Chapter 14. SSP Module and I^2C bus for External EEPROM Access

This chapter deals with serial peripheral interface (SPI) and Inter-IC (I^2C) bus operation using the Master Synchronous Serial Port (MSSP) module of 16F877. We only brief the SPI, and concentrate our focus on I^2C operation and its application to serial EEPROM access.

1. SSP Module and SPI Operation

The Master Synchronous Serial Port (MSSP) module of PIC 16F877 is a serial interface useful for communicating with other peripheral or microcontroller devices. These peripheral devices may be serial EEPROMs, display drivers, or external A/D converters. The MSSP module can operate in one of two modes: Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) bus operation. In the operation, there are two operational modes: Master and Slave modes.

The SPI mode is for transmitting and receiving 8-bits of data simultaneously. All four modes of SPI (i.e., SPI master mode, SPI slave mode, I^2C master mode, and I^2C slave mode) are supported. In SPI mode, typically three pins are used: Serial Data Out (SDO), Serial Data In (SDI), and Serial Clock (SCK). Additionally a fourth pin may be used when in a slave mode of operation: Slave Select (SS).

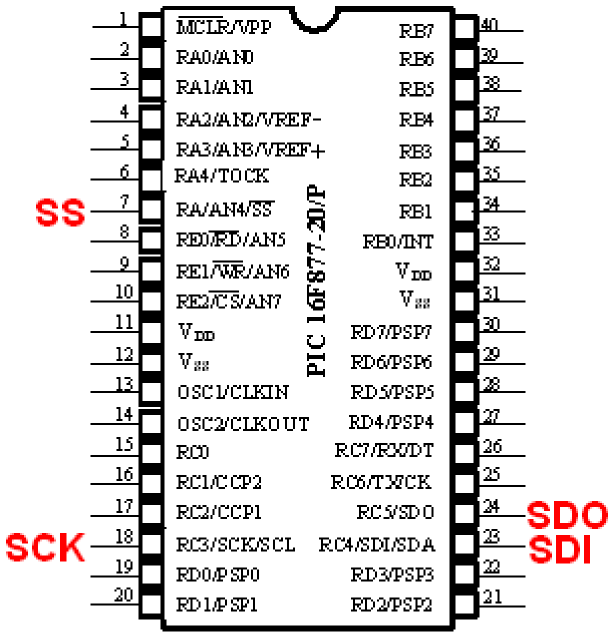


Fig. 86 SS, SCK, SDO and SDI pins of PIC 16F887

The SSP consists of a transmit/receive Shift Register (SSPSR) and a buffer register (SSPBUF). The SSPSR shifts the data in and out of the device, MSB first. The SSPBUF holds the data that was written to the SSPSR, until the received data is ready. Once the 8-bits of data have been received, that byte is moved to the SSPBUF register. Then the buffer full detect bit, BF (SSPSTAT<0>), and the interrupt flag bit, SSPIF, are set. This double buffering of the received data (SSPBUF) allows the next byte to start reception before reading the data that was just received. Any write to the SSPBUF register during transmission/reception of data will be ignored, and the write collision detect bit, WCOL (SSPCON1<7>), will be set. Our code should handle this to clear the WCOL bit so that it can be determined if the following write to the SSPBUF register completed successfully.

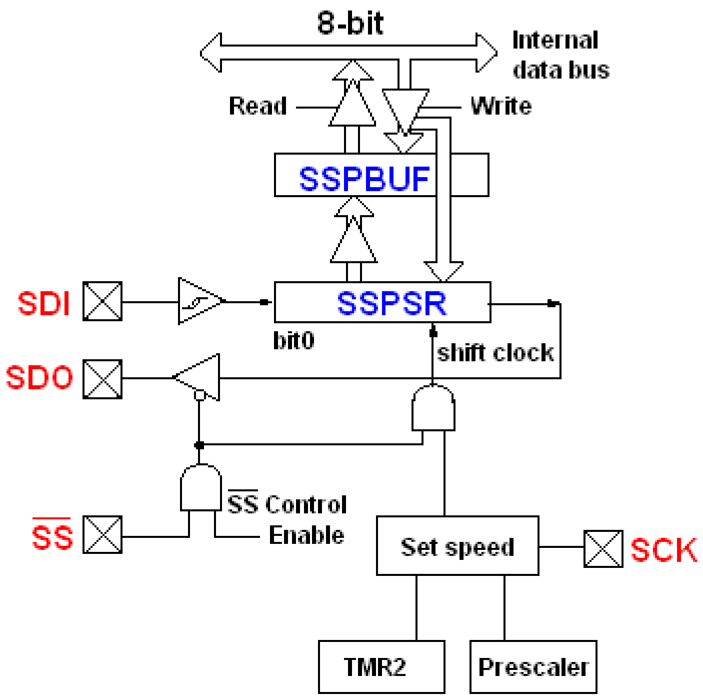


Fig. 87 Synchronous Serial Port (SSP)

When our code is expecting to receive valid data, the SSPBUF should be read before the next byte of data to transfer is written to the SSPBUF. Buffer full bit, BF (SSPSTAT<0>), indicates when SSPBUF has been loaded with the received data (transmission is complete).
When the SSPBUF is read, the BF bit is cleared. This data may be irrelevant if the SPI is only a transmitter. Generally the SSP Interrupt is used to determine when the transmission/reception has completed. The SSPBUF must be read and/or written. If the interrupt method is not going to be used, then software polling can be done to ensure that a write collision does not occur.

Details of the SPI operation including mater and slave modes can be found in the manuals published from Microchip Technology.

2. I²C Bus Operation

A. I²C Bus-Overview

This section is greatly indebted to several publications on I²C bus from Phillips Semiconductor Co. I am merely summarizing otherwise very rich documentation on the subject. If there is any error or discrepancy between the overview here and the original documentation, it's all because of my misunderstanding or mistakes.

In consumer electronics, telecommunications and industrial electronics, there are often many similarities between seemingly unrelated designs. For example, nearly every system includes:

Some intelligent control, usually a single-chip microcontroller; General-purpose circuits like LCD drivers, remote I/O ports, RAM, EEPROM, or data converters; Application-oriented circuits such as digital tuning and signal processing circuits for radio and video systems, or DTMF generators for telephones with tone dialing.

To exploit these similarities to the benefit of both systems designers and equipment manufacturers, as well as to maximize hardware efficiency and circuit simplicity, Philips developed a simple bidirectional 2-wire bus for efficient inter-IC control. This bus is called the Inter IC or I^{2}C-bus. At present, Philips' IC range includes more than 150 CMOS and bipolar I^{2}C-bus compatible types for performing functions in all three of the previously mentioned categories. All I^{2}C-bus compatible devices incorporate an on-chip interface which allows them to communicate directly with each other via the I^{2}C-bus. This design concept solves the many interfacing problems encountered when designing digital control circuits.

Here are some of the features of the I^{2}C-bus:

- Only two bus lines are required; a serial data line (SDA) and a serial clock line (SCL)
- Each device connected to the bus is software addressable by a unique address and simple master/ slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers
- It's a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the standard mode or up to 400 kbit/s in the fast mode
- On-chip filtering rejects spikes on the bus data line to preserve data integrity. The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF.

The I^{2}C-bus supports any IC fabrication process (NMOS, CMOS, bipolar). Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus. Each device is recognized by a unique address — whether it's a microcontroller, LCD driver, memory or keyboard interface — and can operate as either a transmitter or receiver, depending on the function of the device. Obviously an LCD driver is only a receiver, whereas a memory can both receive and transmit data.

In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers. A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave. Some I^{2}C Bus Terminologies are listed below:

• Transmitter:	The device which sends the data to the bus
• Receiver:	The device which receives the data from the bus
• Master:	The device which initiates a transfer, generates clock signals and terminates a transfer
• Slave:	The device addressed by a master
• Multi-master:	More than one master can attempt to control the bus at the same time without corrupting the message

- Arbitration: Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the message is not corrupted
- Synchronization: Procedure to synchronize the clock signals of two or more devices

The I^2C-bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. As masters are usually microcontrollers, let's consider the case of a data transfer between two microcontrollers such as 16F877(A and B) connected to the I^2C-bus. This highlights the master-slave and receiver-transmitter relationships to be found on the I^2C-bus. It should be noted that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows:
1. Suppose microcontroller A wants to send information to microcontroller B:
– microcontroller A (master), addresses microcontroller B (slave)
– microcontroller A (master-transmitter), sends data to microcontroller B (slave-receiver)
– microcontroller A terminates the transfer.
2. If microcontroller A wants to receive information from microcontroller B:
– microcontroller A (master) addresses microcontroller B (slave)
– microcontroller A (master-receiver) receives data from microcontroller B (slave-transmitter)
– microcontroller A terminates the transfer.

Even in the reception case, the master (microcontroller A) generates the timing and terminates the transfer.

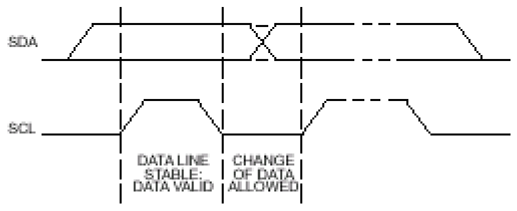
The possibility of connecting more than one microcontroller to the I^2C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid the chaos that might ensue from such an event — an arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I^2C interfaces to the I^2C-bus. If two or more masters try to put information onto the bus, the first to produce a 'one' when the other produces a 'zero' will lose the arbitration. The clock signals during arbitration are a synchronized combination of the clocks generated by the masters using the wired-AND connection to the SCL line.

Generation of clock signals on the I^2C-bus is always the responsibility of master devices; each master generates its own clock signals when transferring data on the bus. Bus clock signals from a master can only be altered when they are stretched by a slow-slave device holding-down the clock line, or by another master when arbitration occurs.

