



I²C™ Master Mode

Overview and Use of the
PICmicro® MSSP I²C Interface
with a 24x01x EEPROM

v 0.40

Getting Started: I²C Master Mode

© 2001

Welcome to the Microchip Technology Presentation on using the MSSP module in Master I²C mode.

In this presentation, we will look at what I²C is and how it is used to communicate data to and from a PICmicro Microcontroller and a serial EEPROM. We will be connecting a device from the popular family of PIC16F87x microcontrollers to a 24x01 serial EEPROM. Both of these devices are manufactured by Microchip Technology Inc and can be found on the popular PICDEM 2 Demonstration Board.

I²C is a popular protocol and is supported by many devices. This presentation answers some questions about I²C and explains with a full example how to connect a PICmicro MSSP module to an EEPROM.

- Covered Topics:
 - Overview of I²C
 - Using I²C on the PICmicro Microcontroller
 - Example: A code walk-through for connecting a 24x01x to a PIC16F87X
 - Finding More Information

In this presentation, we will cover the following topics:

We will first cover an Overview of I²C.

This chapter of the presentation will introduce you to the I²C Protocol and its concepts.

Next, we will examine the use of I²C on the PICmicro microcontroller.

The details of how SPI is implemented on a PICmicro device will be examined. We will look at the MSSP module, which is available on a wide selection of popular PICmicro microcontrollers.

We will then examine a code walk-through.

The walkthrough will explore code for both writing and reading a serial EEPROM. The example sends sample data to the EEPROM, then reads back the data and displays it. The code to do this looks rather long, but it is not complex. We will break the code down into smaller and easy to understand sections.

Finally, there will be a few resources given at the end of the presentation. These resources will allow you to explore in more detail the I²C interface.



Overview

- Used for moving data simply and quickly from one device to another
- Serial Interface
- Synchronous
- Bidirectional

Getting Started: I²C Master Mode

© 2001

I²C stands for Inter-Integrated Circuit Communications.

I²C is implemented in the PICmicro by a hardware module called the Master Synchronous Serial Port, known as the MSSP module . This module is built into many different PICmicro devices. It allows I²C serial communication between two or more devices at a high speed and communicates with other PICmicro devices and many peripheral IC's on the market today.

I²C is a synchronous protocol that allows a master device to initiate communication with a slave device. Data is exchanged between these devices. We will look at this more in detail as we progress through this presentation.

I²C is also bi-directional. This is implemented by an "Acknowledge" system. The "Acknowledge" system or "ACK" system allows data to be sent in one direction to one item on the I²C bus, and then, that item will "ACK" to indicate the data was received. We will look at this in detail later, as you can see, this is a powerful feature of I²C. Since a peripheral can acknowledge data, there is little confusion on whether the data reached the peripheral and whether it was understood.

- I²C is a **Synchronous** protocol
 - The data is clocked along with a clock signal (SCL)
 - The clock signal controls when data is changed and when it should be read
 - Since I²C is synchronous, the clock rate can vary, unlike asynchronous (RS-232 style) communications

I²C is a synchronous protocol that allows a master device to initiate communication with a slave device. Data is exchanged between these devices.

Since I²C is synchronous, it has a clock pulse along with the data. RS232 and other asynchronous protocols do not use a clock pulse, but the data must be timed very accurately.

Since I²C has a clock signal, the clock can vary without disrupting the data. The data rate will simply change along with the changes in the clock rate. This makes I²C ideal when the micro is being clocked imprecisely, such as by a RC oscillator.

- I²C is a Master-Slave protocol
 - The Master device controls the clock (SCL)
 - The slave devices may hold the clock low to prevent data transfer
 - No data is transferred unless a clock signal is present
 - All slaves are controlled by the master clock

I²C is a Master-Slave protocol.

Normally, the master device controls the clock line, SCL. This line dictates the timing of all transfers on the I²C bus. Other devices can manipulate this line, but they can only force the line low. This action means that item on the bus can not deal with more incoming data. By forcing the line low, it is impossible to clock more data in to any device. This is known as “Clock Stretching”.

As stated earlier, no data will be transferred unless the clock is manipulated.

All slaves are controlled by the same clock, SCL.



I²C - Overview

- I²C is a **Bidirectional** protocol
 - Data is sent either direction on the serial data line (SDA) by the master or slave.

I²C is a Bi-directional protocol. Data can flow in any direction on the I²C bus, but when it flows is controlled by the master device.

- I²C is a **Serial Interface** of only two signals:
 - **SDA** **Serial DATA**
This line transfers data to or from the master.
 - **SCL** **Serial CLOCK**
This controls when data is sent and when it is read. The master controls SCL.

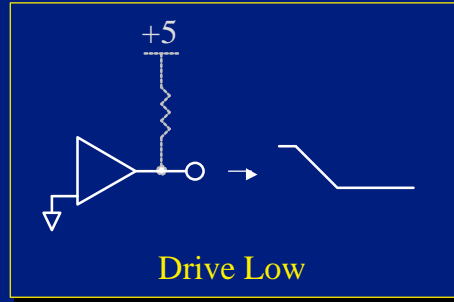
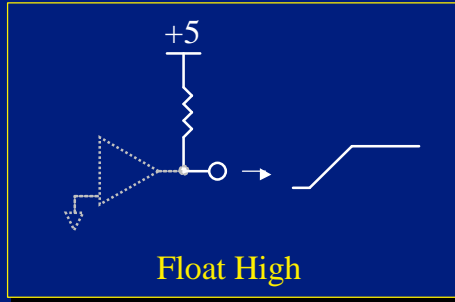
I²C is a Serial Interface and uses only the following two signals to serially exchange data with another device:

SDA - This signal is known as Serial Data. Any data sent from one device to another goes on this line.

SCL - This is the Serial Clock signal. It is generated by the master device and controls when data is sent and when it is read. As mentioned earlier, the signal can be forced low so that no clock can occur. This is done by a device that has become too busy to accept more data.

- **Signal Levels**

- Float High (logic 1)
- Drive Low (logic 0)



Note: Diagrams are symbolic

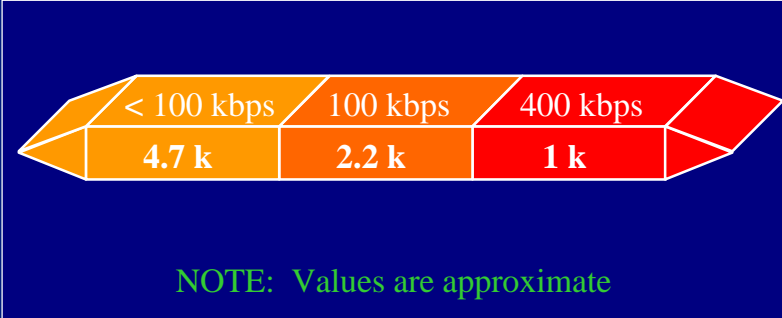
I²C lines can have only two possible electrical states. These states are known as “float high” and “drive low”. I²C works by having a pull-up resistor on the line and only devices pull the line low. If no device is pulling on the line, it will “float high”. This is why pull-up resistors are important in I²C.

If no pull-up resistor were used, the line would float to an unknown state. If one tried to drive the line high, it might cause contention with a device trying to drive the line low. This contention could damage the either or both devices driving the line.

To prevent this, the pull-up-drive low system controls when one device has control of the bus. If another device tried to use the bus when it was busy, it would find the bus to be driven low already and know it was busy. Even if it tried to use the bus accidentally, it would only drive it low and not damage other devices.

The diagrams shown are symbolic. In each case, the solid diagram represents the ACTIVE part of the bus. In the case of driving low, the buffer is actively pulling the line low. In the case of floating high, the resistor pulls the line high, while the buffer is turned off. A buffer turned off has very high impedance and behaves as if it were disconnected. Only the output buffers are shown for simplicity.

I²C Pull-up Resistor Setting Suggestions



I ² C Speed	Recommended Resistor Value
< 100 kbps	4.7 k
100 kbps	2.2 k
400 kbps	1 k

NOTE: Values are approximate

This diagram represents the recommended pull-up resistor value for various I²C speeds. You are free to use any resistor value you like, but the calculation of what to use will depend on the capacitance of the driven line, and the speed of the I²C communication. There may be other factors as well. These values were chosen as they represent values that have been found to work frequently at these speeds. They are provided for reference only as suggested values. Your application may choose other values.

Next, we will examine the building blocks or “elements” of I²C

- Building Blocks of I²C
 - I²C consists of many “conditions” which to simplify this presentation will be represented as “elements”.



The I²C bus has a number of “conditions”. These conditions indicate when a transfer is starting, stopping, being acknowledged, and other events. To simplify the explanation of I²C communications, this presentation will represent these as “elements”, small colored blocks with a letter and color to represent each condition.

These elements will be used throughout the presentation to aid the explanation of I²C. Some sample blocks are shown here.

[pause]

Let's take a closer look at these elements now.

- Start Condition
 - Initializes I²C Bus
 - SDA is pulled low while SCL is high

S

=



The first element we need to look at is the *Start* condition. A start condition indicates that a device would like to transfer data on the I²C bus.

Pictured here is the block with an “S” in it and what the signals look like on the I²C bus. As you can see, SDA is first pulled low, followed by SCL.

The PICmicro microcontroller will take care of the timing details for you. However it will need to be told you want a start condition and you will check for when it completes. We will look at how these blocks relate to using a PICmicro device later.

- Stop Condition
 - Releases I²C Bus
 - SDA is released while SCL is high



The next element we will discuss is the *Stop* condition. A start condition indicates that a device has finished its transfer on the I²C bus and would like to release the bus. Once released other devices may use the bus to transmit data.

As you can see, a block with a “T” in it represents the stop condition. A “T” is used because “S” was already used for start earlier. This convention will continue to be used throughout this presentation.

The signaling used for a stop is a release of the SCL line followed by a release of the SDA line. Remember that releasing a line turns off the driver, and since there is a pull-up resistor on it, the line floats high.

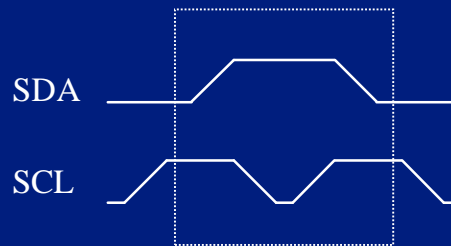
Once the stop condition completes, both SCL and SDA will be high. This is considered to be an *idle* bus. Once the bus is idle a Start condition can be used to send more data.

Again, the PICmicro microcontroller will take care of the timing details of this for you. You will only need to tell it you want a stop condition and wait for it to complete.

- Restart Condition
 - Reinitializes I²C Bus
 - Used when START does NOT follow STOP

R

=



Next is the *Restart* condition. A restart condition indicates that a device would like to transmit more data, but does not wish to release the line. This is done when a start must be sent, but a stop has not occurred. It is also a convenient way to send a stop followed by a start right after each other. It prevents other devices from grabbing the bus between transfers.

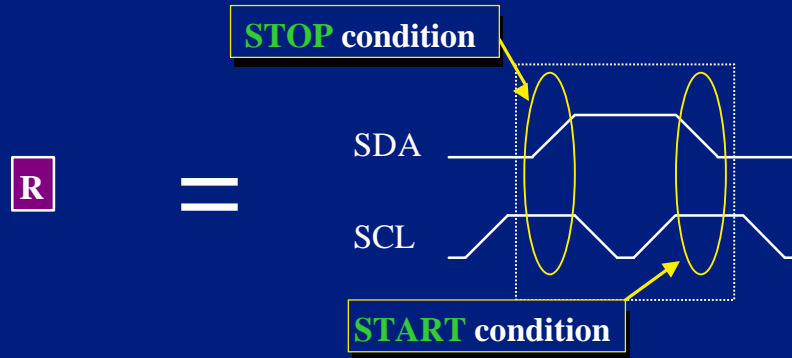
If you are talking to one device, such as a serial EEPROM, you may not want to be interrupted when transmitting addresses and gathering data. A restart condition will handle this.

The restart condition is represented by a “R” in this presentation.

The signaling used for a restart can be seen to be nothing more than a stop condition quickly followed by a start condition.

The PICmicro microcontroller also will handle this. You simply request a restart condition be sent, then wait for it to complete.

- Restart Condition
 - Reinitializes I2C Bus
 - Used when START does NOT follow STOP



Here we can clearly see that the signaling used for a restart can be seen to be nothing more than a stop condition quickly followed by a start condition. Remember that a stop condition is when SDA goes high while SCL is high. A start condition is when SDA is pulled low while SCL is high.

The PICmicro microcontroller automatically generate this as well. One simply requests that a restart condition be sent, then wait for it to complete.

- **Data Transfer**
 - 8 bits of data is sent on the bus
 - Data is valid when SCL is high



Let's now discuss the data transfer element. The data block represents the transfer of 8 bits of information. The data is sent on the SDA line and SCL produces a clock. The clock can be aligned with the data to indicate whether each bit is a "1" or a "0".

