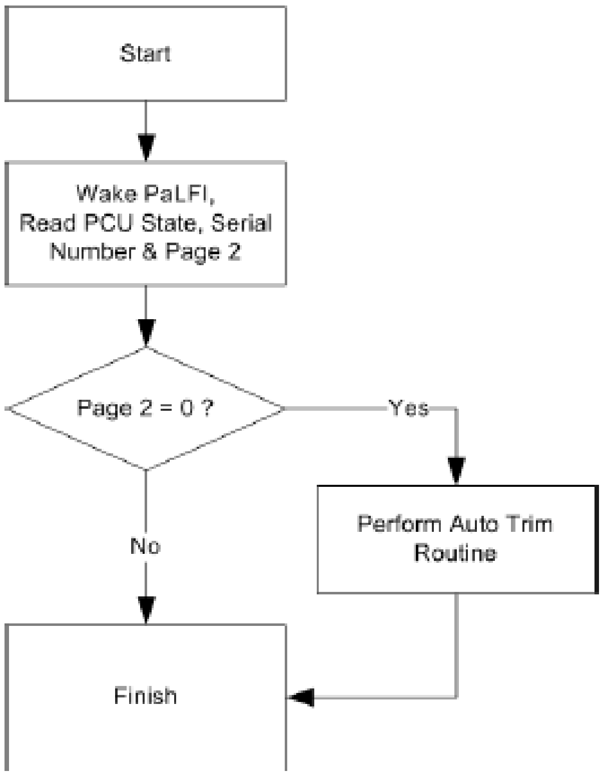

Start
Wake PaLFI, Read PCU State, Serial Number & Page 2
Page 2 = 0 ?
Yes
No
Perform Auto Trim Routine
Finish