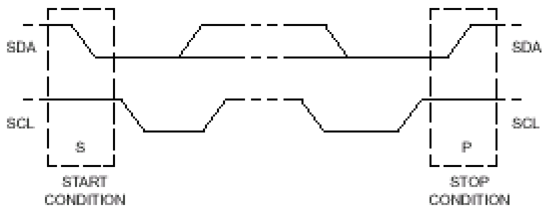
Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a pull-up resistor. When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector in order to perform the wired-AND function. Data on the I^2C-bus can be transferred at a rate up to 100 kbit/s in the standard-mode, or up to 400 kbit/s in the fast-mode. The number of interfaces connected to the bus is solely dependent on the bus capacitance limit of 400pF. The pull-up resistors for SDA and SCL used in our 16F877 and 24LC16 serial EEPROM are both 1.8 kohm.

B. I^2C Bus Protocol

Bit Transfer: Due to the variety of different technology devices (CMOS, NMOS, bipolar) which can be connected to the I 2 C-bus, the levels of the logical ‘0’ (LOW) and ‘1’ (HIGH) are not fixed and depend on the associated level of V DD. One clock pulse is generated for each data bit transferred. The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW.

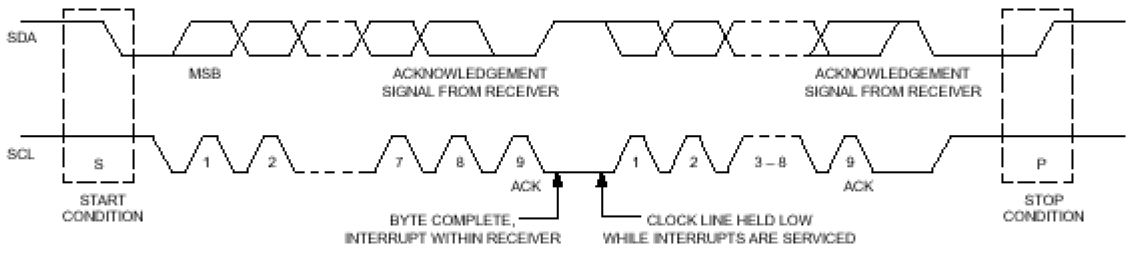


Within the procedure of the I 2 C-bus, unique situations arise which are defined as START and STOP conditions (see Figure below).



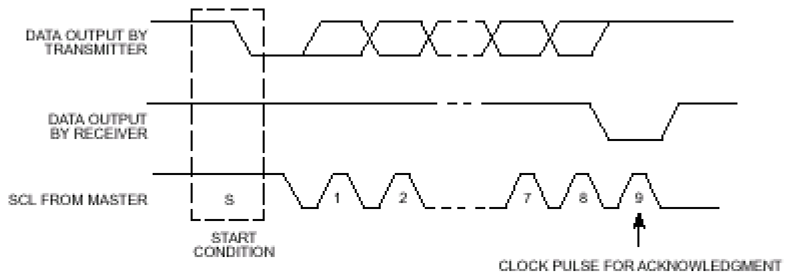
A HIGH to LOW transition on the SDA line while SCL is HIGH is one such unique case. This situation indicates a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition. START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition. This bus free situation is specified in Section 15.0. Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period in order to sense the transition.

Transferring Data: Every byte put on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first.


SDA
MSB
ACKNOWLEDGEMENT SIGNAL FROM RECEIVER
ACKNOWLEDGEMENT SIGNAL FROM RECEIVER
SCL
S
1
2
7
8
9
ACK
1
2
3 – 8
9
ACK
P
START CONDITION
BYTE COMPLETE, INTERRUPT WITHIN RECEIVER
CLOCK LINE HELD LOW WHILE INTERRUPTS ARE SERVICED
STOP CONDITION

If a receiver can't receive another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the transmitter into a wait state. Data transfer then continues when the receiver is ready for another byte of data and releases clock line SCL. In some cases, it's permitted to use a different format from the I 2 C-bus format (for CBUS compatible devices for example). A message which starts with such an address can be terminated by generation of a STOP condition, even during the transmission of a byte. In this case, no acknowledge is generated.

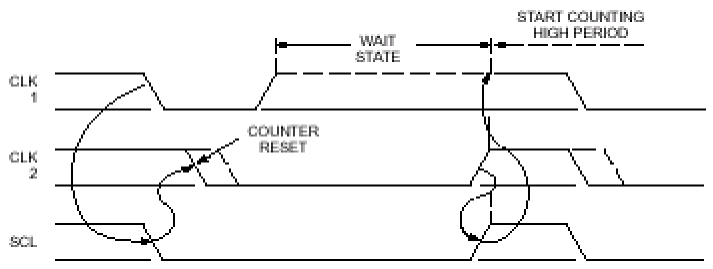
Data transfer with acknowledge is obligatory. The acknowledge-related clock pulse is generated by the master. The transmitter releases the SDA line (HIGH) during the acknowledge clock pulse. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse. Of course, set-up and hold times (specified in Section 15.0) must also be taken into account.



Usually, a receiver which has been addressed is obliged to generate an acknowledge after each byte has been received. When a slave-receiver doesn't acknowledge the slave address (for example, it's unable to receive because it's performing some real-time function), the data line must be left HIGH by the slave. The master can then generate a STOP condition to abort the transfer. If a slave-receiver does acknowledge the slave address but, some time later in the transfer cannot receive any more data bytes, the master must again abort the transfer. This is indicated by the slave generating the not acknowledge on the first byte to follow. The slave leaves the data line HIGH and the master generates the STOP condition.

If a master-receiver is involved in a transfer, it must signal the end of data to the slave-transmitter by not generating an acknowledge on the last byte that was clocked out of the slave. The slave-transmitter must release the data line to allow the master to generate a STOP or repeated START condition.

Arbitration and Clock Generation: All masters generate their own clock on the SCL line to transfer messages on the I^2C-bus. Data is only valid during the HIGH period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place. Clock synchronization is performed using the wired-AND connection of I^2C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and, once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached.



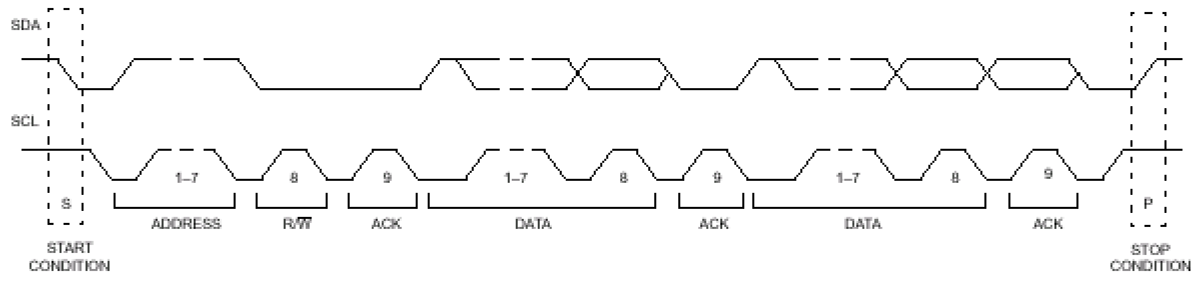
However, the LOW to HIGH transition of this clock may not change the state of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the device with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time. When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW. In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period.

A master may start a transfer only if the bus is free. Two or more masters may generate a START condition within the minimum hold time of the START condition which results in a defined START condition to the bus. Arbitration takes place on the SDA line, while the SCL line is at the HIGH level, in such a way that the master which transmits a HIGH level, while another master is transmitting a LOW level will switch off its DATA output stage because the level on the bus doesn't correspond to its own level. Arbitration can continue for many bits. Its first stage is comparison of the address bits. If the masters are each trying to address the same device, arbitration continues with comparison of the data. Because address and data information on the I^{2}C-bus is used for arbitration, no information is lost during this process.

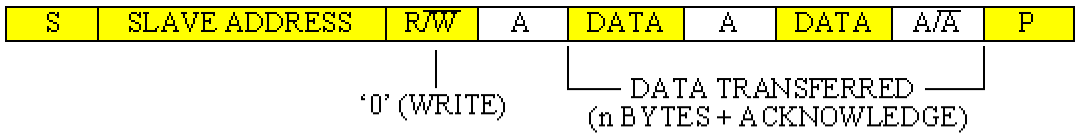
A master which loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration. If a master also incorporates a slave function and it loses arbitration during the addressing stage, it's possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave-receiver mode.

In addition to being used during the arbitration procedure, the clock synchronization mechanism can be used to enable receivers to cope with fast data transfers, on either a byte level or a bit level. On the byte level, a device may be able to receive bytes of data at a fast rate, but needs more time to store a received byte or prepare another byte to be transmitted. Slaves can then hold the SCL line LOW after reception and acknowledgement of a byte to force the master into a wait state until the slave is ready for the next byte transfer in a type of handshake procedure. On the bit level, a device such as a microcontroller without, or with only a limited hardware I^{2}C interface on-chip can slow down the bus clock by extending each clock LOW period. The speed of any master is thereby adapted to the internal operating rate of this device.

Formats with 7-Bit Addresses: Data transfers follow the format shown below. After the START condition (S), a slave address is sent. This address is 7 bits long followed by an eighth bit which is a data direction bit (R/W) — a ‘zero’ indicates a transmission (WRITE), a ‘one’ indicates a request for data (READ). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. Various combinations of read/write formats are then possible within such a transfer.