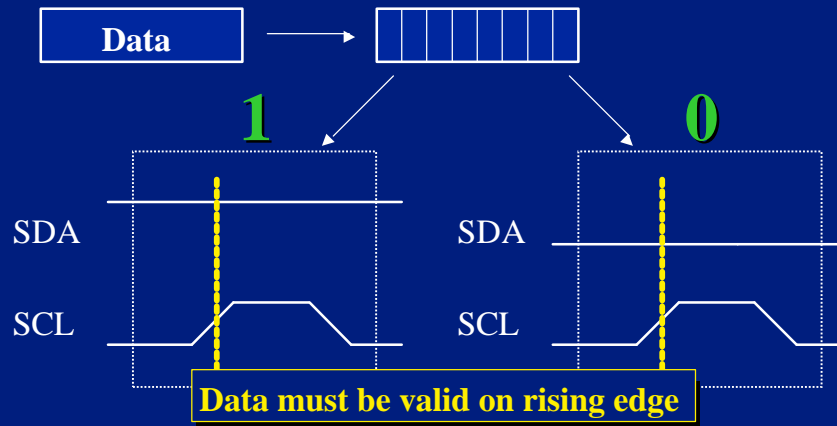
Data on SDA is only considered valid when SCL is high. When SCL is not high, the data is permitted to change. This is how the timing of each bit works.

The PICmicro microcontroller also can transmit data bytes. To do so, we load a buffer with the byte of data to send, tell it to send it and wait for its completion.

Data bytes are used to transfer all kinds of information. When communicating to another I²C device, the 8 bits of data may be a control code, an address or data. Many possibilities exist and they will be discussed in detail in the manual for the device you are interfacing to. In this presentation we will connect a serial EEPROM to the bus and look at the signals involved. Other I²C devices will require similar signals, but may not be identical. Check the device datasheet for the peripheral.

- Data States

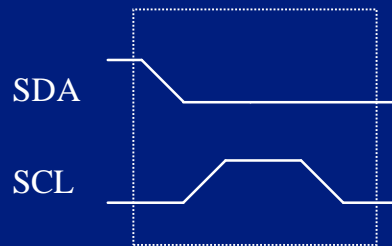
- Each bit of data can be a “1” or “0”



Here is a close-up view of a data block. As you can see, it contains 8 bits of data and the data is valid on the rising edge of SCL. The data then remains valid while SCL is high.

If SDA is high when this happens, the data bit is a “1”. If it is low, it is a “0”. We will see sample data transfers later in this presentation when we look in detail at our example.

- ACK Condition
 - Acknowledges a data transfer
 - ACK is when the recipient drives SDA low



Lastly we will discuss the *ACK* and *NACK* condition. A device can “ACK” or acknowledge a transfer of each byte by bringing the SDA line low during the 9th clock pulse of SCL.

The 9 bits of a transfer look like this: 8 bits are clocked out for the data, then during the 9th bit the item receiving the data grabs the bus for one bit. If it drives this bit low, then the device is signaling an “ACK”. Otherwise, if it allows the SDA line to float high it is transmitting a “NACK”. Remember that the device must actively drive the bus low to send an ACK, but a NACK could be a passive response. This is one of the benefits of I²C.

This diagram shows an “ACK” element. It is shown as a block with an “A” in it.

- **NACK Condition**
 - Negatively acknowledges a data transfer
 - NACK - the recipient does NOT drive SDA low

N


=



This is a *NACK* condition. Remember that a “ACK”s or “NACK”s a byte of data. I²C states that each byte **MUST** be answered with a NACK or ACK. If the device can not decide if it wants to ACK or NACK, then it will hold the clock line low until it makes up its mind. This action is known as “clock stretching” and is a feature of I²C to give devices enough time to respond. We will look at this in more detail later.

Notice that a “NACK” is when the SDA line floats high during the 9th clock pulse. It is the opposite of an ACK. The meaning of these acknowledgements will depend on which byte is being transferred and what device is being talked to.

This diagram shows an “NACK” element. It is shown as a block with an “N” in it.



Writing to a I²C EEPROM

- Write Example

S

Control In

Address

Data

P

A

A

A

From Master

From Slave

Getting Started: I²C Master Mode © 2001

If we put these elements together, we can produce useful I²C transfers. The aim of this presentation is to communicate with a serial I²C EEPROM. Here is an example transfer of *writing* to a small EEPROM.

We need to transfer 3 bytes of information to do this. The transfer begins with a *start*, to signal the beginning of the transfer. Then, the control byte is sent. The control byte for an EEPROM can have two different data bytes in it. One signifies that you want to write a byte to the EEPROM, and the other signifies that we want to read a byte from the EEPROM.

The function of *writing* to the EEPROM is shown here as “Control *IN*”, which represents putting the EEPROM in an “input” mode. Since we are only sending data *to* the EEPROM, we use the “Control In” byte. We will use “Control *OUT*” later.

Next, the EEPROM acknowledges this byte. This is shown by the “A” after the byte. It is put on the next line to indicate this is transmitted by the EEPROM, not the PICmicro device.

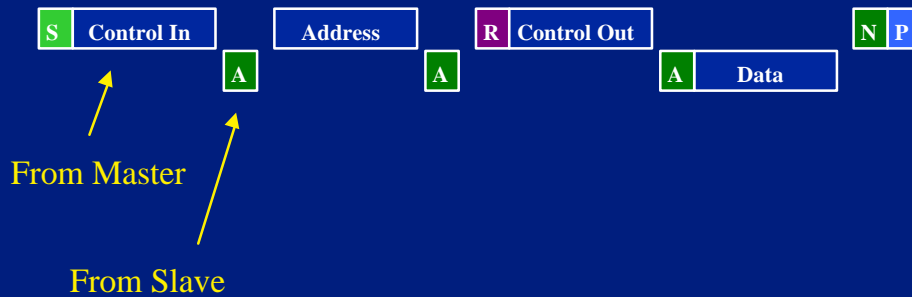
Next the PICmicro sends the Address Byte. The Address Byte contains the address of the location of the EEPROM we want to write data to. Since the address is valid, the data is “ACK”ed by the EEPROM.

Finally, we send the data we want to write. The data is then ACK’ed by the EEPROM. When that finishes, we send a stop condition to complete the transfer. Remember the STOP is represented as the “T” block on the end. Once the EEPROM gets the stop condition it will begin writing to its memory. The write will not occur until it receives the stop condition.



Reading from an I²C EEPROM

● Read Example



Getting Started: I²C Master Mode

© 2001

Here is an example transfer of *reading* from a small EEPROM.

We need to transfer 4 bytes of information. The transfer will use the Control *IN* byte to load the address into the EEPROM. This sends data to the EEPROM which is why we use the control in byte. Once the address is loaded, we want to retrieve the data. So, we send a control *OUT* byte to indicate to the EEPROM that we want data *FROM* it. The EEPROM will acknowledge this and then send the data we requested. When we are done getting data, we send a “NACK” to tell the EEPROM that we don’t want more data. If we were to send an ACK at this point, we could get the next byte of data from the EEPROM. Since we only want to read one byte, we send a NACK. This is detailed in the specifications for the EEPROM.

As you can see, each byte is responded to with an ACK or NACK. If the PICmicro device sends a byte, the EEPROM responds with an ACK or NACK condition. If the EEPROM sends a byte, then the PICmicro microcontroller must reply with the required ACK or NACK condition. When the transfer is finished, a stop bit is sent by the PICmicro device.

You also will notice that a RESTART is used before the Control Out byte is sent. This is sent because the datasheet for the EEPROM states it is needed, but it is needed because the device must receive a start before it will understand the next byte is a control byte. This is part of the internal decoding hardware of the EEPROM. A start condition can not be used, since a stop condition has not yet occurred, and we are in the middle of a transfer. We will look at this in detail next.

- **When do I use a restart condition instead of a start condition?**
- A start condition can only be used when the bus is idle. A STOP will cause an idle bus, but if in the middle of a transmission a restart is needed.
- See the example of an EEPROM read for reference... *(on next slide)*

One frequently asked question about I²C transfers is:

When do I use a restart condition instead of a start condition?

The reasons can vary, but sometimes it becomes necessary to reset a device in the middle of a transfer. In the case of an EEPROM, it demands data to be in a particular order. It must have a Start, followed by a control byte, followed by an address if any, and then any data if applicable.

When reading from an EEPROM, you must WRITE the address IN to the device so that it understands what address you want to read. This is done by sending a Start, then a control IN, then the address desired.

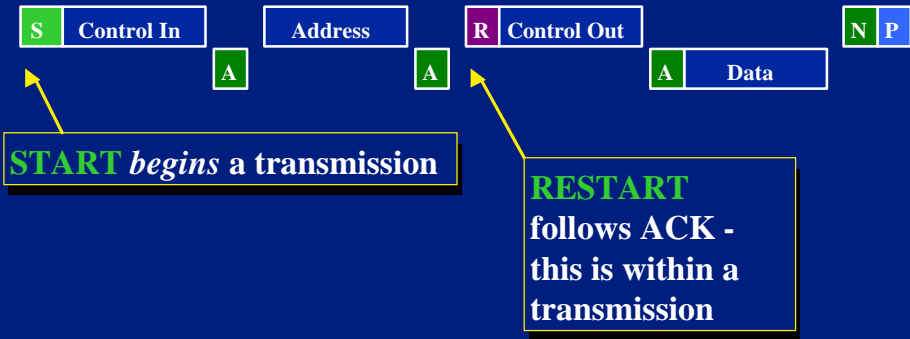
Once it has the address it is ready to be read, so a restart is used to stop the current transfer and immediately send a start condition. Once a start is sent, the control OUT byte is sent then the data can be obtained from the device.

Remember, start conditions can only be used on an idle bus, NOT in the middle of a transfer. An example of this can be seen on the next page.



Reading from an I²C EEPROM

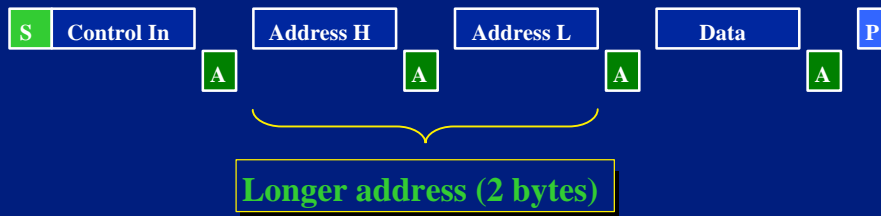
- Read Example - Using RESTART



Here is another look at our EEPROM read example. As you can see, a start condition is used to begin the transfer, a restart is used in the middle of the transfer to reset the EEPROM device, and a stop ends the transfer.

Writing to a Larger EEPROM

- Large EEPROM Write Example



Writing to a large EEPROM is not very different from a small one. The only difference is there are now *two* address bytes instead of one. As you would expect, each byte must be “ACK’ed” as well.

Above is a sample write to a large EEPROM. First the Control In byte is written, then the Address High, then Address Low. Finally, the Data byte is written followed by a stop condition. ACK’s follow each of the four data bytes.



I²C in the PICmicro MCU

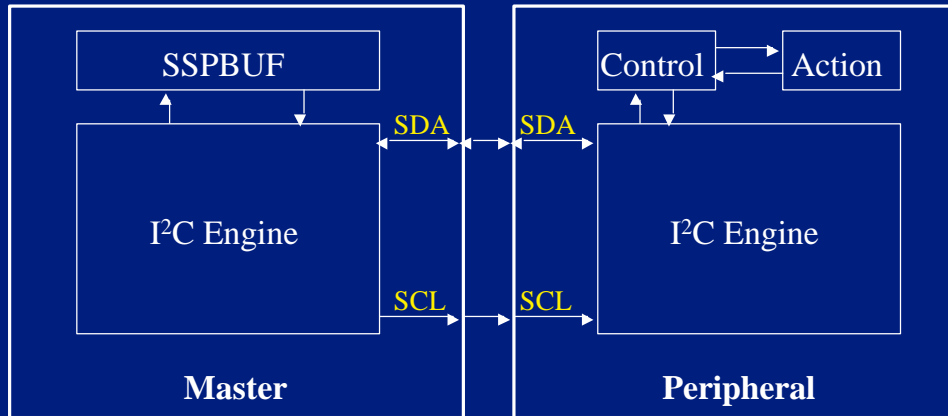
- The MSSP module in the PICmicro microcontroller (MCU) allows I²C and other synchronous serial protocols



In the PICmicro, a module is used for the I²C protocol. This module is named the MSSP module and allows SPI or I²C to be implemented.

I²C and SPI are both synchronous serial protocols, and hence the name of the MSSP module. MSSP stands for “Master Synchronous Serial Port”. If you want to use I²C ensure your PICmicro MCU has this port. Check the product line card or the device datasheet to ensure it has an MSSP module.

- I²C Data Transfer



I²C is implemented through the SDA and SCL lines.

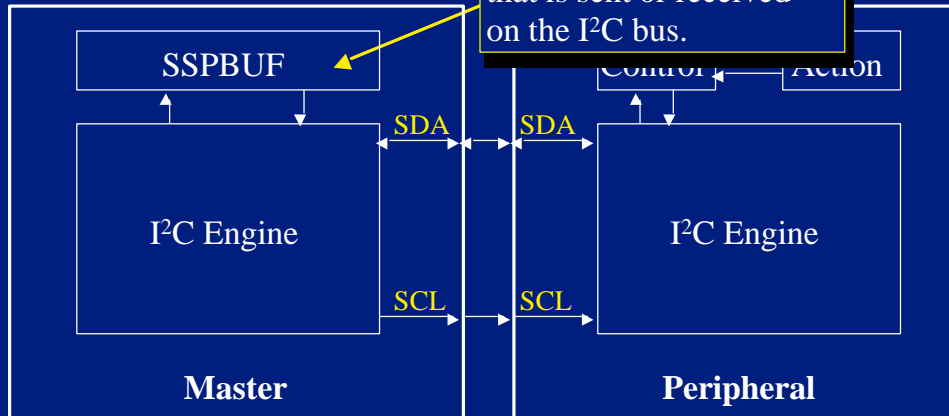
Data that is transmitted or received on the PICmicro I²C interface is sent to the SSPBUF register. The PICmicro microcontroller handles the details of clock generation and other features. If a start, restart, stop, ACK or other condition needs to be generated, one needs only to set the appropriate bits and wait for the condition to complete.