**Figure 9. PUSH Interrupt Service Routine**

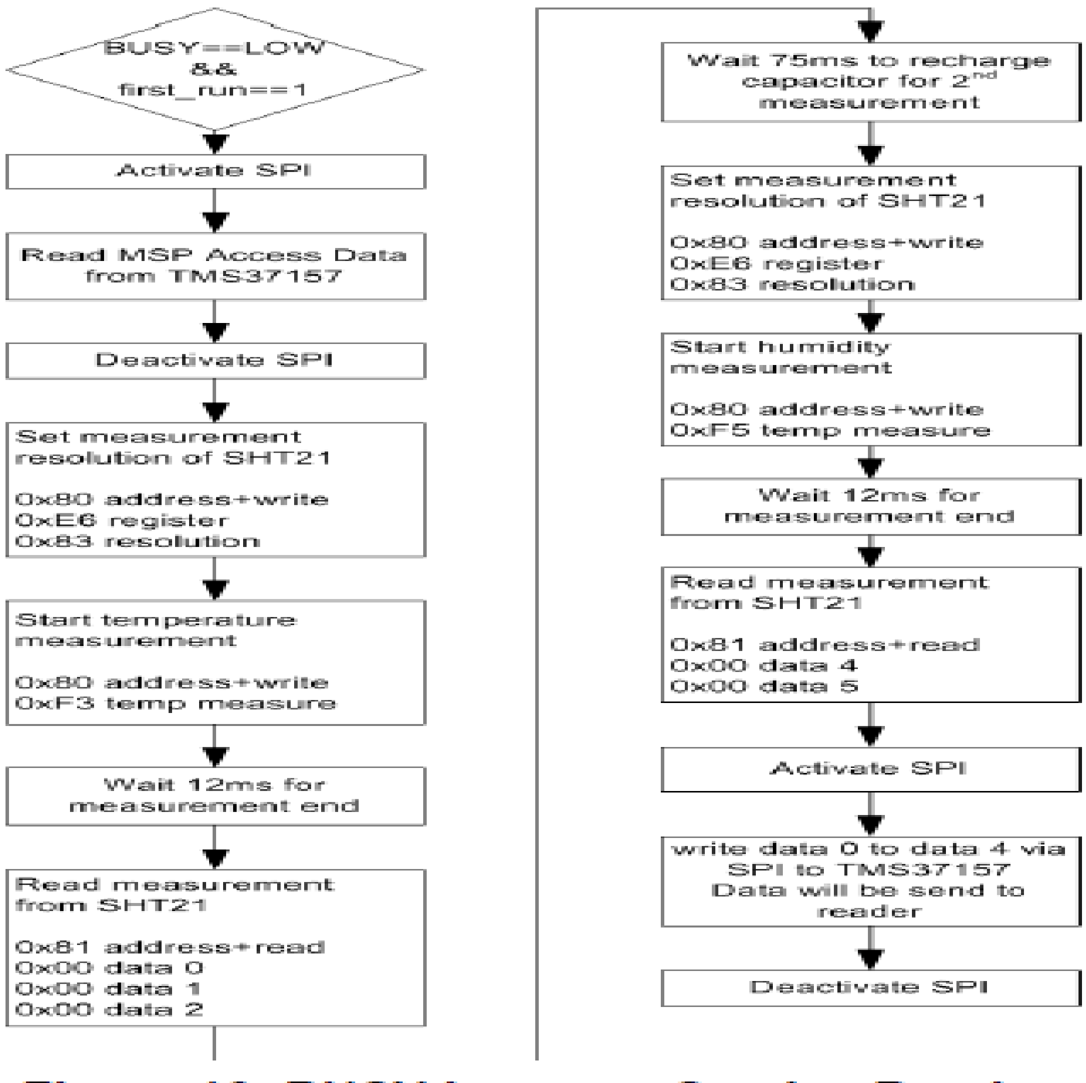

BUSY==LOW && first_run==1
Activate SPI
Read MSP Access Data from TMS37157
Deactivate SPI
Set measurement resolution of SHT21
0x80 address+write
0xE6 register
0x83 resolution
Start temperature measurement
0x80 address+write
0xF3 temp measure
Wait 12ms for measurement end
Read measurement from SHT21
0x81 address+read
0x00 data 0
0x00 data 1
0x00 data 2
Wait 75ms to recharge capacitor for 2nd measurement
Set measurement resolution of SHT21
0x80 address+write
0xE6 register
0x83 resolution
Start humidity measurement
0x80 address+write
0xF5 temp measure
Wait 12ms for measurement end
Read measurement from SHT21
0x81 address+read
0x00 data 4
0x00 data 5
Activate SPI
write data 0 to data 4 via SPI to TMS37157
Data will be send to reader
Deactivate SPI

**Figure 10. BUSY Interrupt Service Routine**

## What are the limitations of SPI protocol?

- Length between master and slave is limited, whereas it depends on no. of slave connected to it. Less the slave connected to it more the length can be use and vice versa.
- It doesn't have data security concept, whereas generally data doesn't loss because first thing is it’s short distance communication protocol and second is it has single master (only master can initiate request for communication) so one time only one communication is done.
- It doesn’t have concept of acknowledge verifying where data is deliver successfully or not.
- As the no. of slave increase no of Slave Select line increase leading occupying more no. of GPIOs, again it has some solution like Daisy-chained SPI bus but it make it some sort of slower.

## Discuss about the humidity sensor?

### What is thehumidity sensor?

A humidity sensor (or hygrometer) senses, measures and reports the relative humidity in the air. It therefore measures both moisture and air temperature. Relative humidity is the ratio of actual moisture in the air to the highest amount of moisture that can be held at that air temperature.

Humidity Sensor is one of the most important devices that has been widely in consumer, industrial, biomedical, and environmental etc. applications for measuring and monitoring Humidity.

Humidity is defined as the amount of water present in the surrounding air. This water content in the air is a key factor in the wellness of mankind. For example, we will feel comfortable even if the temperature is 00C with less humidity i.e. the air is dry.

But if the temperature is 100C and the humidity is high i.e. the water content of air is high, then we will feel quite uncomfortable. Humidity is also a major factor for operating sensitive equipment like electronics, industrial equipment, electrostatic sensitive devices and high voltage devices etc. Such sensitive equipment must be operated in a humidity environment that is suitable for the device.

Hence, sensing, measuring, monitoring and controlling humidity is a very important task.