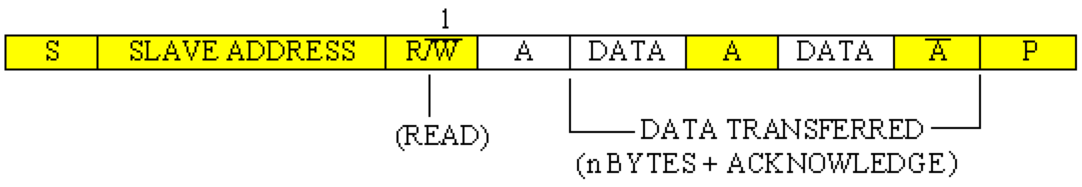


Possible data transfer formats are:
(a) Master-transmitter transmits to slave-receiver. The transfer direction is not changed

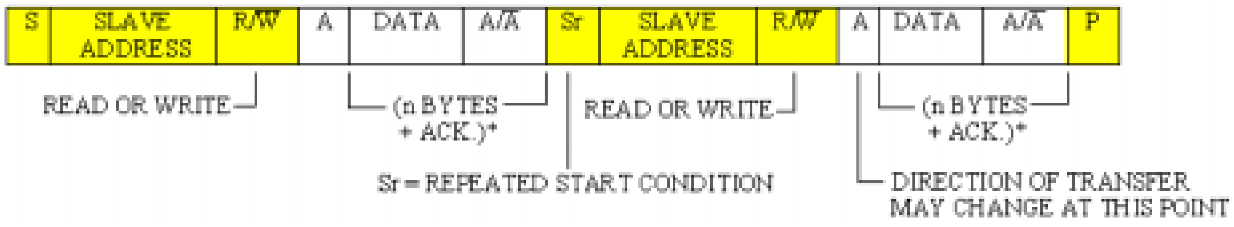


🟨	FROM MASTER TO SLAVE	A = ACKNOWLEDGE (SDA LOW)
		$\overline{A}$ = NOT ACKNOWLWDGE (SDA HIGH)
⬜	FROM SLAVE TO MASTER	S = START CONDITION
		P = STOP CONDITION

(b)Master reads slave immediately after first byte. At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This acknowledge is still generated by the slave. The STOP condition is generated by the master.



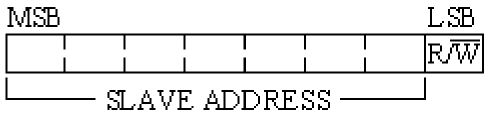
(c) Combined format. During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed. If a master receiver sends a repeated START condition, it has previously sent a not acknowledge (A).



7-Bit Addressing:

The addressing procedure for the I^{2}C-bus is such that the first byte after the START condition usually determines which slave will be selected by the master. The exception is the 'general call' address which can address all devices. When this address is used, all devices should, in theory, respond with an acknowledge. However, devices can be made to ignore this address. The second byte of the general call address then defines the action to be taken.

The first seven bits of the first byte make up the slave address.



The eighth bit is the LSB. It determines the direction of the message. A 'zero' in the least significant position of the first byte means that the master will write information to a selected slave. A 'one' in this position means that the master will read information from the slave.

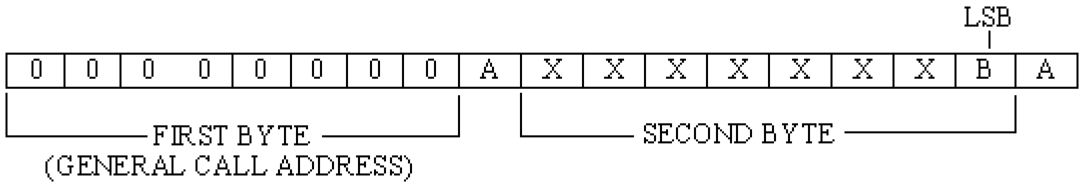
When an address is sent, each device in a system compares the first seven bits after the START condition with its address. If they match, the device considers itself addressed by the master as a slave-receiver or slave-transmitter, depending on the R/W bit. A slave address can be made-up of a fixed and a programmable part. Since it's likely that there will be several identical devices in a system, the programmable part of the slave address enables the maximum possible number of such devices to be connected to the I^{2}C-bus. The number of programmable address bits of a device depends on the number of pins available. For example, if a device has 4 fixed and 3 programmable address bits, a total of 8 identical devices can be connected to the same bus.

The I^{2}C-bus committee coordinates allocation of I 2 C addresses. Two groups of eight addresses (0000XXX and 1111XXX) are reserved for the purposes shown in Table below. The bit combination 11110XX of the slave address is reserved for 10-bit addressing.

SLAVE ADDRESS	R / bit	DESCRIPTION
0000 000	0	General call address
0000 000	1	START byte
0000 001	X	CBUS address
0000 010	X	Address reserved for different bus format
0000 011	X	Reserved for future purposes
0000 1XX	X	
1111 1XX	X	
1111 0XX	X	10 – bit slave addressing

The general call address is for addressing every device connected to the I^2C-bus. However, if a device doesn't need any of the data supplied within the general call structure, it can ignore this address by not issuing an acknowledgement. If a device does require data from a general call address, it will acknowledge this address and behave as a slave-receiver. The second and following bytes will be acknowledged by every slave-receiver capable of handling this data.

A slave which cannot process one of these bytes must ignore it by not acknowledging. The meaning of the general call address is always specified in the second byte.



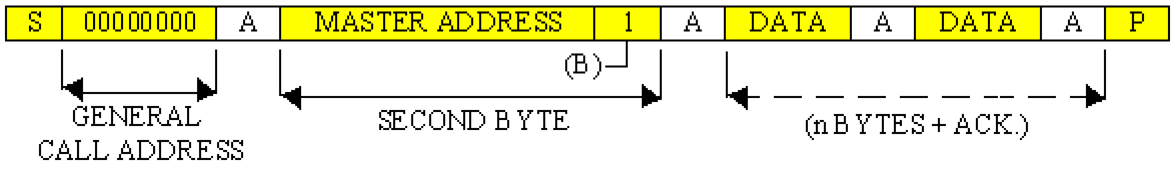
There are two cases to consider:

- When the least significant bit B is a 'zero'
- When the least significant bit B is a 'one'.

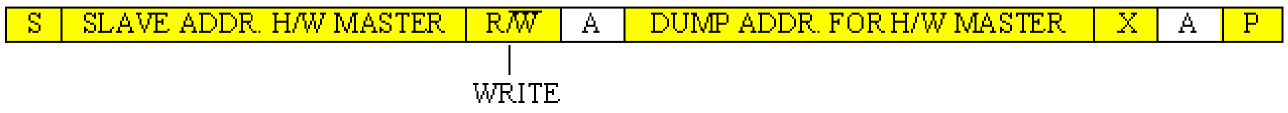
When bit B is a 'zero'; the second byte has the following definition:

- o 00000110 (H'06'). Reset and write programmable part of slave address by hardware. On receiving this 2-byte sequence, all devices designed to respond to the general call address will reset and take in the programmable part of their address. Precautions have to be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus
- o 00000100 (H'04'). Write programmable part of slave address by hardware. All devices which define the programmable part of their address by hardware (and which respond to the general call address) will latch this programmable part at the reception of this two byte sequence. The device will not reset.
- o 00000000 (H'00'). This code is not allowed to be used as the second byte.

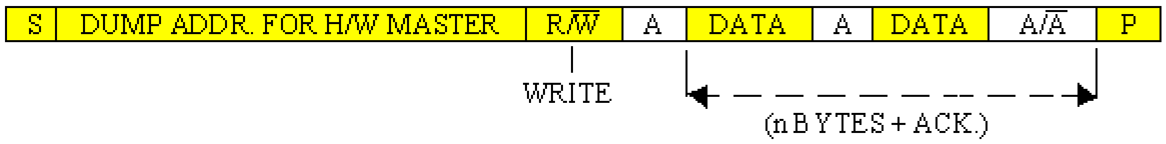
When bit B is a 'one'; the 2-byte sequence is a 'hardware general call'. This means that the sequence is transmitted by a hardware master device, such as a keyboard scanner, which cannot be programmed to transmit a desired slave address. Since a hardware master doesn't know in advance to which device the message has to be transferred, it can only generate this hardware general call and its own address — identifying itself to the system.



The seven bits remaining in the second byte contain the address of the hardware master. This address is recognized by an intelligent device (e.g. a microcontroller) connected to the bus which will then direct the information from the hardware master. If the hardware master can also act as a slave, the slave address is identical to the master address. In some systems, an alternative could be that the hardware master transmitter is set in the slave-receiver mode after the system reset. In this way, a system configuring master can tell the hardware master-transmitter (which is now in slave-receiver mode) to which address data must be sent. After this programming procedure, the hardware master remains in the master-transmitter mode.

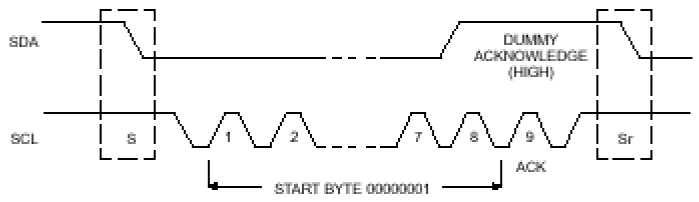


(a) Configuring master sends dump address to hardware master



(b) Hardware master dumps data to selected slave

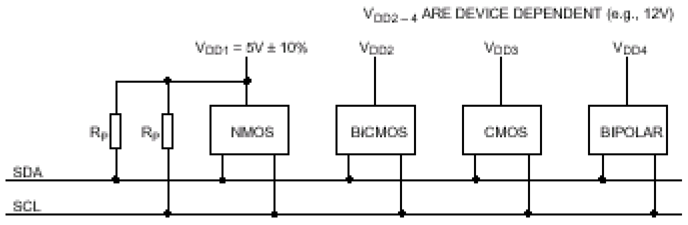
Microcontrollers can be connected to the I^{2}C-bus in two ways. A microcontroller with an on-chip hardware I^{2}C-bus interface can be programmed to be only interrupted by requests from the bus. When the device doesn’t have such an interface, it must constantly monitor the bus via software. Obviously, the more times the microcontroller monitors, or polls the bus, the less time it can spend carrying out its intended function. There is therefore a speed difference between fast hardware devices and a relatively slow microcontroller which relies on software polling.
In this case, data transfer can be preceded by a start procedure which is much longer than normal.