We will look at this diagram in some more detail now.

I²C in the PICmicro MCU

- I²C Data Transfer

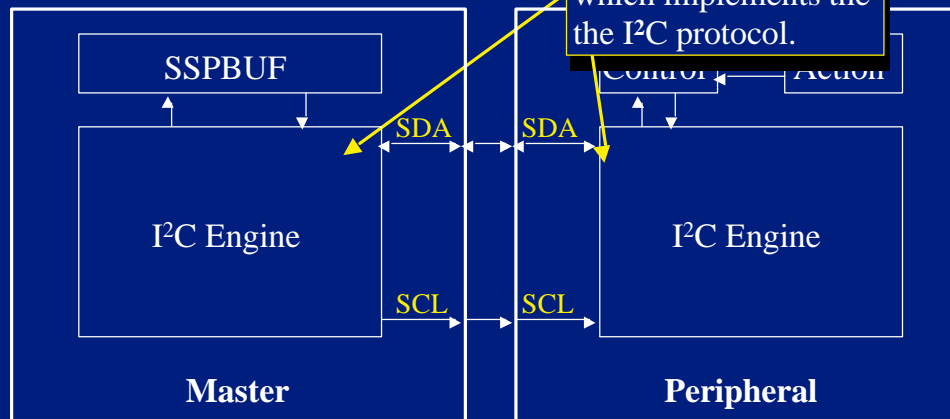
SSPBUF:
A register that stores data that is sent or received on the I²C bus.



Once a byte of data has been sent to the master via I²C, it is sent to the SSPBUF register. Data to be sent on the I²C bus is also sent to the SSPBUF register, which is then sent via I²C. SSPBUF holds data to transmit or received data, depending on the current mode of the MSSP module.

I²C in the PICmicro MCU

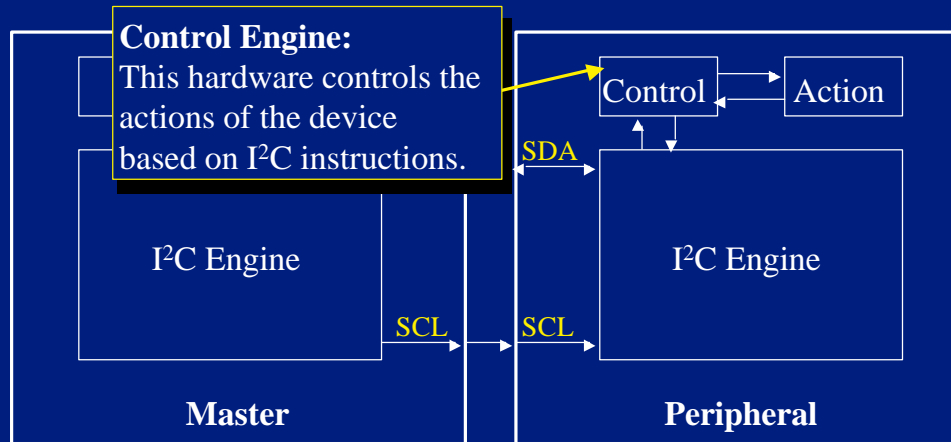
- I²C Data Transfer



The I²C engine sends data out on the I²C bus using the Clock (SCL) and Data (SDA) lines for communication. The I²C engine on the PICmicro device contains many registers which configure it as well as control its operation. The user has full access to these registers and we will look at them later in this presentation.

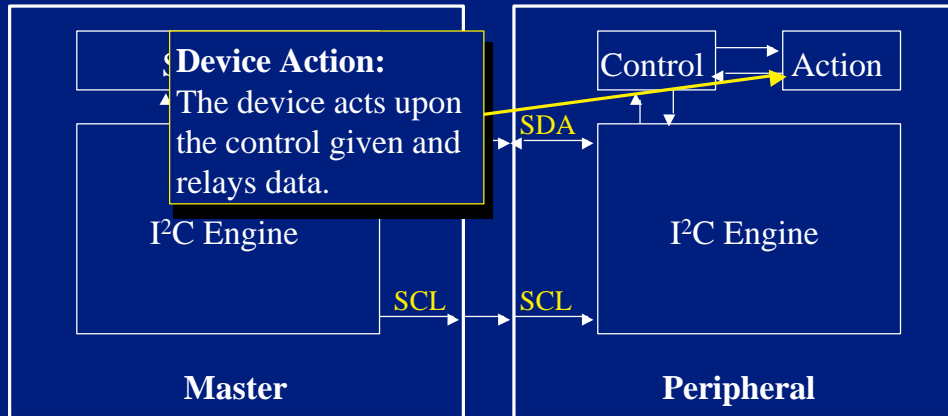
The I²C engine on a peripheral is usually fairly transparent to the user. The data sheet on the peripheral will tell you how to use the peripheral by telling you what commands must be sent and how it will respond.

- I²C Data Transfer



The peripheral will also contain a control engine of some kind. This engine will recognize valid I²C commands and direct that the peripheral perform the desired action. It could be thought of as a kind of instruction decoder. This same block will also take the data generated from the action of the device and communicate this to the I²C engine for transmission.

- I²C Data Transfer



The device action block represents that once the I²C request is decoded, it will either generate an action by the peripheral or it will generate data which is sent via I²C.

A common example of an action would be to have an EEPROM store some data. The Command would be received by the I²C engine, then the control block would decode what to do with the command, then the action block would perform the command, which is to store the data.

An example of generating data would be to instead request data from the EEPROM. The command would be decoded and generate an action. This action would be to have the EEPROM perform a read of the desired location, and then send the data to the control block. The control block would format the data and give it to the I²C engine. The I²C engine would then send the requested data.

This is a simplification of the process. However it gives you a general idea of how an I²C transfer works.



I²C in the PICmicro MCU

- 4 Registers control the function I²C in the PICmicro microcontroller
- The values to be placed in the registers will often depend on your application
- See device datasheet for details
- The next few slides will discuss the registers that control the MSSP module, and how it is used for I²C in Master Mode.

Getting Started: I²C Master Mode

© 2001

I²C on the PICmicro is controlled by 4 registers which we will look at in detail shortly. The values placed in these registers will control every aspect of the I²C communication. The device datasheet and reference manual will contain many details on using I²C and configuring these values, but we will give an introduction to them here. Note also that the MSSP module on a PICmicro device is capable of many different modes and configurations.

To simplify this presentation and to cover the most frequently asked topics, we will only be looking at how to talk to a standard Microchip Technology serial EEPROM. This is known as “Master mode” and contains only one master, the PICmicro microcontroller, and one slave, the EEPROM.



I²C – PICmicro (SSPCON)

- Here are the bits in the **SSPCON** Register:
 - **WCOL** *Write Collision*
 - **SSPOV** *Overflow*
 - **SSPEN** *Enable*
 - **SSPM3:SSPM0** (4 bits) *Function Control*
- *CKP is also in SSPCON but is not used for master mode I²C.*

Getting Started: I²C Master Mode

© 2001

The SSPCON register is one of the 4 registers that controls the I²C engine.

The bits on these registers can indicate errors and control the mode of the MSSP module.

WCOL - is an error flag and indicates that a “Write Collision” has occurred.

SSPOV - is also an error flag. It indicates an “Overflow” condition.

SSPEN - stands for “Synchronous Serial Port Enable”. This enables the MSSP module.

The last bits in this register are the SSPM bits 3 through 0. SSPM bits control if the SSP module is in an I²C mode and whether it is in master or slave mode. It also can control timing and other features. More details on all of these bits are in the device datasheet.



I²C - PICmicro (SSPCON)

- **WCOL** stands for Write COLLision and is set when the user tries to write to SSPBUF, but the I²C bus is not ready
- Generally used for debugging or for “Multi-Master Mode.”

WCOL - is an error flag and indicates that a “Write Collision” has occurred.

A Write collision occurs when the I²C module tries to output to the bus and it was found to be in use. This should never happen in our example or when commanding an EEPROM, but this bit is useful for error checking and handling. It is also used when more than one master is controlling the I²C bus. Multi-master communication is not discussed in this presentation.

When using many I²C devices, it is important that your code check this bit to handle the condition of devices writing to the bus when you do not expect it.



I²C - PICmicro (SSPCON)

- SSPOV stands for SSP OVerflow and is set when there is an overflow error
- Read data in SSPBUF before new data comes in to prevent this error

As mentioned previously, SSPOV means “Synchronous Serial Port OVerflow” and is set by the microcontroller whenever there is an overflow error.

An overflow error occurs whenever an I²C transfer finishes, but the previous data had not been read from the SSPBUF.

If SSPOV is set, it must be cleared by the user program. The user program should check to ensure SSPOV remains clear. This is part of good error checking in program design.

Note, data in the SSPBUF will not be updated until the overflow condition is cleared.



I²C - PICmicro (SSPCON)

- SSPEN stands for SSP Enable
- Set SSPEN to 1 to turn on the MSSP module
- Leave on for the entire time the MSSP module is in use
- SSPEN can be cleared to 0 to disable the MSSP module and to help conserve power

Getting Started: I²C Master Mode

© 2001

SSPEN is the “Synchronous Serial Port Enable” bit.

SSPEN is set to 1 to turn on the SSP module, such as when it is to be used for I²C communications.

The SSP module must be left on for the entire time the SSP module is in use.

SSPEN can be cleared to 0 to disable or reset the SSP module.



I²C - PICmicro (SSPCON)

- **SSPM3:SSPM0** control the MSSP mode.
- To enable Master Mode use the binary value of **1000**.

Getting Started: I²C Master Mode

© 2001

SSPM3:SSPM0 are 4 bits that yield 16 different Synchronous Serial Port modes.

These bits control whether the MSSP module is configured for SPI or I²C, whether it is in slave or master mode and other options. Full details can be found in the device data sheet.

The MSSP module has one important feature that no other SSP module has. It provides a “Hardware Master Mode”. It is this mode that is used in this presentation for I²C communications. This mode allows the MSSP module to handle all of the details of generating conditions, and sending and receiving data.

This mode is also known as “Master Mode” but it is *NOT firmware* master mode. Firmware mode means your code will handle the timing, but the MSSP or SSP module will detect the conditions, but not generate them for you.

To use “Hardware Master Mode” set these control bits to a binary value of **1000**.



I²C - PICmicro (SSPCON2)

- Here are the bits in the **SSPCON2** Register:
 - **GCEN** *General Call Enable*
 - **ACKSTAT** *ACK Bit Status*
 - **ACKDT** *ACK Transmit Data*
 - **ACKEN** *ACK Transmit Enable*
 - **RCEN** *Receive Enable*
 - **PEN** *Stop Condition Enable*
 - **RSEN** *Restart Condition Enable*
 - **SEN** *Start Condition Enable*

Getting Started: I²C Master Mode

© 2001

The SSPCON2 register is other register that controls the I²C engine on the PICmicro microcontroller.

All 8 bits in this register are used for this I²C mode and are the following:

GCEN - indicates “General Call ENable”

ACKSTAT - stands for “ACKnowledge bit STATus”

ACKDT - refers to the “ACKnowledge bit DaTa”

ACKEN - controls the “ACKnowledge ENable”

RCEN - is the “ReCeive ENable”

PEN - is the bit for “stoP condition ENable”

RESEN - is the control for a “ReStart condition ENable”

SEN - is the bit for “Start condition ENable”

We will look at these in more detail in a moment. Remember, more information on all of these bits is located in the device datasheet.



I²C - PICmicro (SSPCON2)

- GCEN allows an interrupt to be generated when an I²C “general Call Address” is generated.
- This feature is almost never used in I²C systems.
- This feature is not used in the example in this presentation

GCEN is a feature of I²C that allows the MSSP module to be backward compatible with older and/or slower systems. It is almost never used in an I²C system and is not used for our example. For more information on the “General Call” feature of I²C, consult the I²C specification.



I²C - PICmicro (SSPCON2)

- **ACKSTAT** indicates whether an ACK (acknowledge) or NACK (not acknowledge) was sent by the slave
- When talking to an I²C device (a slave), the device will return either an ACK or NACK for each byte transferred
- The user must read and interpret these conditions in their program

The ACKSTAT bit is set when an ACK or NACK has been received from the peripheral device. Remember, the peripheral must acknowledge all data bytes, and this is done by sending an ACK or NACK condition. This bit can be polled or tested to determine if an ACK or NACK condition has occurred. Its usage will be shown in our example program, which we will see later in this presentation.



I²C - PICmicro (SSPCON2)

- **ACKDT** is the data the master will transmit when “ACKing” an I²C device.
- When responding to an I²C device, there are only 2 possibilities: ACK and NACK, this bit controls which of the two is sent.

ACKDT indicates the data that will be transmitted if it is desired to send an acknowledge bit to the peripheral device. Remember, when the master reads data from a device, it must acknowledge the transfer by sending an ACK or NACK condition. This bit holds the value of the condition to be sent. If loaded with a 0, an ACK is sent, and if loaded with a 1, a NACK will be sent.



I²C - PICmicro (SSPCON2)

- **ACKEN** controls WHEN the PICmicro MCU will send the ACK or NACK signal.
- Once the user program has set or cleared ACKDT to set up an ACK or NACK condition, set this bit to start sending it.

ACKEN controls exactly *when* the acknowledge bit is sent. Regardless of the state of ACKDT, it is not sent *until* ACKEN is set. This allows one to set up the desired acknowledge bit to be sent, then send it when ready.



I²C - PICmicro (SSPCON2)

- RCEN enables I²C receive mode.
- When the PICmicro MCU must listen to data from another device, set RCEN
- RCEN automatically clears when one receive byte completes. This will cause the MSSP to revert back to transmit mode
- set each time more data is to be received

RCEN places the MSSP module into I²C receive mode.

In order to get a data byte from a peripheral, the PICmicro device must be put in receive mode. To activate this mode, the RCEN bit is set. Note that when one byte of data is received, this bit automatically clears and the PICmicro device returns to transmit mode. If you would like to receive another byte, set this bit again, but don't forget to ACK or NACK the data first!



I²C - PICmicro (SSPCON2)

- PEN sends a STOP condition
- Remember, “P” refers to a stoP condition
- Set this to start sending a stop condition on the I²C bus.
- PEN automatically clears when the STOP condition completes.

Setting the PEN bit will send a stop condition.

This stop condition will be sent automatically by the microcontroller. Once it completes you may send the next condition as the bit is automatically cleared at the end of the start condition.



I²C - PICmicro (SSPCON2)

- **RSEN** sends a RESTART condition
- Remember, “**R**” refers to a **R**estart condition
- Set this to start sending a restart condition on the I²C bus.
- RSEN automatically clears when the STOP condition completes.
- A restart condition is used when a start bit is needed, but there was no stop before it.

RSEN allows the user to send a restart condition on the I²C bus.

To send a restart condition, set this bit, then wait for the transfer to complete. This bit will also reset to 0 automatically, simply wait for the condition to complete before sending another condition or data.



I²C - PICmicro (SSPCON2)

- SEN sends a START condition
- Remember, “S” refers to a Start condition
- Set this to start sending a start condition on the I²C bus.
- SEN automatically clears when the START condition completes.

SEN is the “Start condition ENable” bit.

Just like sending a stop or restart condition, set the SEN bit to send a start condition. Wait for it to complete before sending another condition. Just like the other bits we just mentioned, the SEN bit will reset to 0 after the start condition completes.



I²C - PICmicro (SSPSTAT)

- Looking at the **SSPSTAT** Register, three bits help to control master mode I²C transfers:
 - **SMP** Slew Rate Control
 - **CKE** Signal Level Control
 - **BF** Buffer full
- Note: The other bits in this register are used for I²C, but not in the mode discussed for this presentation.

Getting Started: I²C Master Mode

© 2001

The next register that controls I²C is the SSPSTAT register. SSPSTAT stands for “Synchronous Serial Port STATUS” and provides a few bits for controlling the I²C communication.

Three bits in the SSPSTAT register control I²C. They are called “SMP”, “CKE” and “BF”. The bits are named after their functions in SPI, but they are used for I²C control.

SMP - enables the slew rate control of the I²C stream

CKE - controls the I²C levels.

and

BF - is the “Buffer Full” bit.

More details on all of these bits are in the device datasheet, and they will be discussed next.



I²C - PICmicro (SSPSTAT)

- **SMP** enables a Slew rate control to reduce EMI in 400 kps mode.
- The slew rate is enabled by the user when using 400 kbps on I²C.
 - If using 400 kbps on I²C, clear this bit to enable the slew rate control.
 - If other rates, set SMP to 1 to disable the slew rate control.

SMP enables the slew rate control of the MSSP module.

Depending on your desired I²C bus speed, you may want to enable the slew rate control. It is a filter that controls the slew rate on the I²C waveform to improve performance of 400 kbps I²C transmissions. If the I²C speed is too high, this filter will squelch the output, if the I²C speed is too slow, it will have little effect. Speeds around 400 kbps will have the sharp transitions replaced with a smooth waveform which generates less ElectroMagnetic Interference (EMI).



I²C - PICmicro (SSPSTAT)

- CKE controls the I²C input levels
- When using standard I²C, this bit is cleared to meet I²C levels.
- When using SMBus (similar to I²C), the voltage levels are different. This bit is set to 1 to conform to SMBus levels.

Getting Started: I²C Master Mode

© 2001

CKE controls the voltage range used by the PICmicro microcontroller when receiving I²C signals.

This bit allows the choice between standard I²C signal levels and SMBus signal levels. SMBus shares many of the features of I²C, but one major difference is the signal levels are a different range to be valid. This bit allows the MSSP module to handle SMBus peripherals.

This presentation will demonstrate communicating with a standard I²C device and so will not discuss SMBus further. If you need more information on SMBus, consult the specification.



I²C - PICmicro (SSPSTAT)

- BF stands for Buffer Full
- BF is set when the SSPBUF needs to be read
- BF is set and cleared by the PICmicro MCU

Getting Started: I²C Master Mode

© 2001

The “BF” bit stands for “Buffer Full”.

When this bit is set, it means that the SSPBUF contains data that has not yet been read. SSPBUF holds data that is received via I²C. The data should be read before any more data is written or received. This is true whether the device is a master or a slave.

The BF flag is set and cleared by the PICmicro. Note that if the SSPBUF is not read before another byte of data is exchanged, the SSPBUF will overflow and the SSPOV bit will get set.

As previously mentioned, when SSPOV is set, it indicates an overflow condition has occurred and the module must be reset to clear this condition. Toggling the SSPEN bit will reset the SSP module.



I²C - PICmicro (SSPADD)

- The **SSPADD** register controls the speed of the I²C bus transmissions. It controls the baud rate generator.
- Calculating I²C Baud Rate:

$$\frac{F_{osc}}{4 * (SSPADD + 1)}$$

The the last register to be discussed that controls I²C is the SSPADD register. SSPADD stands for “Synchronous Serial Port ADDRESS”, but in master mode this register has a different function. This register controls the speed of the I²C bus. The I²C bus speed is a calculation based on F_{osc}, which is the clock speed of the microcontroller, and the value loaded into SSPADD.

The formula is shown here. Lets try a sample calculation.



I²C - PICmicro (SSPADD)

- **Baud Rate Calculation:**

- Fosc = 4 MHz SSPADD = 9 (decimal)

$$\frac{F_{osc}}{4 * (SSPADD + 1)}$$

- = 4 MHz / (9 + 1) * 4
- = 4 MHz / 10 * 4
- = 4 MHz / 40
- = 100 kHz or **100 kbps**

If Fosc were 4 MHz, and SSPADD were 9 (decimal), then lets calculate the I²C speed:

Calculating the denominator first, we add the SSPADD value of 9 to 1 giving us 10. Then 10 is multiplied by 4 to yield 40.

4 MHz divided by 40 is 100 kHz or 100 kbps.

We use this value in our code example which we will look at shortly.



I²C - Code Example

- Code Example
- Send data from a PIC16F877 (or other PIC16F87X device) to a 24x01x I²C EEPROM.
- The data is written to the EEPROM, then read. The read data is then displayed on the LEDs on PORTB.
- Code works on PICDEMTM2, or build a circuit.

Getting Started: I²C Master Mode

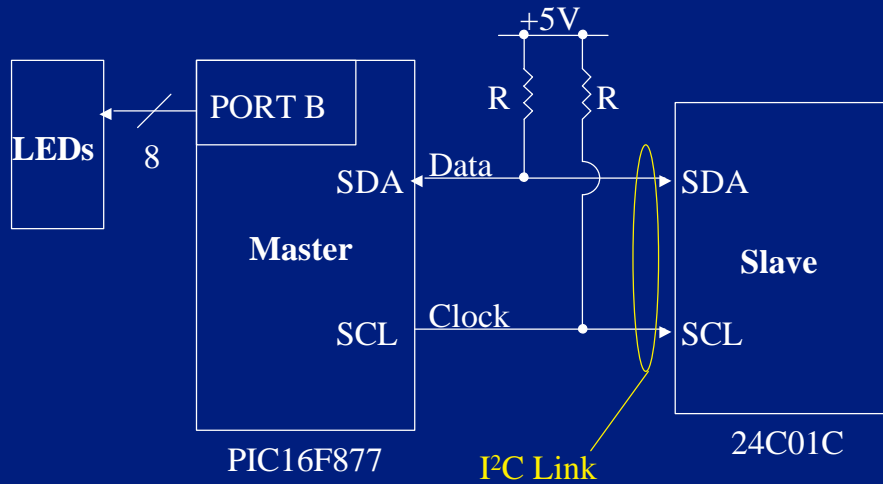
© 2001

Next we will show you a code example to demonstrate I²C on the PICmicro device. This example uses a Microchip PIC16F877 device connected to a 24x01x I²C EEPROM. We happened to use a 24C01C, but many different devices could be used.

These items were chosen since the PICDEM-2 will support the PIC16F877 and it comes with a 24C01C or newer serial EEPROM already soldered to it. Plus, this code outputs the value read on PORTB. On the PICDEM-2, there are LEDs already connected to this port. If you use a PICDEM-2 with a PIC16F877, you will not need to build any hardware at all to test this sample code.

I²C - Code Example

- Code Example - Schematic



If you prefer, you can also build the hardware yourself. Here is a simplified schematic of what is done in this example. An I²C link is set up between PICmicro MCU, in this case the PIC16F877 and the EEPROM. As mentioned earlier the EEPROM is a 24C01C or newer device, and many substitutions are viable. Pull-ups will be needed on the clock and data lines of the I²C, the values of which will depend on the desired speed. 2.2k is quite sufficient for the example but if you desire higher speed for later experimentation, consider using a 1k resistor for each pull-up.

As shown here, PORTB is used to drive 8 LEDs. These LEDs will display the value that is read from the EEPROM, after we have written and then read back from it with the sample code.

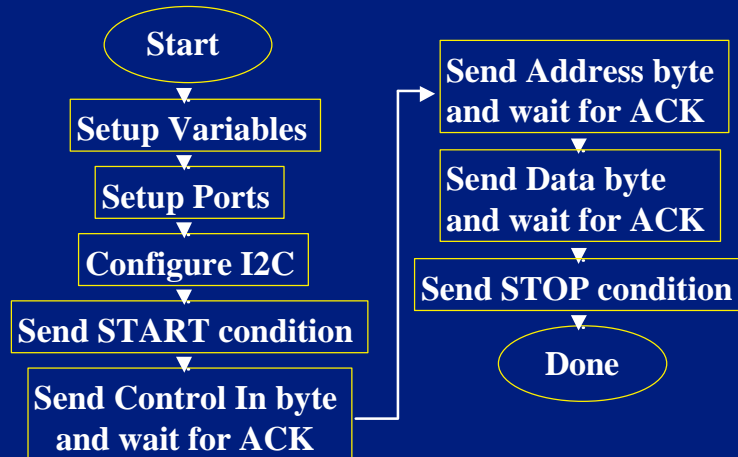
Data is sent from the master to the slave on the I²C link. The slave is our EEPROM and it will store some sample data that we send to it. Next, we will read the data back and then display it on the LEDs. Once this is done the code is finished.

We happen to write to the address “12” (hex), the value “34” (hex) and then read it back. If you see “34” (hex) on the LEDs, the system is working correctly. Any value could have been used, but these were arbitrarily chosen to prove you could write any data desired to any valid location of the EEPROM.

It is also recommended that you observe the I²C data on an oscilloscope. Doing so will show you the transfer in action. You can then compare the data that you found with the scope traces that will be shown at the end of this presentation.

I²C - Code Example - Write

- Flowchart for Writing (read on next slide)



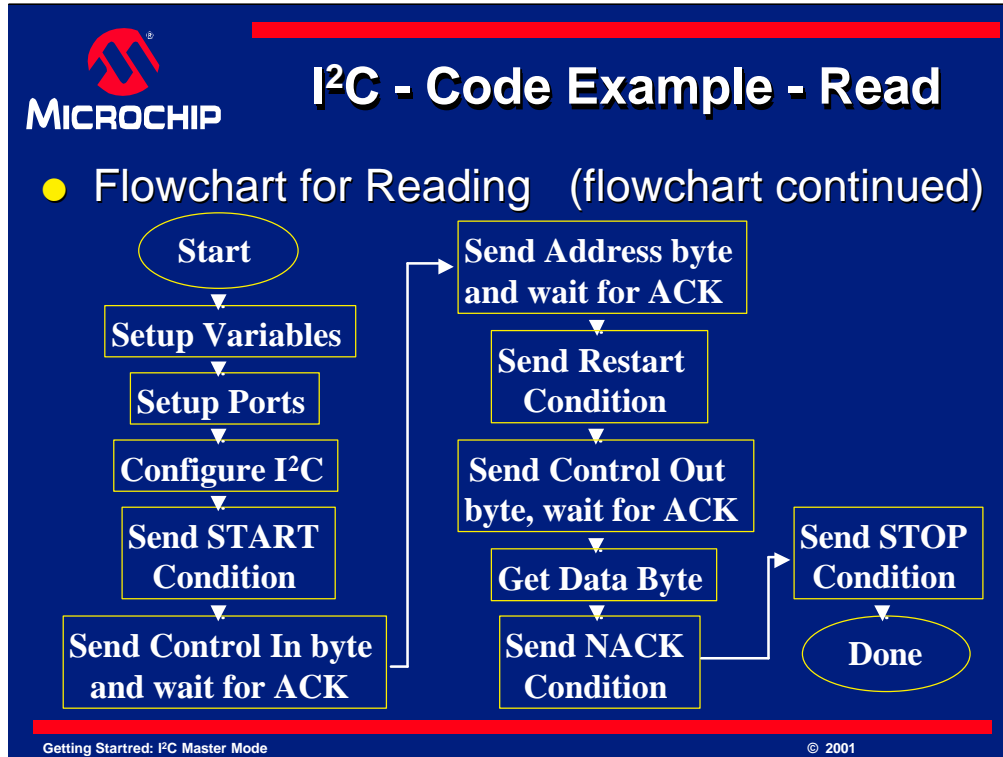
Next we will look at some example code.