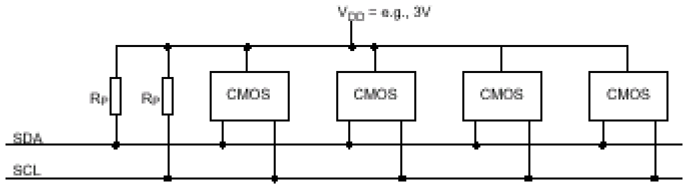
The start procedure consists of:
– A START condition (S)
– A START byte (00000001)
– An acknowledge clock pulse (ACK)
– A repeated START condition (Sr).

After the START condition S has been transmitted by a master which requires bus access, the START byte (00000001) is transmitted. Another microcontroller can therefore sample the SDA line at a low sampling rate until one of the seven zeros in the START byte is detected. After detection of this LOW level on the SDA line, the microcontroller can switch to a higher sampling rate to find the repeated START condition Sr which is then used for synchronization. A hardware receiver will reset on receipt of the repeated START condition Sr and will therefore ignore the START byte. An acknowledge-related clock pulse is generated after the START byte. This is present only to conform with the byte handling format used on the bus. No device is allowed to acknowledge the START byte.

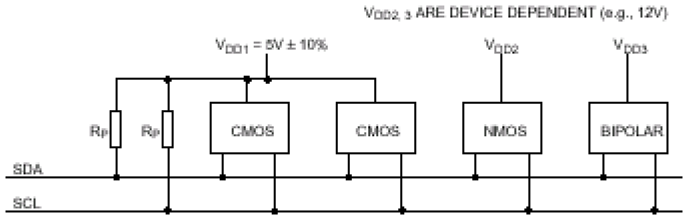
Electrical Characteristics for I^{2}C Bus Devices:
I^{2}C-bus devices with fixed input levels of 1.5 V and 3 V can each have their own appropriate supply voltage. Pull-up resistors must be connected to a 5V supply.



I^{2}C-bus devices with input levels related to V DD must have one common supply line to which the pull-up resistor is also connected.

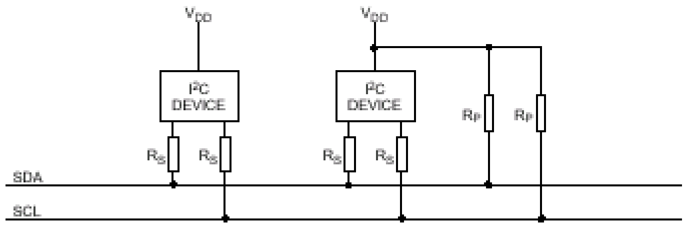


When devices with fixed input levels are mixed with devices with input levels related to V_{DD} , the latter devices must be connected to one common supply line of 5 V and must have pull-up resistors connected to their SDA and SCL pins.



Input levels are defined in such a way that:

- The noise margin on the LOW level is 0.1 V_{DD}
- The noise margin on the HIGH level is 0.2 V_{DD}
- As shown in figure below, series resistors (R S) of, e.g., 30ohm can be used for protection against high-voltage spikes on the SDA and SCL lines (due to flash-over of a TV picture tube, for example).



3. Serial EEPROM

Basic Serial EEPROM Operation

Serial EEPROM technology is one of the non-volatile memory technologies that has emerged as a leading embedded control solution. Serial EEPROMS are ideal non-volatile cost effective memory solutions in applications that require: (a) small footprint and board space as in cellular phone applications, (b) BYTE level ERASE, WRITE, and READ of data as in a TV tuner, (c) low voltage and current for handheld battery applications as in a keyless entry transmitter, (d) multiple non-volatile functions in the same application such as a VCR, or (e) low availability of microcontroller I/O lines

The common applications for serial EEPROMS are:TV tuners, VCRs, CD players, cameras, radios, and remote controls; Airbags, anti-lock brakes, odometers, radios, and keyless entry; Printers, copiers, PCs, and portable PCs; Cellular, cordless and full feature phones, faxes, modems, pagers, and satellite receivers; Bar code readers, point-of-sale terminals, smart cards, lock boxes, garage door openers, and test measurement.

The typical functions that serial EEPROMs are utilized for are: memory storage of channel selectors or analog controls (volume, tone, etc.) in consumer electronics products, power down storage and retrieval of events such as fault detection or error diagnostics in automotive products, electronic real time event or maintenance logs such as page counting in office automation products. Also, configuration or DIP switch storage in office automation products, last number redial storage and speed dial number storage in telecom products, and user in-circuit reprogrammable look up tables such as bar code readers, point-of-sale terminals, environmental controls and other industrial products

Other application examples include: data storage from a learn function as in a remote control transmitter, ID number storage for security or remote access for electronic keys and entry databases, and reprogrammable calibration data for test equipment or analog interface products

There are two primary protocols for serial EEPROM access: 3-wire bus and 2-wire bus.
A 2- wire product is utilized in applications that require an I^{2}C bus, noise immunity, limited microcontroller I/O pin availability, or a WRITE buffer for multiple bytes to be stored with one instruction. A 3-wire product is utilized in applications that have limited protocol requirements, an SPI protocol, higher clock frequency requirements, or 16-bit data width applications. Since our discussion in the chapter is I^{2}C bus, our focus here is the 2-wire bus protocol. Readers who are interested in 3-wire protocol can refer Microchip Technology for manual valuable manuals and application notes. However, there still is merit to brief on the 3-wire protocol, since the popular serial EEPROMs from Microchip Technology has different lines of product depending upon which protocol they use. The 3-wire common serial EEPROM from Microchip Technology has the common nomenclature of 93XXXX.

- 93XX06: 256 bit
- 93XX46: 1K bit
- 93XX56: 2K bit
- 93XX66: 4K bit

The 3-wire bus requires the following 4 pins: CS (Chip Select), DI (data in), CLK (Clock), DO (data out). All 93XXXX parts are hardware compatible for these four pins.

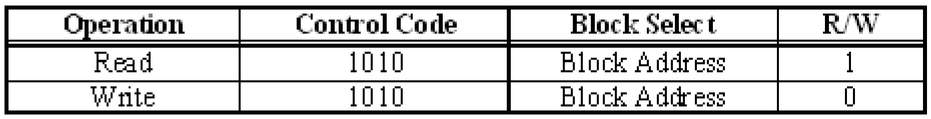
2-wire bus operation
2-wire bus system uses the following pins: SCL (Serial clock), SDA (Serial Data),WP (Active High WRITE Protection), and A0, A1, and A2 (Chip or block select). Only the SCL and SDA pins are essential for bus operation, and the other pins are supplementary. SDA's open-drain requires a pull-up resistor to VDD. The data is organized as 8 bit. Signals are level triggered, not edge triggered. Also, there are filters on the inputs that will filter noise glitches <100ns wide.

The I^{2}C protocol utilizes master/slave bi-directional communication. A device that sends data onto the bus is defined as the transmitter, and a device that is receiving data is the receiver. Both the master and the slave can operate as the transmitter or receiver. The bus must be controlled by a master device (most often a microcontroller), which generates the serial clock (SCL), controls the bus direction, and generates the START and the STOP conditions.

The serial EEPROM is the slave. The serial EEPROM will be the bus transmitter during READ operations and when the serial EEPROM must acknowledge data transmitted by the master. START and STOP bits control the bus activity. Operations must begin with a START bit and end with a STOP bit. A START bit is when SDA transitions LOW while SCL is HIGH while observing the START set-up and hold time specifications. A STOP bit is when SDA transitions HIGH while SCL is HIGH while observing the STOP set-up and hold time specifications. Data is recognized as valid while SCL is high. The data on SDA must observe data in set-up and hold specifications before and after SCL is pulsed. There is only one bit of data for each SCL pulse.

The common 2-wire device of Microchip Technology nomenclature is 24XXXX and 85XXXX. The 2-wire serial EEPROM we use in this chapter is 24LC16B. The 24LC16B is a 16K bit EEPROM, and it is organized as eight blocks of 256 x 8 bit memory. The 24LC16B is available in the standard 8-pin DIP, surface mount SOIC, and TSSOP packages.

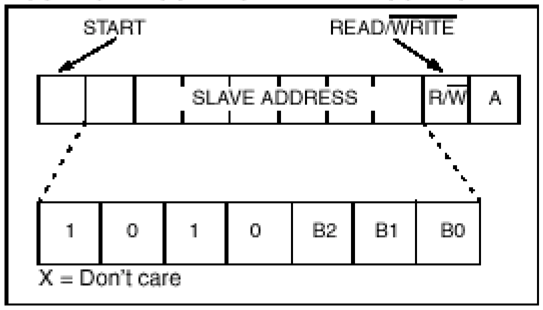
In the 24LC16B access, a control byte is the first byte received following the start condition from a master device. The control byte consists of a four-bit control code, for the 24LC16B this is set as 1010 binary for read and write operations.



Operation	Control Code	Block Select	R/W
Read	1010	Block Address	1
Write	1010	Block Address	0

The next three bits of the control byte are the block select bits (B2, B1, B0). They are used by the master device to select which of the eight 256 word blocks of memory are to be accessed. These bits are in effect the three most significant bits of the word address. It should be noted that the protocol limits the size of the memory to eight blocks of 256 words, therefore the protocol can support only one 24LC16B per system.
The last bit of the control byte defines the operation to be performed. When set to one a read operation is selected, when set to zero a write operation is selected. Following the start condition, the 24LC16B monitors the SDA bus checking the device type identifier being

transmitted, upon a 1010 code the slave device outputs an acknowledge signal on the SDA line. Depending on the state of the R/W bit, the 24LC16B will select a read or write operation.