Here is the flowchart for the EEPROM Write portion of our example code.

[pause]

Notice that the first few steps are used to configure the device, including the I²C port. After that is done, A start condition is sent to indicate we wish to begin sending data. Then, the control in byte is sent to indicate what we want to talk to, in this case an EEPROM, and that we want to write to its registers.

After that occurs, the address byte is sent. This tells the EEPROM which address we intend to write data too. Following the address is the data byte. The data byte contains the 8 bits of data that is to be written to the requested address. A Stop condition is then sent to close the transfer. Note that an ACK must occur after each byte. In this case, after the control in, address and data bytes, there is an ACK from the EEPROM. The sample code will look for the ACK and ensure that it is an ACK, not a NACK that is being sent. This helps to ensure the data is correct.



Here is the flowchart for the EEPROM Read portion of our example code.

[pause]

Once again, the first few steps are used to configure the device, including the I²C port. After that is done, A start condition is sent to indicate we wish to begin sending data. Then, the control in byte is sent to indicate we wish to communicate with the EEPROM again and that we want to write to its registers. We want to write because we need to tell it what address we want to read from.

As before, we will then send the address byte. Since we wish to read from the address we just wrote to, we will set this address to be the same value as before. Once this is done, we send a restart condition to indicate that we want to send new commands to the EEPROM.

When the restart completes, the Control OUT byte is sent, which will tell the EEPROM that we now want it to send data to the PICmicro microcontroller. The data byte is then clocked out of the EEPROM and once that finishes we reply with a NACK. The NACK tells the EEPROM in this case, that we do not need any more data. A stop condition is then sent to complete the transfer.

As before, after each transfer of a byte, an ACK or NACK is sent. ACK is sent from the EEPROM after the control in and Address byte. It also replies with an ACK after the control out byte, which follows the restart condition. When the data is clocked out of the EEPROM, the PICmicro device replies with a NACK to indicate it is finished. If it replied with an ACK, it would be telling the EEPROM that it wants the EEPROM to increment the address and send the next data byte. Since we only want to read one byte we send a NACK.



I²C - Code Example - Write

```

; I2C connected to 24C01C (or similar) EEPROM.
; Write to location 0x12, data 0x34 and read it back.
; The MSSP module is used in I2C MASTER mode.

#define LC0CTRLIN  H'AD'  ; I2C value for CONTROL BYTE when
                        ; Inputting data to the EEPROM
#define LC0CTRLOUT H'A1'  ; I2C value for CONTROL BYTE when
                        ; requesting OUTPUT from the EEPROM
#define LC0IADDR   H'12'  ; Sample value for ADDRESS BYTE
#define LC0IDATA   H'34'  ; Sample data to write to EEPROM
#define BAUD       D'100'  ; Desired Baud Rate in khps
#define PCLK       D'4000' ; Oscillator Clock in kHz

#include "p16f877.inc" ; Processor include file, for standard names
__CONFIG _CP_OFF & _DEBO_OFF & _WDT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BROWN_OFF & _PWRTE_ON & _MCLR_OFF & _XT_OSC

ORG 0
; Start of code (location 0)

; *** Setup I/O ***
clrf PORTB ; PORTB pins set to drive low when enabled
BANKSEL TRISC ; BANK 1
movlw B'00011000' ; RC2, RC4 are inputs for PORTC
movwf TRISC ; Remaining PORTC I/O lines are outputs
clrf TRISB ; all PORTB pins configured for output mode
; (enables all PORTB drivers for driving LEDs)

; *** Setup Registers for I2C ***
; Configure MSSP module for Master Mode
BANKSEL SSPCON
movlw B'00101000' ; Enables MSSP and uses appropriate
; PORTC pins for I2C mode (SSP7M set) AND
; Enables I2C Master Mode (SSPM6 bits)
movwf SSPCON ; This is loaded into SSPCON

; Configure Input Levels and slew rate as I2C Standard Levels
BANKSEL SSPSTAT
movlw B'10000000' ; Slew Rate control (SNP) set for 100kHz
movwf SSPSTAT ; mode and input levels are I2C spec.
; loaded in SSPSTAT

; Configure Baud Rate
BANKSEL SSPADD
movlw FREQ / (4 * BAUD) - 1 ; Calculates SSPADD Setting for
movwf SSPADD ; desired Baud rate and sets up SSPADD

; *** Begin I2C Data Transfer Sequences ***
I2CWRITE
; Send START condition and wait for it to complete
BANKSEL SSPCON2 ; BANK 1
btf SSPCON2,SEN ; Generate START Condition
call WaitMSSP ; Wait for I2C operation to complete

; Send and Check CONTROL BYTE, wait for it to complete
movlw LC0CTRLIN ; Load CONTROL BYTE (input mode)
call Send_I2C_Byte ; Send Byte
call WaitMSSP ; Wait for I2C operation to complete
BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send and Check ADDRESS BYTE, wait for it to complete
movlw LC0IADDR ; Load Address Byte
call Send_I2C_Byte ; Send Byte
call WaitMSSP ; Wait for I2C operation to complete
BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send and Check DATA BYTE, wait for it to complete
movlw LC0IDATA ; Load Data Byte
call Send_I2C_Byte ; Send Byte
call WaitMSSP ; wait for I2C operation to complete
BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send and Check the STOP condition, wait for it to complete
BANKSEL SSPCON2
btf SSPCON2,PEN ; Send STOP condition
call WaitMSSP ; Wait for I2C operation to complete
; The WRITE has now completed successfully. Begin the Read Sequence

```

This is an overview of the write portion of the sample code. It looks complex, but we will be breaking it up into portions shortly.



I²C - Code Example - Read

```
I2CRead
; Send RESTART condition and wait for it to complete
BANKSEL SSPCON2
bsf SSPCON2,RSEN ; Generate RESTART Condition
call WaitMSP ; Wait for I2C operation to complete

; Send and Check CONTROL BYTE, wait for it to complete
movlw LC0CTRLIN ; Load CONTROL BYTE (input)
call Send_I2C_Byte ; Send Byte
call WaitMSP ; Wait for I2C operation to complete

; Now check to see if I2C EEPROM is ready
BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CReady ; ACK Poll waiting for EEPROM write to complete

; Send and Check ADDRESS BYTE, wait for it to complete
movlw LC0IADDR ; Load ADDRESS BYTE
call Send_I2C_Byte ; Send Byte
call WaitMSP ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send REPEATED START condition and wait for it to complete
bsf SSPCON2,RSEN ; Generate REPEATED START Condition
call WaitMSP ; Wait for I2C operation to complete

; Send and Check CONTROL BYTE (out), wait for it to complete
movlw LC0CTRLOUT ; Load CONTROL BYTE (output)
call Send_I2C_Byte ; Send Byte
call WaitMSP ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Switch MSSP module to I2C Receive mode
bsf SSPCON2,RCEN ; Enable Receive Mode (I2C)

; Get the DATA BYTE and wait for it to complete. Data is in SSPBUF when done.
; The receive mode is disabled at end automatically by the MSSP module.
call WaitMSP ; Wait for I2C operation to complete
```

```
; Send NACK bit for Acknowledge Sequence
BANKSEL SSPCON2
bsf SSPCON2,ACKDT ; ACK DATA to send is 1, which is NACK.
bsf SSPCON2,ACKEN ; Send ACK DATA now.
; Once ACK or NACK is sent, the ACKEN is automatically cleared by the MSSP

; Send and Check the STOP condition and wait for it to complete.
bsf SSPCON2,PEN ; Send STOP condition
call WaitMSP ; Wait for I2C operation to complete

; I2C Write and Read have both finished, the value is output on LEDs.
BANKSEL SSPBUF ; BANK 0
movf SSPBUF,W ; Get data from SSPBUF into W register
movwf PORTB ; Output W register to LEDs on PORTB

; Program has finished and completed successfully.
goto $ ; Wait forever at this location

;*** SUBROUTINES & ERROR HANDLERS ***
; I2C Operation Failed code sequence - This will normally not happen.
; but if it does, a STOP is sent and the entire code is tried again.
I2CFail
BANKSEL SSPCON2
bsf SSPCON2,PEN ; Send STOP condition
call WaitMSP ; Wait for I2C operation to complete

BANKSEL PORTB ; BANK 0
movlw 0xFF ; Turn on all LEDs on PORTB
movwf PORTB ; to show error condition
goto $ ; Wait forever at this location

; This routine sends the W register to SSPBUF, thus transmitting a byte.
; When the SSPIF flag is checked to ensure the byte has been sent successfully.
; When that has completed, the routine exits, and executes normal code.
Send_I2C_Byte
BANKSEL SSPBUF ; BANK 0
movwf SSPBUF ; Get value to send from W, put in SSPBUF
retlw 0 ; Done, Return 0

; This routine waits for the last I2C operation to complete.
; It does this by polling the SSPIF flag in PIR1.
WaitMSP
BANKSEL PIR1 ; BANK 0
btfss PIR1,SSPIF ; Check if done with I2C operation
goto $-1 ; I2C module is not ready yet
bsf PIR1,SSPIF ; I2C module is ready, clear flag.
retlw 0 ; Done, Return 0

END
```

This is the other half of the sample code which controls the read of the EEPROM. Again, it looks complex, but this is the complete example. You will soon see the code is lengthy, but not complex.



I²C - Code Example - Write

```

; I2C connected to 24C01C (or similar) EEPROM.
; Write to location 0x12, data 0x34 and read it back.
; The MSSP module is used in I2C MASTER mode.

#define LC01CTRLIN  H'A0'    ; I2C value for CONTROL BYTE when
                           ; Inputting data to the EEPROM

#define LC01CTRLOUT H'A1'    ; I2C value for CONTROL BYTE when
                           ; requesting OUTPUT from the EEPROM

#define LC01ADDR    H'12'    ; Sample value for ADDRESS BYTE

#define LC01DATA    H'34'    ; Sample data to write to EEPROM

#define BAUD        D'100'    ; Desired Baud Rate in khps
#define PRC        D'4000'    ; Oscillator Clock in kHz

#include <pic16f877.inc> ; Processor Include file, for standard names

; _CONFIG _CP_OFF & _DESRG_OFF & _WDT_ENABLE_OFF & _CPD_OFF & _LVP_OFF
& _BODEN_OFF & _PWRTE_ON & _WDT_OFF & _XT_OSC

ORG 0 ; Start of code (location 0)

; *** Setup I/O ***
clrf PORTB ; PORTB pins set to drive low when enabled
BANKSEL TRIS ; BANK 1
movlw B'00011000' ; RC3, RC4 are inputs for PORTC
movwf TRISC ; Remaining PORTC I/O lines are outputs
clrf TRISB ; all PORTB pins configured for output mode
; enables all PORTB drivers for driving LEDs

; *** Setup Registers for I2C ***
; Configure MSSP module for Master Mode
BANKSEL SSPCONW
movlw B'00101000' ; Enables MSSP and uses appropriate
                 ; PORTC pins for I2C mode (SSPEN set) AND
                 ; Enables I2C Master Mode (SSPM0 bits)

movwf SSPCONW ; This is loaded into SSPCONW

; Configure Input Levels and Slew rate as I2C Standard Levels
BANKSEL SSPSTAT
movlw B'10000000' ; Slew Rate control (SMP) set for 100kHz
movwf SSPSTAT ; Mode and Input levels are I2C spec,
              ; loaded in SSPSTAT

; Configure Baud Rate
BANKSEL SSPADD
movlw SSPCON / (4 * BAUD) - 1 ; Calculates SSPADD Setting for
movwf SSPADD ; desired Baud rate and sets up SSPADD

; *** Begin I2C Data Transfer Sequences ***
I2CWRITE
; Send START condition and wait for it to complete
BANKSEL SSPCON2 ; BANK 1
btf SSPCON2,SEN ; Generate START Condition
call WaitMSSP ; Wait for I2C operation to complete

; Send and Check CONTROL BYTE, wait for it to complete
movlw LC01CTRLIN ; Load CONTROL BYTE (input mode)
call Send_I2C_Byte ; Send Byte
call WaitMSSP ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send and Check ADDRESS BYTE, wait for it to complete
movlw LC01ADDR ; Load Address Byte
call Send_I2C_Byte ; Send Byte
call WaitMSSP ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send and Check DATA BYTE, wait for it to complete
movlw LC01DATA ; Load Data Byte
call Send_I2C_Byte ; Send Byte
call WaitMSSP ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto I2CFail ; failed, skipped if successful

; Send and Check the STOP condition, wait for it to complete
BANKSEL SSPCON2
btf SSPCON2,REN ; Send STOP condition
call WaitMSSP ; Wait for I2C operation to complete
; The WRITE has now completed successfully. Begin the Read Sequence

```

We will break the write code example into 6 parts which we will look at shortly.



I²C - Code Example - Read

```
I2CRead
; Send RESTART condition and wait for it to complete
BANKSEL SSPCON2
bsf  SSPCON2,RSEN  ; Generate RESTART Condition
call WaitMSP      ; Wait for I2C operation to complete

; Send and Check CONTROL BYTE, wait for it to complete
movlw LC0CTRLIN   ; Load CONTROL BYTE (input)
call  Send_I2C_Byte ; Send Byte
call  WaitMSP     ; Wait for I2C operation to complete

; Now check to see if I2C EEPROM is ready
BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto  I2CReady       ; ACK Poll waiting for EEPROM write to complete
```

7

```
; Send and Check ADDRESS BYTE, wait for it to complete
movlw LC0ADDR     ; Load ADDRESS BYTE
call  Send_I2C_Byte ; Send Byte
call  WaitMSP     ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto  I2CFail      ; failed, skipped if successful

; Send REPEATED START condition and wait for it to complete
bsf  SSPCON2,RSEN  ; Generate REPEATED START Condition
call WaitMSP      ; Wait for I2C operation to complete
```

8

```
; Send and Check CONTROL BYTE (out), wait for it to complete
movlw LC0CTRLOUT  ; Load CONTROL BYTE (output)
call  Send_I2C_Byte ; Send Byte
call  WaitMSP     ; Wait for I2C operation to complete

BANKSEL SSPCON2
btfsc SSPCON2,ACKSTAT ; Check ACK Status bit to see if I2C
goto  I2CFail      ; failed, skipped if successful

; Switch MSSP module to I2C Receive mode
bsf  SSPCON2,RCEN  ; Enable Receive Mode (I2C)

; Get the DATA BYTE and wait for it to complete. Data is in SSPBUF when done.
; The receive mode is disabled at end automatically by the MSSP module.
call  WaitMSP     ; Wait for I2C operation to complete
```

9

```
; Send NACK bit for Acknowledge Sequence
BANKSEL SSPCON2
bsf  SSPCON2,ACKDT ; ACK DATA to send is 1, which is NACK.
call WaitMSP      ; Send ACK DATA now.
; Once ACK or NACK is sent, the ACKEN is automatically cleared by the MSSP

; Send and Check the STOP condition and wait for it to complete.
bsf  SSPCON2,SEN   ; Send STOP condition
call WaitMSP     ; Wait for I2C operation to complete

; I2C Write and Read have both finished, the value is output on LEDs.
BANKSEL SSPBUF
movf  SSPBUF,W    ; Get data from SSPBUF into W register
movwf PORTB      ; Output W register to LEDs on PORTB
```

10

```
; Program has finished and completed successfully.
goto $           ; Wait forever at this location

*** SUBROUTINES & ERROR HANDLERS ***
; I2C Operation Failed code sequence - This will normally not happen.
; but if it does, a STOP is sent and the entire code is tried again.
I2CFail
BANKSEL SSPCON2
bsf  SSPCON2,SEN   ; Send STOP condition
call WaitMSP     ; Wait for I2C operation to complete

BANKSEL PORTB
movf  W,PORTB     ; Turn on all LEDs on PORTB
movwf PORTB      ; to show error condition
goto $           ; Wait forever at this location
```

11

```
; This routine sends the W register to SSPBUF, thus transmitting a byte.
; When, the SSPIF flag is checked to ensure the byte has been sent,
; when that has completed, the routine exits, and executes normal code.
Send_I2C_Byte
BANKSEL SSPBUF
movf  W,SSPBUF   ; BANK 0
movwf SSPBUF    ; Get value to send from W, put in SSPBUF
retlw 0         ; Done, Return 0

; This routine waits for the last I2C operation to complete.
; It does this by polling the SSPIF flag in PIR1.
WaitMSP
BANKSEL PIR1
btfss PIR1,SSPIF ; BANK 0
goto  $          ; I2C module is not ready yet
bsf  PIR1,SSPIF  ; I2C module is ready, clear flag.
retlw 0         ; Done, Return 0

END
```

12

As you can see, we have also done the same to the read example code. It has been broken down into sections numbered 7 through 12, while the earlier code was broken into sections 1 through 6. There are only 12 sections to look at to study the entire code example.

- Plan for EEPROM Write



- Send Start.
- Send Control (input mode). Get Ack.
- Send Address. Get Ack.
- Send Data. Get Ack.
- Send Stop.

Lets begin looking at the example code now. Remember that to write to the EEPROM, the sequence goes like this:

The PICmicro microcontroller sends a Start bit, followed by the Control In byte. This is then ACKed by the EEPROM. The PICmicro device waits for the ACK and ensures it received an ACK. Then the Address byte is sent. The ACK is again waited for and tested. Finally, the data to write to the EEPROM is sent, which is also ACK'ed by the EEPROM. Again the PICmicro microcontroller waits for the ACK and tests it. Once all of this has completed it sends a STOP condition to end the transfer.



I²C Code - Section 1 of 12

```
; I2C connected to 24C01C (or similar) EEPROM. ← Comments
; Write to location 0x12, data 0x34 and read it back.
; The MSSP module is used in I2C MASTER mode.

#define LC01CTRLIN    H'A0'    ; I2C value for CONTROL BYTE when
                                ; INputing data to the EEPROM

#define LC01CTRLOUT   H'A1'    ; I2C value for CONTROL BYTE when
                                ; requesting OUTput from EEPROM

#define LC01ADDR      H'12'    ; Sample value for ADDRESS BYTE

#define LC01DATA      H'34'    ; Sample data to write to EEPROM

#define BAUD          D'100'   ; Desired Baud Rate in kbps
#define FOSC          D'4000'  ; Oscillator Clock in kHz
```

Getting Started: I²C Master Mode

© 2001

In section 1 of our code, we set up the basics similar to other programs.

The first part of the program is like any other. Some comments at the top state what the program is and what it does. Having well documented and commented code is good general coding practice.

Several `#define` statements follow which define items like the value of a CONTROL IN byte, and the value of a CONTROL OUT byte. We have also defined the address that we will write to, in this case the address of 0x12 (hex). The sample data value is also defined here, 0x34 (hex). Finally, we have also defined the intended I2C baud rate (100 kbps) and the oscillator frequency, Fosc. As you can see, we are assuming a 4 MHz oscillator. If you wish to use other values, they can be changed in this section.

Using `#define` statements prevent the need to change large numbers of literal values thought the program. Instead of changing countless values, you need only change the defined values. This is again good coding practice for any programming.

```

#include <p16F877.inc>           ; Processor Include file

__CONFIG _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF &
_CPD_OFF & _LVP_OFF & _BODEN_OFF & _PWRTE_ON & _WDT_OFF &
_XT_OSC

ORG 0

; *** Setup I/O ***
clrf   PORTB           ; PORTB pins set to drive low
BANKSEL TRISC          ; BANK 1
movlw  B'00011000'    ; RC3, RC4 are inputs for PORTC
movwf  TRISC           ; Remaining PORTC I/O is output

clrf   TRISB           ; all PORTB pins in output mode
                        ; (enables all PORTB drivers)

```

Processor Setup (points to `<p16F877.inc>`)

Configuration Bits (points to `__CONFIG` lines)

Code Start (points to `ORG 0`)

Port Setup (points to `clrf PORTB`)

The next section is the processor setup, and indicates the processor used for the code. This example uses a PIC16F877, but many other devices could be used.

Next, the `__CONFIG` directive is used to set the configuration bits. Doing this prevents mistakes during programming. The configuration bits can be changed at program time, but this directive changes the default state. As a general rule this should be used in your program. The values for configuration bits are found in the include file for your processor. For this example, see the bottom of file “P16F877.inc” in your MPLAB directory.

The code starts at “ORG 0”. The ORG is a directive to MPLAB and stands for “ORiGin” and tells the assembler where in program memory to locate the next instruction. Any time an “ORG” is encountered, the next program memory instruction will begin at the new location. In this example, the program will begin at 0, while the next instruction will be at program memory location 1, then 2 and so on.

The directive `BANKSEL` is used so that the registers `TRISC` and `TRISB` can be accessed. Remember that the PICmicro microcontroller uses banked registers, and so care must be taken to be in the correct bank at all times. If you build this program and others like it in MPLAB, it will warn you whenever a register is not in bank 0. It does this to provide a helpful reminder of this issue.

```

; *** Setup Registers for I2C ***
; Configure MSSP module for Master Mode
BANKSEL SSPCON
movlw B'00101000' ; Enables MSSP and uses
                  ; PORTC pins for I2C mode
                  ; (SSPEN set) AND
                  ; Enables I2C Master Mode
                  ; (SSPMx bits)

movwf SSPCON ; This is loaded into SSPCON

; Input Levels and slew rate as I2C Standard Levels
BANKSEL SSPSTAT
movlw B'10000000' ; Slew Rate control (SMP) 100kHz
movwf SSPSTAT ; mode and input levels are I2C
               ; loaded in SSPSTAT

```

↖ I2C Configuration

Now we begin the code to set up the I²C control registers.

SSPCON is loaded with the value to set I²C Hardware Master Mode, which we use to make communication with the EEPROM easy. We simply request an action and wait for it to complete. The MSSP module is also enabled here as we set the SSPEN bit to turn it on.

Next SSPSTAT is configured. The slew rate control is set for 100 kbps use and input levels are set to standard I²C levels.

```

; Configure Baud Rate
BANKSEL SSPADD
movlw (FOSC / (4 * BAUD)) - 1 ; Calculates SSPADD
movwf SSPADD ; for desired Baud rate
; and sets up SSPADD

; *** Begin I2C Data Transfer Sequences ***
I2CWrite
; Send START condition and wait for it to complete
BANKSEL SSPCON2 ; BANK 1
bsf SSPCON2,SEN ; Generate START Condition

call WaitMSSP ; Wait for I2C operation

```

I²C Configuration (continued)

Start of I²C Communications

S

In this section, the baud rate is set up. Using features of MPLAB allows the baud rate configuration value to be calculated automatically. The values that we defined earlier (see section 1), are used in the formula to calculate a value to load into SSPADD. The value in SSPADD controls the baud rate. Details of this were discussed in this presentation and are found in the device data sheet.

At this point, the I²C configuration registers are set up. It is now time to begin our write to the EEPROM.

Setting bit SEN will begin our start condition, we want to wait for it to complete, so a subroutine, named “WaitMSSP”, is called. This subroutine will test for when the start condition or other action is finished.

The “WaitMSSP” call will run a subroutine that we will look at in detail later. This subroutine polls a flag repeatedly until the flag indicates that the MSSP action has finished. Once the flag indicates the MSSP action is done, the subroutine ends and the next instruction is executed.

```

; Send and Check CONTROL BYTE, wait for it to complete
movlw  LC01CTRLIN      ; Load CONTROL BYTE (input mode)
call   Send_I2C_Byte   ; Send Byte
call   WaitMSSP        ; Wait for I2C operation

BANKSEL SSPCON2
btfsc  SSPCON2,ACKSTAT ; Check ACK Status bit
goto   I2CFail         ; failed, skipped if successful

; Send and Check ADDRESS BYTE, wait for it to complete
movlw  LC01ADDR        ; Load Address Byte
call   Send_I2C_Byte   ; Send Byte
call   WaitMSSP        ; Wait for I2C operation

BANKSEL SSPCON2
btfsc  SSPCON2,ACKSTAT ; Check ACK Status bit
goto   I2CFail         ; failed, skipped if successful

```

Control In

A

Address

A

The next few lines of code send a control in byte. This is needed to tell the EEPROM we want to send data to it. Call WaitMSSP is used again to wait for this to complete.

The EEPROM will send an ACK or a NACK to respond to this. To read the condition, we read the ACKSTAT bit. In this code, if the EEPROM responds with a NACK, the code will go to the “I2CFail” location of the code to handle the error. If it responds with an “ACK”, the next line executes.

The next few lines send the address byte to the EEPROM and wait for an ACK. Notice how these lines look very similar to the sending of the control in byte.

[pause]

You should start to see a pattern by now. When sending a byte in I²C, W is loaded with the value to send and a subroutine “Send_I2C_Byte”, is called to send it. Then the “WaitMSSP” subroutine waits for the byte or other condition to finish. These subroutines will be looked at in detail at the end of the program.

To check for ACK, the ACKSTAT bit is checked in SSPCON2 and a decision is made from there as to what the program does next.


```

; Send and Check DATA BYTE, wait for it to complete
movlw  LC01DATA      ; Load Data Byte
call   Send_I2C_Byte ; Send Byte
call   WaitMSSP      ; Wait for I2C operation

BANKSEL SSPCON2
btfsc  SSPCON2,ACKSTAT ; Check ACK Status bit
goto   I2CFail       ; failed, skipped if successful

; Send and Check the STOP condition, wait for it to complete
BANKSEL SSPCON2
bsf    SSPCON2,PEN   ; Send STOP condition
call   WaitMSSP      ; Wait for I2C operation
; The WRITE has now completed successfully.
; Begin the Read Sequence

```

Data

A

P

Next, the data byte is sent. Notice that since we are sending a byte, W is loaded with the value, the `Send_I2C_Byte` code is run, and we wait for it to complete. Then the ACK bit is checked.