4. Serial EEPROM Access with 16F877

After a length introduction of MSSP with I^2C operation and serial EEPROM, we are finally here to discuss the PIC16F877 access to the serial EEPROM 24LC16B. There are several registers involved in the I^2C operation. We will move around, from MSSP module set up and initialization to byte writing and byte reading, controlling the bits of the registers. Before we proceed, it's beneficial at least to be familiar with the involved registers. They are listed below:

- SSP Control Register (SSPCON)
- SSP Control Register2 (SSPCON2)
- SSP Status Register (SSPSTAT)
- Pin Direction Control (TRISC)
- Serial Receive/Transmit Buffer (SSPBUF)
- SSP Shift Register (SSPSR) - Not directly accessible
- SSP Address Register (SSPADD)
- SSP Hardware Event Status (PIR1)
- SSP Interrupt Enable (PIE1)
- SSP Bus Collision Status (PIR2)
- SSP Bus Collision Interrupt Enable (PIE2)

<u>MSSP Module Initialization and Set-up</u>

Step 1: Since the two-wire point, SDA and SCL, share with PORTC pins, SCL with PORTC<3>, and SDA with PORTC<4>, the declaration of the direction of the pins must be properly configured at the first step of initialization. An important thing here is that we have to set the two pins as inputs.

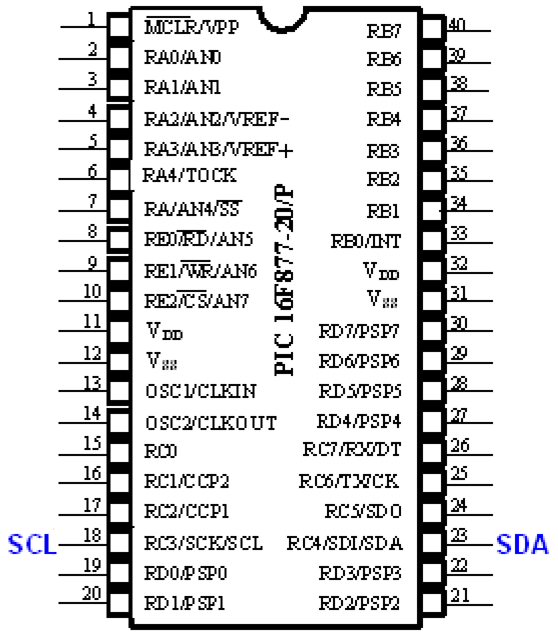


Fig. 88 SDA and SCL pins of PIC 16F877

The following lines of code would be enough for the PORTC direction for the two wires.

```
;PORTC setup  - SDA and SCL both as inputs
  movlw    0x18        ;0001 1000
  banksel  TRISC
  movwf    TRISC       ;RC4 (SDA) and RC3(SCL) as inputs
```

Step 2: In this step, we select I^{2}C operation of MSSP. Let's consider the SSPCON. Here we focus only on I^{2}C application, and therefore, each bit configuration indicated here is valid only for I^{2}C operation. SSPCON is for the module activation and mode selection. As indicated below, for I^{2}C operation of SSP, there are SSPEN bit for enabling the SSP module and the four bits of SSPM for mode selection. In I2C operation for 16F877 master mode, SSPEN must be set and SSPM<3:0> must be configured as 1000.

SSPCON1: SSP Control Register1

WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0

bit 7 bit 0

bit 7	**WCOL:** Write Collision Detect bit 0 = No collision
bit 6	**SSPOV:** Receive Overflow Indicator bit 0 = No overflow
bit 5	**SSPEN:** Synchronous Serial Port Enable bit 1 = Enables the serial port and configures the SDA and SCL pin as the source of the serial port pins 0 = Disables serial port and configures these pins as I/O port pins

bit 4	**CKP:** Clock Polarity Select bit
	Unused in I^2C master mode
bit 3 – 0	**SSPM3:SSPM0:** Synchronous Serial Port Mode Select bit
	0110 = I^2C slave mode, 7-bit address
	0111 = I^2C slave mode, 10-bit address
	1000 = I^2C master mode, clock = Fosc / (4 * (SSPADD + 1))

The following lines would serve as the SSPCON configuration of I^2C operation.

```
  movlw     0x28      ;0010 1000 (SSPEN=1, SSPM3:0= 1000 ) master mode
  banksel   SSPCON
  movwf     SSPCON
```

One thing we note here is that since we selected 1000 as for SSPM<3:0>, the clock speed of data flow (this is called baud rate of I^2C operation) is determined by $Clock_{speed} = \frac{F_{osc}}{4*[SSPADD]+1}$.

Step3: We decide the clock (or speed) of the data flow in I^2C operation. There are two types: a more standard of 400kHz, and modest speed of 100kHz. In the example we choose to use the speed of 100kHz. Then, from the equation in the step 2, we can an equation for the content of SSPADD register.

$$[SSPADD] = \frac{1}{4}\left(\frac{F_{osc}}{Clock_{speed}} - 1\right)$$

If we plug in 20MHz as F_{osc} and 100kHz as $Clock_{speed}$, then the content of SSPADD is:

$$[SSPADD] = \frac{1}{4}\left(\frac{20{,}000{,}000}{100{,}000} - 1\right) = 49.75.$$

The number 49.75 is approximated to 49, and its hexadecimal equivalent is 0x31. The following lines show the initialization of SSPADD register with 100kHz speed of data transmission/reception between 16F877 and 24LC16B, the 16Kbit serial EEPROM.

```
  movlw     0x31           ;100KHz speed
  banksel   SSPADD
  movwf     SSPADD         ;write baud rate of 100KHz
```

In case one wants to use the high speed data transmission of 400kHz, all he or she has to do is to change the content of SSPADD registers to 0x0B.

Lastly, we take care the SSPSTA register by putting 0x80.

```
  movlw     0x80      ;1000 0000
  banksel   SSPSTAT
  movwf     SSPSTAT   ;100KHZ (no slew rate control)
```

Writing a byte of data to the serial EEPROM

Now the initialization process is finished and the MSSP module is geared for 2-wire I^2C operation. For simplicity, we will first discuss how to write a byte of data into the serial EEPROM. The schematic for 24LC16B EEPROM connection to PIC 16F877 is illustrated below.

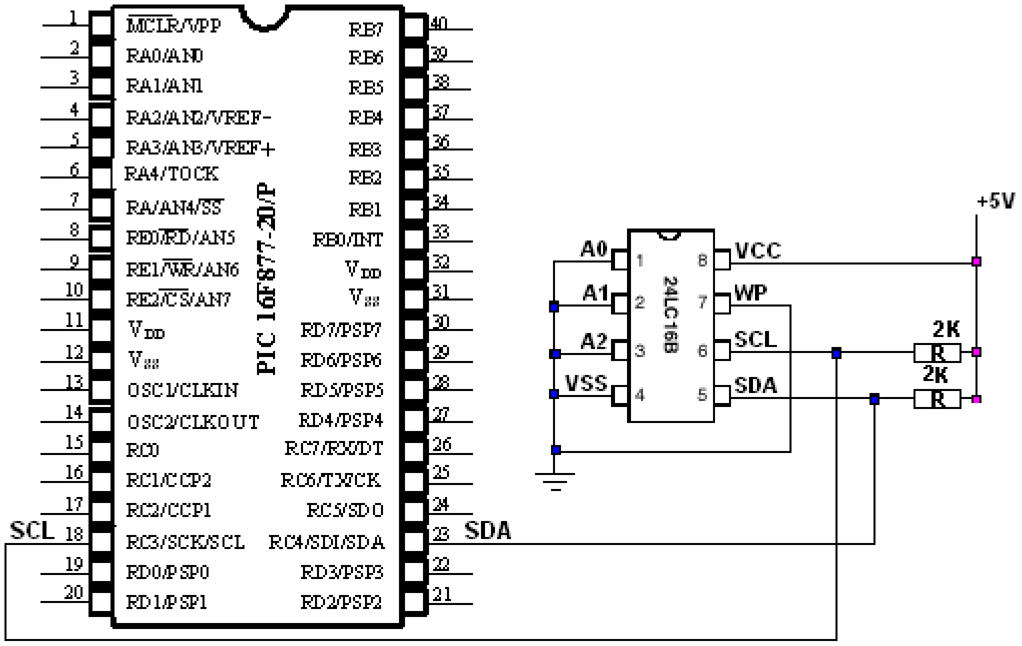


Fig. 89 24LC16B EEPROM connection to PIC 16F877

The byte writing takes numerous steps we discuss here and don't ask why these many steps: it's the combination of hardware and communication protocol, and much more. But if you follow all the steps described here, you surely get your 1 byte data at the designated address location inside the serial EEPROM.

Step 1: Clear SSPIF bit of the PIR1 register. SSPIF bit (synchronous serial port (SSP) interrupt flag), the bit number 3 of PIR1 register, PIR1<3>, is set when SSP interrupt condition occurred. This SSPIF is routinely monitored in the writing process since the flagging of SSPIF shares many different interrupt conditions. By monitoring this bit we can see the internal SSP module operational status and move to another step of byte writing operation. In I^{2}C operation in master mode, which is the mode we are applying in this example case, any of the following conditions sets SSPIF bit:

- A transmission or reception has taken place.
- The initiated start condition was completed by the SSP module
- The initiated stop condition was completed by the SSP module
- The initiated restart condition was completed by the SSP module
- The initiated acknowledge condition was completed by the SSP module

Once the SSPIF is set, it must be cleared by software (meaning that you have to clear the bit in the code). So it is apparent that we start our routine by clearing the SSPIF bit since no interrupt condition has occurred when we start.