The EEPROM write is now almost finished. All that remains is to send a stop condition and wait for it to complete. This is what the next few lines do.

Much like sending a start condition, instead of setting SEN, the PEN bit is set. This tells the MSSP module to send a stop condition. After waiting for that to complete, the EEPROM Write is finished.

I²C - Code Example - Read

- Plan for EEPROM Read



- Send Start.
- Send Control (input mode). Get Ack.
- Send Address. Get Ack.
- Send Restart and Control (output mode).
- Get Ack. Get Data.
- Send Nack.
- Send stoP.

Here is what is needed to read the data from the EEPROM. Remember, a start condition is sent, followed by the control in byte. The EEPROM responds with an ACK and the PICmicro then sends the address information. Again the EEPROM responds with an ACK. The address is now loaded, so a restart condition is initiated followed by a control out byte to indicate to the EEPROM that a read is required next. The EEPROM ACK's this and sends the data. NACKing this data indicates no more data is needed and a stop condition follows. Lets quickly review the code step by step. We will also review the subroutines "Send_I2C_Byte" and "WaitMSSP", at the end of this code.

[pause]



I²C - Code Example - Read

- ACK Polling



- When the EEPROM ACK's the control byte, it is ready for new commands
- EEPROM will NACK data if busy, so test again

Getting Started: I²C Master Mode

© 2001

There is one problem with the approach we just discussed. If the EEPROM is busy, it will NACK the next command. This NACK will occur after we send the control byte. See the diagram here. If it is still busy, it will NACK again. If one were to keep sending a control byte, one could discover exactly when the EEPROM was ready. Doing this allows the fastest access time and is called “ACK Polling”.

In order to perform ACK polling, one sends the control byte, in this case control in, and checks the ACK or NACK response from the EEPROM. If it is a NACK, the EEPROM is busy, so we send a restart to reset the EEPROM and then try again by sending another control byte. This is done over and over until an ACK is returned.

When an ACK is returned, then we can continue the transfer.



I²C - Code Example - Read

- Plan for EEPROM Read - with ACK Polling



- The EEPROM may still be busy writing
- When EEPROM is busy it will NACK commands
- ACK Polling keeps asking “Are you ready?”
- RESTART is substituted for START

Notice in this diagram the start condition has been changed to a restart condition. This is done because a restart is simply a stop followed by a start condition. So, even if we just had a stop before the restart, we get two stops, followed by a start. Two stop conditions in a row is not a problem and perfectly legal in I²C. So, the start has been replaced with a restart.

Now the EEPROM can be tested if it is ready for commands by looping. First a restart is sent, then the control in byte, then the ACK or NACK is checked. If it is an ACK, the program loops back to try again. If it is an NACK, it can continue. This handles the possibility of the EEPROM being busy after the next command.

```

I2CRead
; Send RESTART condition and wait for it to complete
  BANKSEL SSPCON2
  bsf      SSPCON2,RSEN      ; Generate RESTART Condition
  call    WaitMSSP          ; Wait for I2C operation

; Send and Check CONTROL BYTE, wait for it to complete
  movlw   LC01CTRLIN        ; Load CONTROL BYTE (input)
  call    Send_I2C_Byte     ; Send Byte
  call    WaitMSSP          ; Wait for I2C operation

; Now check to see if I2C EEPROM is ready
  BANKSEL SSPCON2
  btfsc   SSPCON2,ACKSTAT   ; Check ACK Status bit
  goto    I2CRead           ; ACK Poll waiting for EEPROM
                                ; write to complete
  
```

R

Control In

A

Here is the continuation of the I²C sample code. This code performs a read of the EEPROM.

To start the read sequence, a restart condition is used. A restart is sent by setting bit “RSEN”. After the restart sequence completes, the control in byte is sent to tell the EEPROM we want to send data to it. The EEPROM will reply with either a NACK if busy, or an ACK if ready for more data.

The code handles both conditions. The ACK bit is tested, and if it is a NACK, the instruction at the bottom “goto I2C read”, is executed. This is the heart of the ACK polling. When the code is sent to “I2C read”, it returns to the top and runs the sequence again. This will happen repeatedly until the EEPROM answers with an ACK.

When the EEPROM answers with ACK, the data transfer continues.

I²C Code - Section 8 of 12

```
; Send and Check ADDRESS BYTE, wait for it to complete
movlw  LC01ADDR      ; Load ADDRESS BYTE
call   Send_I2C_Byte ; Send Byte
call   WaitMSSP      ; Wait for I2C operation

BANKSEL SSPCON2
btfsc  SSPCON2,ACKSTAT ; Check ACK Status bit
goto   I2CFail        ; failed, skipped if successful

; Send REPEATED START condition and wait for it to complete
bsf    SSPCON2,RSEN   ; Generate RESTART Condition
call   WaitMSSP      ; Wait for I2C operation
```

Now it is time to send the address to the EEPROM. The address is sent by the next few lines and again we wait for this action to finish. When it has finished, it is again ACK'd by the EEPROM.

Next the restart condition is sent. To send a restart, RSEN is set. When that completes, the program continues.

I²C Code - Section 9 of 12

```

; Send and Check CONTROL BYTE (out), wait for it to complete
movlw  LC01CTRLOUT      ; Load CONTROL BYTE (output)
call   Send_I2C_Byte    ; Send Byte
call   WaitMSSP         ; Wait for I2C operation

                                ← Control Out
BANKSEL SSPCON2
btfsc  SSPCON2,ACKSTAT  ; Check ACK Status bit
goto   I2CFail         ; failed, skipped if successful

                                ← A
; Switch MSSP module to I2C Receive mode
bsf    SSPCON2,RCEN    ; Enable Receive Mode (I2C)

                                ← Data
; Get the DATA BYTE and wait for it to complete.
; Data is in SSPBUF when done.
; The receive mode is disabled at end automatically by the
; MSSP module.

call   WaitMSSP         ; Wait for I2C operation

```

Its time now to send the control out byte. This tells the EEPROM to send data out to the PICmicro MCU. After we check for the ACK condition, it is time to receive the data.

To receive data, the MSSP module must be put in receive mode. This is done by setting the receive mode bit “RCEN”. The line “bsf SSPCON2,RCEN” sets this bit to enable the receive mode. The data byte is automatically clocked out of the EEPROM by the PICmicro and is in SSPBUF register when finished.

Note that once that the *one* byte of data has be transferred, receive mode automatically ends. It can take some time to receive a byte, so the “WaitMSSP” code is called again to allow the byte to fully transfer before continuing.

I²C Code - Section 10 of 12

```

; Send NACK bit for Acknowledge Sequence
BANKSEL SSPCON2
bsf    SSPCON2,ACKDT    ; ACK DATA to send is 1 (NACK)
bsf    SSPCON2,ACKEN    ; Send ACK DATA now.
; Once ACK or NACK is sent, ACKEN is automatically cleared

; Send and Check the STOP condition and wait for it
bsf    SSPCON2,PEN      ; Send STOP condition
call   WaitMSSP         ; Wait for I2C operation

; I2C Write and Read have both finished, value is on LEDs
BANKSEL SSPBUF          ; BANK 0
movf   SSPBUF,W         ; Get data from SSPBUF into W
movwf  PORTB           ; Output W register to LEDs

; Program has finished and completed successfully.
goto   $                ; Wait forever at this location

```

Output on LEDs ↑

End of I2C Communications

After the data byte is received, we must reply to the EEPROM with an ACK or NACK condition. The EEPROM expects an ACK, when more data is desired and a NACK to say that no more data is required. Since this example only reads one byte of data, a NACK is sent.

To send an ACK or NACK condition, see the code at the top of this slide.

[pause]

The desired data, a 0 for ACK, a 1 for a NACK reply, is loaded into the ACKDT bit. Recall this is the *ACK data* bit. Once this is done, the ACK or NACK condition is sent by setting ACKEN, which is the *ACK enable* bit.

Just like when sending a start, stop or restart condition, when the ACK or NACK is finished, the ACKEN bit is automatically cleared. Once the code waits for it to complete (using the WaitMSSP routine), it is time to send the stop condition to indicate the end of the transfer.

The stop condition code should look very familiar. The PEN bit is set, and we wait for the condition to complete with WaitMSSP.

Finally, this code sends the result of the read to PORTB for display on LEDs. Recall the data that was received was loaded into SSPBUF from the read earlier. So, this code copies SSPBUF data into PORTB for display.

At this point, the main code is finished, so to prevent the program counter from advancing further, the “goto \$” code forces the the PICmicro to freeze at that line. “\$” means “this line” so “goto \$” means “go to this line”, where this, is the current line.


```
; *** SUBROUTINES & ERROR HANDLERS ***  
; I2C Operation Failed code sequence - This will normally not  
; happen, but if it does, a STOP is sent and the entire code  
; is tried again.  
I2CFail  
    BANKSEL SSPCON2  
    bsf     SSPCON2,PEN      ; Send STOP condition  
    call   WaitMSSP         ; Wait for I2C operation  
  
    BANKSEL PORTB          ; BANK 0  
    movlw  0xFF             ; Turn on all LEDs on PORTB  
    movwf  PORTB           ; to show error condition  
    goto   $               ; Wait forever at this location
```

Error Handler ←

Its now time to look at the error handling code and subroutines.

This code is used to handle any error that results from the I²C main code.

You may recall seeing “goto I2CFail” lines in the earlier code. They were placed after we tested the ACK or NACK status. If the ACK or NACK status was not what was expected, the “goto I2CFail” line would execute and this code would run. Lets see what it does.

This code is known as an error handler. As you can see, this error handler sends a stop condition and then waits for it to complete. This is important to release the I²C bus. Then it goes to PORTB and puts 0xFF (hex) on it. This will light all of the LEDs on PORTB. This is done to provide a very easy indication to you that something is wrong.

Since the code has been proven, it could be hardware related, or perhaps an error has been placed in your code. No errors have been placed in this code on purpose and it has been tested extensively.

If you would like to see this code run, try removing the EEPROM or ground the SCL or SDA line. This will force an error which is handled by this code.

You should recognize the “goto \$” code. It prevents further execution past that line.

You may have noticed that this code is not “called” and “returned” from so it is not technically a subroutine. It has been separated from the main code and placed here to make the code explanation easier to understand.



I²C Code - Section 12 of 12

```
; This routine sends the W register to SSPBUF, thus
; transmitting a byte. The SSPIF flag is checked to ensure
; the byte has been sent. On completion, the routine exits.
Send_I2C_Byte
    BANKSEL SSPBUF          ; BANK 0 ← Send Byte Subroutine
    movwf  SSPBUF          ; Get value to send put in SSPBUF
    retlw  0               ; Done, Return 0

; This routine waits for the last I2C operation to complete.
; It does this by polling the SSPIF flag in PIR1.
WaitMSSP
    BANKSEL PIR1           ; BANK 0 ← Wait Subroutine
    btfss  PIR1,SSPIF     ; Check if I2C operation done
    goto  $-1             ; I2C module is not ready yet
    bcf   PIR1,SSPIF     ; I2C ready, clear flag
    retlw  0               ; Done, Return 0

END ← End of Program
```

Getting Started: I²C Master Mode

© 2001

This is the last slide to review for this code example. It contains two very small subroutines that we have used extensively in the main code. Lets look at them now.

“Send_I2C_Byte”, simply takes the value currently in W and places it in SSPBUF. If all other conditions are correct, this will send a byte. Remember that W was loaded with a value, which you probably noticed was from one of the #defines used at the beginning of the code. It ends with “retlw 0” which writes 0 to W and then exits the subroutine. The 0, indicates “no errors”, but is not important in this case.

The “WaitMSSP” subroutine waits for an MSSP action to complete. To do this, we poll the “SSPIF” flag. SSPIF stands for “Synchronous Serial Port Interrupt Flag”. This flag is set by the MSSP module when *any* MSSP action completes. So, polling this flag is used to wait though this code. However, notice that the flag is cleared again after we discover it has been set. This *must* be done in order to use it again. The PICmicro will only set this flag, it is up to the user code to clear it. This is the case with many interrupt flags. Check the data sheet for details on these flags. Again, since it is a subroutine, “retlw 0” is used to end the routine and return to the main code.

This completes the code review. Next, we will look at the waveforms you should see, complete with sample oscilloscope traces.



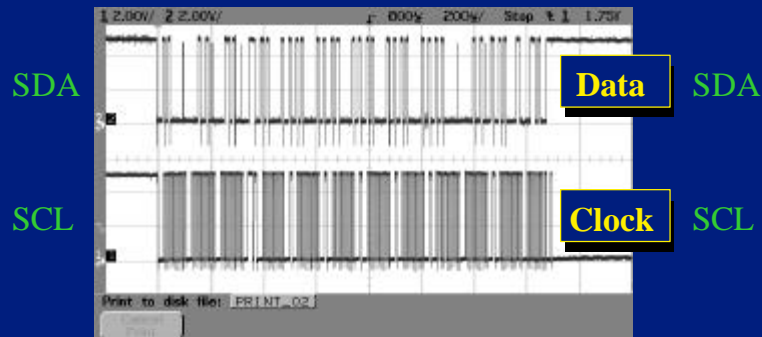
I²C - Example Waveforms

- The following slides show the waveforms produced from the sample code.
 - Writing to EEPROM
 - Reading from EEPROM
 - I²C Bus Conditions (Start, Stop, Restart)
 - Waveform Close-ups

The next slides in this presentation will look at what you should see if you look at the SCL and SDA lines during the data transfer. It is recommended that you try this as it will be helpful to learning I²C.