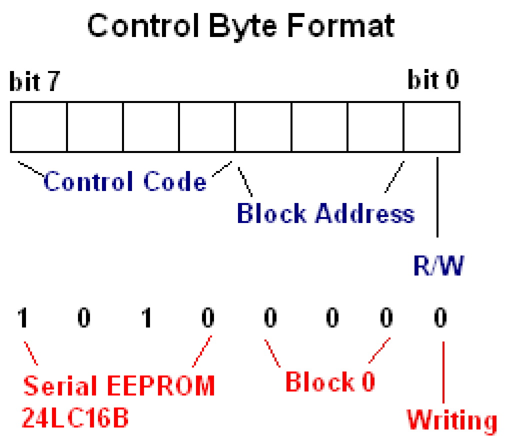
Step 2: Enable the start condition by setting the SEN bit of SSPCON2 register. The SEN (Start Condition Enable) bit, the bit 0 of SSPCON2 register, SSPCON2<0>, in I^{2}C master mode, when set, initiates Start condition on SDA and SCL pins. The clearing is done by hardware so we do not have to worry about it once we set to start I^{2}C operation.

Step 3: Wait until the SSPIF bit of PIR1 register is set, then clear it. As mentioned in the step 1, one of the several conditions that set the SSPIF bit is when the initiated start condition is completed by the SSP module. This step is nothing but waiting for the completion of the start condition initiated in step 2. When it is completed, SSPIF bit would be set, and therefore we, not the hardware, have to clear it. Simple enough. There may be a collision in the process when a writing is attempted before the start initiation is completed.

Step 4: Send 0xA0 to SSPBUF. This is to establish I2C protocol from 16F877 to 24LC16B. As we discussed in the section 3, a byte code must be sent followed by the start condition. The byte code consists of the 4-bit code for serial EEPROM, which is 1010, 3 bits of block selection, followed by 1 bit of read/write selection. The reason we put 000 for block address is, in the example code, we try to access the first block (block address 0) of the 8 block 24LC16B. The R/W bit is cleared since we are reading from the serial EEPROM. Again, the following illustration would serve to give you clear understanding of the step. Anyway, that's why we send 10100000 or its hexadecimal equivalent 0xA0 to the SSP buffer register SSPBUF.

Step 5: Wait until SSPIF bit of PIR1 is set, then clear it. The purpose of this step is similar to step 3. The only difference is, here, we wait until the transmission of the byte code in step is completed. This is, in other words, waiting for an acknowledgment from the serial EEPROM that it received the control byte. Now EEPROM is ready to send a byte data from a designated address which will be received in the next step.

Step 6: Load the address byte of the serial EEPROM to SSPBUF. This step is to select an address in the block we selected in the above steps. If the address to which write a byte data is 0, then we put 00000000 or 0x00 to the SSP buffer register SSPBUF.



Step 7: Wait until SSPIF bit of PIR1 is set, then clear it. Exactly the same purpose as step 5.

Step 8: Load the byte data into SSPBUF. Now finally we are sending the data byte we want to store to the serial EEPROM at the address 0 of the block 0.

Step 9: Wait until SSPIF bit of PIR1 is set, then clear it. Exactly the same purpose as step 7.

Step 10: Enable stop condition by setting the PEN (Stop Condition Enable) bit of SSPCON2 register. PEN bit, the second bit of SSPCON2, SSPCON2<2>, in I^2C master mode, when set, initiates the stop condition on SDA and SCL pins. It is automatically cleared when the stop condition is completed. So this closes the writing process by stopping the I^2C operation mode.

Step 11: Wait until SSPIF bit of PIR1 is set, then clear it. Exactly the same purpose as step 3. The only different here is that we are waiting the stop is completed instead of start.

Step 12: One last step ending the writing process: wait for at least 10ms so that 24LC16 has enough time of writing. As you see here, the writing process takes time and even it extends after the end of stop condition.

As we saw above, there are similar steps in the process, the waiting of SSPIF bit in particular. And it is apparent we do check the SSPIF bit after each step of sending start/stop condition, control byte, or writing a byte data. Therefore, we can include this SSPIF bit checking at the enc of each process, and we can reduce the number of steps.

Steps 1- 3 are then called a start event which send start condition for I^2C bus operation ("START"). Steps 4- 5 transmit control byte (code + block address + R/W) to the serial EEPROM and wait for acknowledgement from the serial EEPROM ("CONTROL"). Steps 6 - 7 transmit the address, where we are going to store a byte data, to the serial EEPROM, and wait for acknowledgment ("ADDRESS"). Steps 8 -9 transmit the 1-byte data to the serial EEPROM, and wait of acknowledgment ("DATA"). Steps 10 -11 transmit the stop condition so that we close the I^2C operation mode, and wait until the stop initiation is completed ("STOP"). Finally, step 12 is to give time to the serial EEPROM for actually writing the byte data into the memory space ("TIME"). Then, the simplified steps are in the order of: START>CONTROL>ADDRESS>DATA>STOP>TIME.

Now let's discuss about coding the byte writing process. Let's make a small subroutine for each step. First, the START step. We set the SEN bit of SSPCON2 register and wait for SSPIF of PIR1 to be set, and just to be cleared again. The START subroutine, `i2cStart` is shown below.

```
;subroutine I2CSTART =====
I2Cstart
     banksel   SSPCON2
     bsf       SSPCON2, SEN     ;START
     banksel   PIR1
Swait
     btfss     PIR1, SSPIF
     goto      Swait            ;wait for SSPIF to be set
     bcf       PIR1, SSPIF      ;then clear it
     return
```

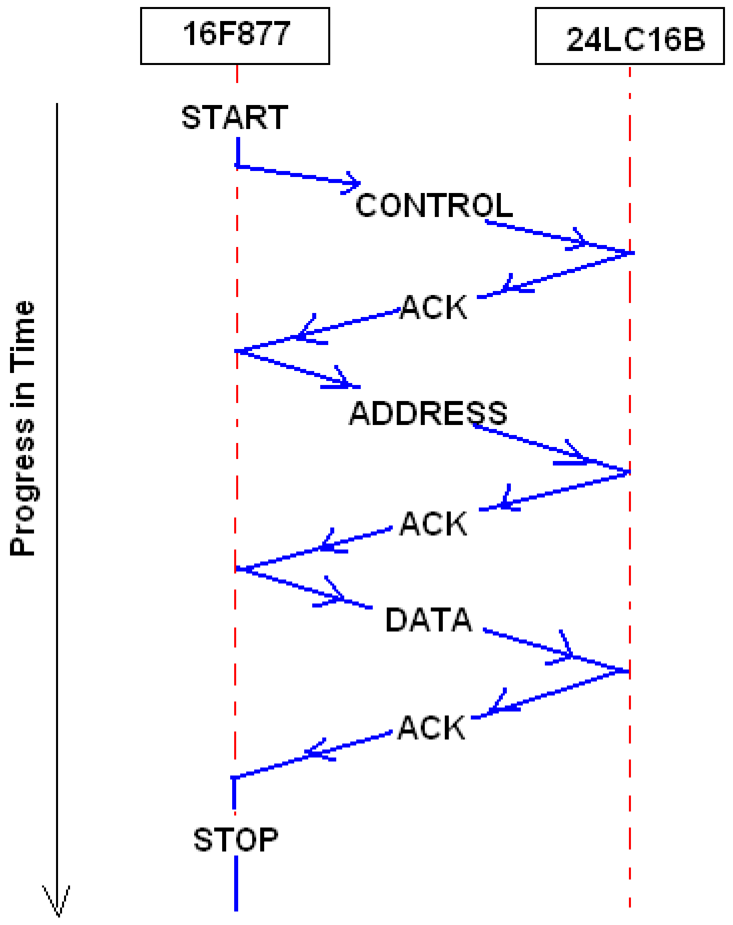
The CONTROL step transmits the control byte, which is in the example 0xA0, to indicate: (a) accessing serial EEPROM 24LC16B, (b)accessing the block 0 of the 8-block memory space, and (c) accessing in order to write. However, we notice that CONTROL, ADDRESS, and DATA steps are putting data/address/code into the SSP buffer register SSPBUF. So instead of making three separate subroutines, we make just one subroutine for all three steps called `i2cSend`. One thing you have to know here is you have to store the code/address/data into the **W** register before calling the `i2cSend` subroutine. In the subroutine, we put what's in the **W** register into SSPBUF and wait for the completion of the transmission. The `i2cSend` subroutine is as simple as indicated below.

```
;subroutine i2cSend for putting code/address/data to SSPBUF register
;Note: The code/address/data must be placed in W register
;before this subroutine is called.
i2cSend
     banksel   SSPBUF
     movwf     SSPBUF
     banksel   PIR1
cwait
     btfss     PIR1, SSPIF
     goto      cwait
     bcf       PIR1, SSPIF
     return
```

Then, the only step left is the STOP. The STOP is to set the PEN bit of SSPCON2 register, and of course, wait for the completion of the stop initiation. The STOP subroutine i2cStop is also simple enough.

```
;SUBROUTINE I2CSTOP ============================================
i2cStop
     banksel   SSPCON2
     bsf       SSPCON2, PEN
     banksel   PIR1
Pwait
     btfss     PIR1, SSPIF
     goto      Pwait
     bcf       PIR1, SSPIF
     return
```

Final word in byte writing before a complete code. The writing or reading (which is the subject of our discussion after the example code) is actually a protocol or handshaking between PIC 16F877 (as mater) and the serial EEPROM 24LC16B (as slave). The sequence and handshake are illustrated for better understanding the writing process to a serial EEPROM.



Ok. Let's jump to the actual code of the 1-byte data writing to address 0 of the block 0 of 24LC16B serial EEPROM. Since we already discussed about all the necessary subroutines for the writing process, we ignore them in the code listed below.

```
;I2Cwrite.asm
;
;i2C Master Mode Program
;for use in 24LC16B Serial EEPROM access by 16F877
;in I2C Master Mode Operation of SSP Module of 16F877 (with Fosc = 20 MHz )
;
;1-byte data writing to 24LC16B
;
;
  list      p=16f877

INTCON    EQU  0x0B
STATUS    EQU  0x03
ZERO      EQU  0x00   ;for STATUS
PIR2      EQU  0x0D   ;I2C Modules
PIR1      EQU  0x0C
SSPIF     EQU  0x03   ;SSP Interrupt Flag (must be cleared for next I2C)
PIE1      EQU  0x8C
PIE2      EQU  0x8D
```

```
SSPCON    EQU  0x14
SSPEN     EQU  0x05   ;SSP Enable bit
SSPCON2   EQU  0x91
PEN       EQU  0x02
SEN       EQU  0x00
SSPSTAT   EQU  0x94
SMP       EQU  0x07   ;SMP=1 for 100KHz, SMP=0 for 400KHz  (Choose 100KHZ)
SSPADD    EQU  0x93
baud100   EQU  0x31 ;100KHZ standard speed
baud400   EQU  0x0B ;400KHz fast speed  ($0B)
TRISC     EQU  0x87
SSPBUF    EQU  0x13
;
;

     org    0x0000           ;line 1
     goto   START
     org    0x05
START
;i2c operation INITIALIZATION
;PORTC setup  - SDA and SCL both as inputs
     movlw  0x18      ;0001 1000
     banksel TRISC
     movwf  TRISC       ;RC4 (SDA) and RC3(SCL) as inputs
;MSSP Module Setup
     movlw  baud100     ;100KHz speed
     banksel SSPADD
     movwf  SSPADD      ;write baud rate
;
     movlw  0x80      ;1000 0000
     banksel SSPSTAT
     movwf  SSPSTAT   ;100KHZ (no slew rate control)
          ;selection with I2C mode
;
     movlw  0x28      ;0010 1000 (SSPEN=1, SSPM3:0= 1000 ) master mode
     banksel SSPCON
     movwf  SSPCON

;============================================================================
;WRITING for a byte data
;Sequence
;1. Start event <S>
;2. Send Control Byte for EPEROM with Write Op <C>
;    CONTROL BYTE STRUCTURE OF 24LC16B
;    1   0   1   0   A2   A1   A0  R/W
;    The first 4-bit code is establisihed for serial EEPROM by the I2C Bus
Specification
;    Next 3 bits are for Block Select  (26LC16B has 8 blocks, with each block
having 256 Bytes)
;    R/W = 1 for reading
;    R/W = 0 for writing
;3. Wait for ACK from EEPROM (Use SSPIF of PIR1)<K>
;4. Send the address in a block of EEPROM<A>
;5. Wait for ACK from EEPROM<K>
;6  Send a byte of data<D>
;7. Wait for ACK from EEPROM <K>
;8. STop EVENT<P>
```

```
;9. Give 24LC16 a few ms to write the data
;=====================================================================
; SEQUENCE IN WRITING
; PIC (master Side)>S>> >C>>    >A>>      >D>>       >P>>
; 24LC16 (slave side)            <<K<     <<K<       <<K<
;=====================================================================
;
;Flag Clear
     banksel   PIR1
     bcf       PIR1, SSPIF    ;clear the SSP flag

;>S>> START
     call      i2cStart
;>C>>  CONTROL
     movlw     0xA0
     call      i2cSend   ;block code write to 24LC16
;>A>> ADDRESS
     movlw     0x00      ;address info
     call      i2cSend
;>D>> DATA
     movlw     'm'       ;1 byte data
     call      i2cSend
;>P>> STOP
     call      i2cStop

     call      delay10ms   ;Give 24LC16B time to write the data

;your subroutines here

;your subroutines here
     END
```

<u>Reading Randomly 1-byte Data from serial EEPROM</u>

The first three steps of writing are used without any change in the reading process. In other words, we proceed START>CONTROL>ADDRESS as we did above. Then, comes the RESTART step.

In the RESTART step, we set the RSEN bit of SSPCON2 register. The RSEN (Repeated Start Condition Enable) bit, in I^{2}C mater mode, when set, initiates repeated start condition on SDA and SCL pins. This bit will be cleared automatically by hardware. Why do we need this RESTRAT in the reading process? My best answer is "Accept it just as a part of 2-wire I^{2}C protocol." The RESTART subroutine, `i2cRstart` is shown below.

```
;subroutine I2CRSTART (Repeated Start. Use in READING )
i2cRstart
     banksel   SSPCON2
     bsf       SSPCON2, RSEN    ;REPEATED START
     return
```

After RESTART, according to the specification of I2C protocol with 24LC16B, we have to wait for 100us. Then, comes the CONTROL_R step.

This CONTROL_R is the same as CONTROL except that the R/W bit (the LSB of the control byte) must be set to indicate that we intend to read from the serial EEPROM. Therefore, we can use the same `i2cSend` subroutine. The value to be sent to SSPBUF is then 10100001 (or 0xA1) this time. However, this CONTROL_R step unlike the CONTROL does not need acknowledgment from 24LC26B. By the way, the above step of RSTART does not need acknowledgment either. In CONTROL_R, we give a short time delay so that this process finishes without collision. Therefore instead of using the subroutine, we can simply have the following lines to satisfy the COTROL_R step:

```
;>C>> CONTROL_R
; Control Byte = 10100001=0xA1 for READ
     movlw     0xA1
     banksel   SSPBUF
     movwf     SSPBUF
     call      delay100us
```

The next step is the actual READ step. Here we have enable the receive condition be setting the RCEN bit of SSPCON2 register. RCEN (Receive Enable) bit, the 3rd bit of SSPCON2, SSPCON2<3>, in I^{2}C master mode, when set, enables receive mode for I^{2}C. After initiating the receive mode, we have to make sure if the initiation is completed by monitoring the SSPIF bit of SSPCON2 as usual. The READ subroutine `i2cRead` is shown below.

```
;SUBROUTINE i2cREAD READ event =============
i2cRead
     banksel   SSPCON2
     bsf       SSPCON2, RCEN
     banksel   PIR1        ;read event completion check
Rwait  btfss   PIR1, SSPIF
     goto      Rwait
     bcf       PIR1, SSPIF
     return
```

Then comes the NACK step. This step is necessary since, after reading a byte from 24LC16B, the serial EEPROM waits for acknowledgment from master, 16F877. This acknowledgment process is initiated by setting the ACKEN (Acknowledge sequence enable: SSPCON2<4>) bit of SSPCON2, which initiates the acknowledge sequence on SDA and SCL pins. However, since we are reading, randomly (meaning: you can read a byte data from any address you set in the ADDRESS step), only one byte, there is no further connection is necessary between 16F877 and 24LC16B. Therefore, we just disconnect without actually acknowledging. This "Not Acknowledge" is enabled by setting the ACKDT (Acknowledge Data: SSPCON2<5>) of the SSPCON2 register. This whole thing is realized by a step NACK. And the subroutine for NACK, `i2cNack` is shown below. As you see below, it monitors the SSPIF bit for the completion of the initiation.

```
;subroutine
i2cNack
     banksel   SSPCON2
     bsf       SSPCON2, ACKDT
     bsf       SSPCON2, ACKEN
     banksel   PIR1
```

```
nwait  btfss  PIR1, SSPIF
     goto     Nwait
     bcf      PIR1, SSPIF
     return
```

The last step is the STOP step we used in the writing process. We use the same subroutine `i2cStop` here without change. Then, we move the content stored in the SSPBUF register to **W** register, and this content is the data retrieved from the serial EEPROM.

Writing and Reading a Byte to/from 24LC16B - A complete code

Without printing (or displaying) the content retrieved from 24LC16B, we do not know if our reading process is correct or incorrect. Therefore, even though cumbersome it is, we have to add (or borrow from previous chapters) serial communication routine to send the data to the screen of PC. So the code shown below write letter 'm' into the address 0 of the block 0, then read from the address 0 of the block 0, and print out the retrieved letter to the computer screen. If you see 'm' on the screen, you know your code is properly working. Then, you can revise the code so that you send as many bytes of data as possible, and read them all or just a few of them.

```
;I2C.asm
;
;i2C Master Mode Program
;for use in 24LC16B Serial EEPROM access by 16F877
;in I2C Master Mode Operation of SSP Module of 16F877 (with Fosc = 20 MHz )
;
;
; I2C Master Mode Operation for EEPROM (slave) Access
; EEPROM SPEC:
;    1.Microchip 24LC16B  (16K Bits or 2KBytes)
;    2.8 Internal blocks (256 Bytes for each block)
; I2C ADDRESS SPEC:
;    1. I2C Spec code for EEPROM SELECTION: 1010 must be the highest 4 Bits in
;       the Higher Address
;    2. 3-bit Block Code (from 000 to 111) follows the EEPROM SELECTION bits
;   3. R/W bit follows to Complete the 8-bit Higher Address
;   4. Actual EEPROM address is selected by the 8-bit Lower Address
;
;
; ==================================================================

   list      p=16f877


INTCON    EQU  0x0B
STATUS    EQU  0x03
ZERO      EQU  0x00   ;for STATUS
PIR2      EQU  0x0D   ;I2C Modules
BCLIF     EQU  0x03   ;Bus Interrupt Flag
PIR1      EQU  0x0C
SSPIF     EQU  0x03   ;SSP Interrupt Flag (must be cleared for next I2C)
PIE1      EQU  0x8C
PIE2      EQU  0x8D
BCLIE     EQU  0x03   ;bUS iNTERRUPT Enable
```

```
SSPCON    EQU  0x14
WCOL      EQU  0x07   ;Write Collision detect
SSPSOV    EQU  0x06   ;Receive Overflow Indicator
SSPEN     EQU  0x05   ;SSP Enable bit
SSPCON2   EQU  0x91
ACKSTAT   EQU  0x06
RCEN      EQU  0x03
PEN       EQU  0x02
RSEN      EQU  0x01
SEN       EQU  0x00
ACKEN     EQU  0x04
ACKDT     EQU  0x05
SSPSTAT   EQU  0x94
SMP       EQU  0x07   ;SMP=1 for 100KHz, SMP=0 for 400KHz  (Choose 100KHZ)
SSPADD    EQU  0x93
baud100   EQU  0x31 ;100KHZ standard speed
baud400   EQU  0x0B ;400KHz fast speed  ($0B)
TRISC     EQU  0x87
SSPBUF    EQU  0x13
;
TXSTA     EQU  0x98   ;TX status and control
RCSTA     EQU  0x18   ;RX status and control
SPBRG     EQU  0x99   ;Baud Rate assignment
TXREG     EQU  0x19   ;USART TX Register
RXREG     EQU  0x1A   ;USART RX Register
PIR1      EQU  0x0C   ;USART RX/TX buffer status (empty or full)
RCIF      EQU  0x05   ;PIR1<5>: RX Buffer 1-Full  0-Empty
TXIF      EQU  0x04   ;PIR1<4>: TX Buffer 1-empty 0-full
TXMODE    EQU  0x20   ;TXSTA=00100000 : 8-bit, Async
RXMODE    EQU  0x90   ;RCSTA=10010000 : 8-bit, enable port, enable RX
BAUD      EQU  0x0F   ;0x0F (19200), 0x1F (9600)

;
;
;
  CBLOCK 0x20
    temp
    Kount20us
    Kount120us   ;Delay count (number of instr cycles for delay)
    Kount100us
    Kount1ms
    Kount10ms
    Kount1s
    Kount10s
    Kount1m
  ENDC
;==========================================================

     org    0x0000        ;line 1
     goto   START
     org    0x0005
START
     call   ASYNC_MODE  ;for RS232

;i2c operation
;PORTC setup  - SDA and SCL both as inputs
  movlw     0x18      ;0001 1000
  banksel   TRISC
  movwf     TRISC       ;RC4 (SDA) and RC3(SCL) as inputs
;MSSP Module Setup
  movlw     baud100     ;100KHz speed
  banksel   SSPADD
  movwf     SSPADD      ;write baud rate
;
  movlw     0x80      ;1000 0000
  banksel   SSPSTAT
  movwf     SSPSTAT   ;100KHZ (no slew rate control)
          ;selection with I2C mode
;
  movlw     0x28      ;0010 1000 (SSPEN=1, SSPM3:0= 1000 ) master mode
  banksel   SSPCON
  movwf     SSPCON

;============================================================================
;WRITING for a byte data
;Sequence
;1. Start event <S>
;2. Send Control Byte for EPEROM with Write Op <C>
;    CONTROL BYTE STRUCTURE OF 24LC16B
;    1   0   1   0   A2   A1   A0  R/W
;    The first 4-bit code is establisihed for serial EEPROM by the I2C Bus
Specification
;    Next 3 bits are for Block Select  (26LC16B has 8 blocks, with each block
having 256 Bytes)
;    R/W = 1 for reading
;    R/W = 0 for writing
;3. Wait for ACK from EEPROM (Use SSPIF of PIR1)<K>
;4. Send the address in a block of EEPROM<A>
;5. Wait for ACK from EEPROM<K>
;6  Send a byte of data<D>
;7. Wait for ACK from EEPROM <K>
;8. STop EVENT<P>
;9. Give 24LC16 a few ms to write the data
;=====================================================================
; SEQUENCE IN WRITING
; PIC (master Side)>S>> >C>>    >A>>      >D>>       >P>>
; 24LC16 (slave side)            <<K<     <<K<       <<K<
;=====================================================================
;
;Flag Clear
  banksel   PIR1
  bcf       PIR1, SSPIF   ;clear the SSP flag

;>S>>
  call      i2cStart
;>C>>
  movlw     0xA0
  call      i2cSend   ;block code write to 24LC16
;>A>>
  movlw     0x00      ;address info
  call      i2cSend
;>D>>
  movlw     'm'       ;1 byte data
  call      i2cSend
```

```
;>P>>
  call     i2cStop

  call     delay10ms

; END OF WRITING
;===============================================================
;Sequence of Data (random) Read operation

; >S>>  >C>>    >A>>      >RS>>   >C>>      <<D<   >P>>
;           <<K<     <<K<            <<K<
;1. Start Event >S>>
;2. Send Control Byte to Slave (with RW=0 i.e., write) >C>>
;3. Check for ACK from slave  <<K<
;4. send actual EEPROM address in a block  >A>>
;5. Wait for ACK  <<K<
;6. Repeated Start Event   >RS>>
;7.Send Control Byte to EEPROM again (with RW=1 i.e. now we read) >C>>
;8. wait for ACK  <<K<
;9. Read a byte of data <<D<
;10. STop event  >P>>

;================================================================

;>S>>
  call     i2cStart
;>C>>
  movlw    0xA0
  call     i2cSend
;>A>>
  movlw    0x00
  call     i2cSend

;>RS>> REPEATED START
  call     i2cRstart
  call     delay100us
;>C>>
; Control Byte = 10100001=0xA1 for READ
  movlw    0xA1
  banksel  SSPBUF
  movwf    SSPBUF
  call     delay100us
;<<D<
  call     i2cRead

; Send NACK to 24LC16
  call     i2cNack

;>P>>
  call     i2cStop

  banksel  SSPBUF
  movf     SSPBUF,0    ;EEPROM data --> W
  call     TXpoll
  movlw    0x0D   ;CR
  call     DTXpoll
  movlw    0X0A   ;LF
  call      TXpoll

FINI        nop
  goto      FINI      ;idling


;subroutine I2CSTART =====
I2Cstart
  banksel   SSPCON2
  bsf       SSPCON2, SEN     ;START
  banksel   PIR1
Swait  btfss  PIR1, SSPIF
  goto      Swait
  bcf       PIR1, SSPIF
  return
;---------------------------------------------------
;subroutine i2cblock for write

i2cSend
  banksel   SSPBUF
  movwf     SSPBUF
  banksel   PIR1
cwait  btfss  PIR1, SSPIF
  goto      cwait
  bcf       PIR1, SSPIF
  return
;-------------------------------------------------
;subroutine I2CRSTART (Repeated Start. Use in READING =====
i2cRstart
  banksel   SSPCON2
  bsf       SSPCON2, RSEN    ;REPEATED START
  return
;---------------------------------------------------

;SUBROUTINE i2cREAD READ event =============
i2cRead
  banksel   SSPCON2
  bsf       SSPCON2, RCEN
  banksel   PIR1        ;read event completion check
Rwait  btfss  PIR1, SSPIF
  goto      Rwait
  bcf       PIR1, SSPIF
  return
;----------------------------------------------
;SUBROUTINE I2CSTOP =============================================
i2cStop
  banksel   SSPCON2
  bsf       SSPCON2, PEN
  banksel   PIR1
Pwait  btfss  PIR1, SSPIF
  goto      Pwait
  bcf       PIR1, SSPIF
  return
;----------------------------------------
;subroutine
i2cNack
  banksel   SSPCON2
  bsf      SSPCON2, ACKDT
  bsf      SSPCON2, ACKEN
  banksel  PIR1
nwait  btfss  PIR1, SSPIF
  goto     Nwait
  bcf      PIR1, SSPIF
  return
;--------------------------------
;==subroutine TXpoll
;W contains the data to be transferred to PC
TXpoll
  banksel  PIR1
  btfss    PIR1, TXIF
  goto     TXpoll   ;if full, wait
  banksel  TXREG
  movwf    TXREG
  return

;========================================
Async_mode
  banksel  SPBRG
  movlw    BAUD
  movwf    SPBRG    ;Baud Rate Setup
  movlw    TXMODE
  movwf    TXSTA
  banksel  RCSTA
  movlw    RXMODE
  movwf    RCSTA
  return
;=========================================
;==========================================================
;DELAY SUBROUTINES


; 1 instruction cycle for 20MHz clock is 0.2 us
; Therefore 120 uS delay needs 600 instuction cycles
;  600 =199*3 +3 ---->Kount=199=0xC7
;  or  =198*3 +6 ---->Kount=198=0xC6
;  or  =197*3 +9 ---->Kount=197=0xC5

Delay20us
  banksel  Kount20us
  movlw    H'1F'    ;D'31'
  movwf    Kount20us
R20us  decfsz Kount20us
  goto     R20us
  return
;
;
Delay120us
  banksel  Kount120us
  movlw    H'C5'    ;D'197'
  movwf    Kount120us
R120us decfsz Kount120us
  goto     R120us
  return
;
```

```
;100us delay needs 500 instruction cycles
;  500 =166*3 +2 ---->Kount=166=0xA6
;  or  =165*3 +5 ---->Kount=165=0xA5
;  or  =164*3 +8 ---->Kount=164=0xA4
Delay100us
  banksel   Kount100us
  movlw     H'A4'
  movwf     Kount100us
R100us decfsz Kount100us
  goto      R100us
  return

;
;10ms delay
; call 100 times of 100 us delay  (with some time discrepancy)
Delay10ms
  banksel   Kount10ms
  movlw     H'64'  ;100
  movwf     Kount10ms
R10ms  call delay100us
  decfsz    Kount10ms
  goto      10ms
  return
;
;=====================================================

  END
```