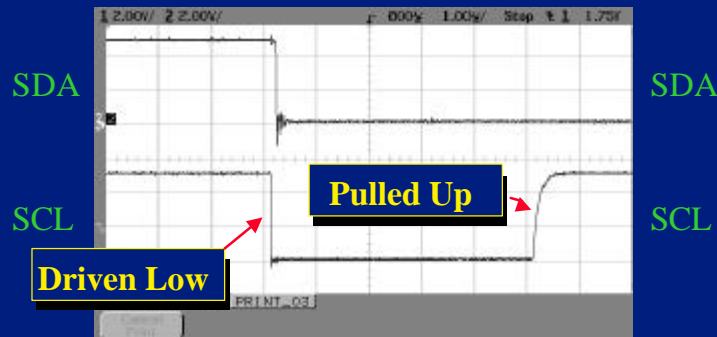
I²C - Example Waveforms

- Complete Transfer



Here is an overview of the transfer. It is not easy to understand what is happening yet, but if your results look like this, you are probably doing well.

- Close-up of Waveforms



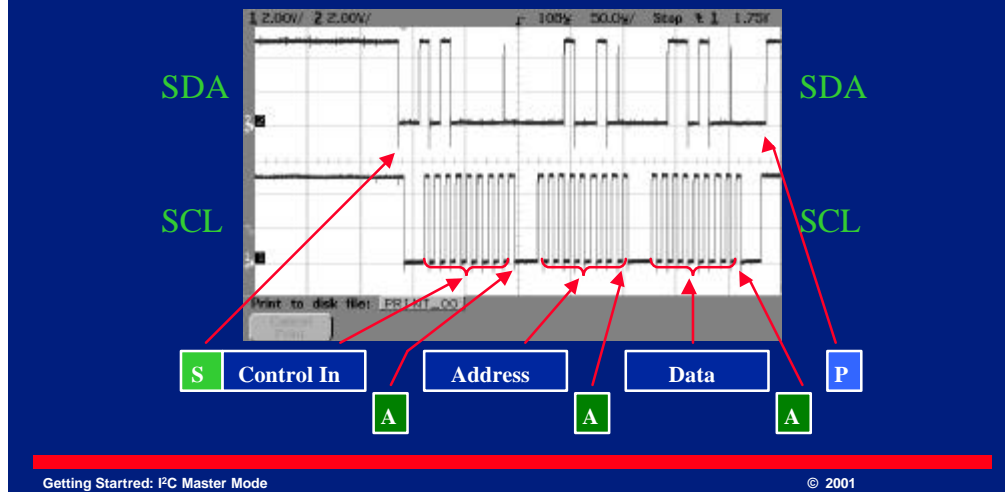
Here is a greatly magnified look at a high and a low. When the I²C lines are driven low, the signal falls, and often quite rapidly. There may be some ringing as you can see, but this can be tweaked in your system if needed. Note that when a signal floats high, the slope is quite gentle. This slope will depend on your pull-up resistor. It is a result of the pull-up resistor and the capacitance of the I²C line. If you choose too large of a pull-up resistor for your application, you may notice the slope is so slow, the signal never becomes high, or the results may be ambiguous. If the pull-up is too small, the I²C drivers may have trouble driving enough current to compensate and your average current consumption will rise as well.

Pull-up values were suggested earlier in this presentation, but the results were only a suggestion, it is these waveforms that will determine the success of your I²C. It is also this that limits your distance and speed. As you try to increase distance, you will increase capacitance. As capacitance increases, so will the time constant and the rise curve will get longer. If it gets too long, the data can not be read reliably as the signal will not rise high enough. This also limits speed, as to go faster, this rise must happen faster.

As a general rule, I²C is not made for long distances. Keep it on one PCB or at least in one box. If you want to go longer distances, slow down and use smaller pull-ups, and remember there are limits to its drive capabilities.

I²C - Example Waveforms

- Writing a Data Byte To the EEPROM



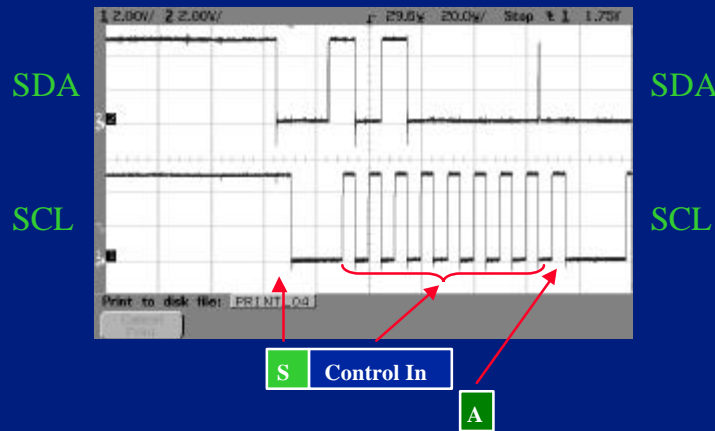
Here is a close-up of the writing to the EEPROM. The diagram points to the various conditions that make up the write transfer sequence. Note, even though the arrows appear to only point to the SDA or SCL signals, both the SDA *and* SCL are part of each condition. For example, data is not data unless there is a clock for it, and a Start condition is made up of the timing of *both* SDA and SCL.

If you look at the results on your oscilloscope, you should be able to see a similar result here and be able to recognize each condition.

Have you noticed those spikes near clock pulse number 9? Consider them as a low for now. We will look at them in detail shortly.

I²C - Example Waveforms

- Start Condition and First (Control In) Byte

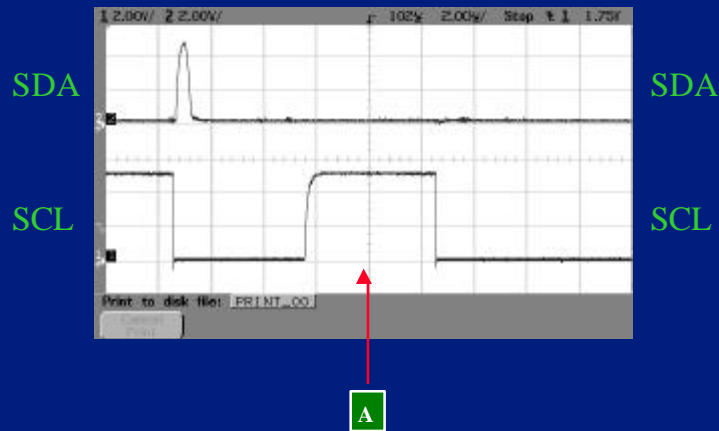


Here is a close up of a start condition, the control in byte and its ACK. You can clearly see the data now. After the start, SDA goes high, then low, then high again. Then it remains low until the spike. This translates into a data bit at each clock pulse. At the first SCL pulse, SDA is high, so this bit is a 1, then it goes low for the next pulse so this is a 0, and so on. You should be able to recognize the binary value of 1010 0000, or 0xA0 (hex).

Have you seen this before? It is used in the code used to define a “Control In”. Remember the first thing we did when sending a byte was to send a control in byte. A control in byte was defined in the “#define” section of the code as 0xA0 hex. Clearly things are working well.

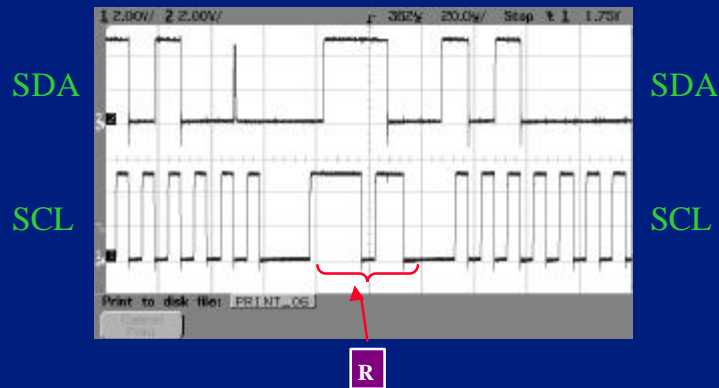
The 9th bit is the ACK from the EEPROM. The reason for the spike is that the PICmicro MCU releases the data line so that the EEPROM can answer. If the EEPROM pulls the line low, its an ACK, if it floats high, its a NACK. The spike is caused due to the fact that as soon as the PICmicro MCU releases the data line, it starts to go high. A very short time later, the EEPROM pulls it low. This happens to be a short time, but this is the reason you see a spike here. The important thing to I²C is that the line is low when the clock is high, and this condition is satisfied. What it does until then is not very important.

- Close-up of ACK Signaling



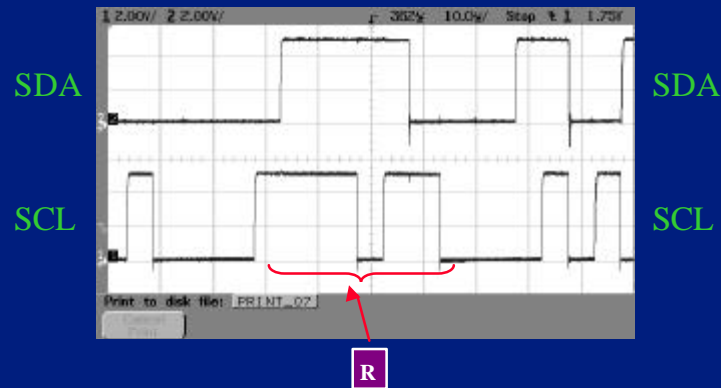
Here is a close up of the ACK pulse and the nearby spike we have mentioned. Notice that it only happens when SCL is low. Remember that data only has to be valid when SCL is high. This is part of the definition of I²C protocol. As you can see, this does happen and so the ACK will be correctly recognized. If you have trouble with your I²C system, be sure to check this. *The data must be valid when SCL goes high.*

- Restart Condition



Here is a look at the restart condition. This is the first restart that occurs in the waveform. It happened after the write to the EEPROM completed and we begin the first part of the ACK polling. You can see the start and stop conditions contained within it.

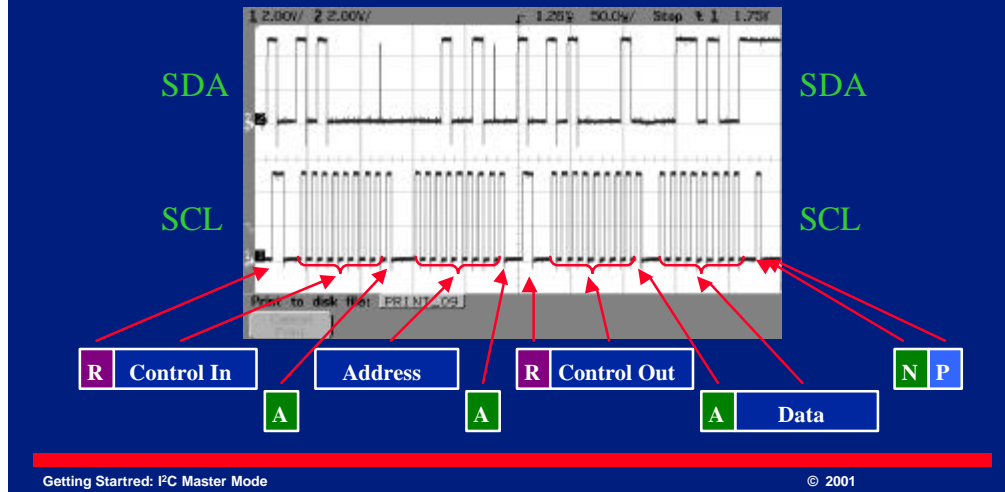
- Close-up of Restart Condition



Here is a closer look at the same restart condition. You can now clearly see the stop and start condition contained within it. Remember, a stop condition is a release of the SCL line followed by a release of the SDA line. A start condition is when SDA is pulled low, followed by SCL. Restart conditions may sometimes look different from each other, but they must contain a stop followed by a start condition by definition.

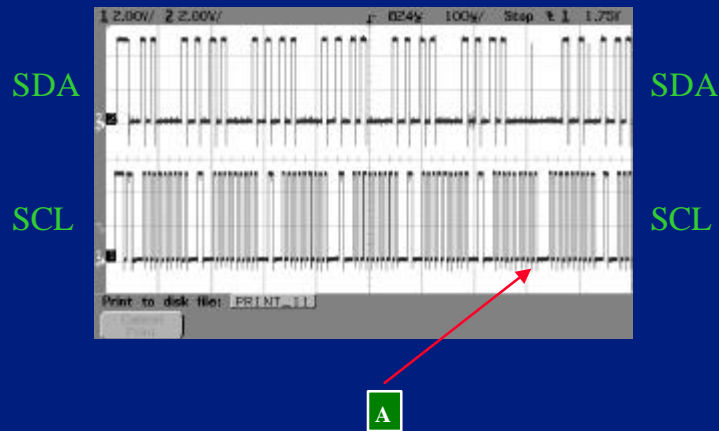
I²C - Example Waveforms

- Reading a Data Byte From the EEPROM



Here is a look at our EEPROM read waveform. Shown here is the first control byte that is ACKed by the EEPROM. None of the ACK polling is shown here. You can see the Control in byte, which has the value of 0xA0 (hex). You can also see the address byte and data byte. Remember we wrote 0x34 (hex) to address 0x12 (hex).

- ACK Polling

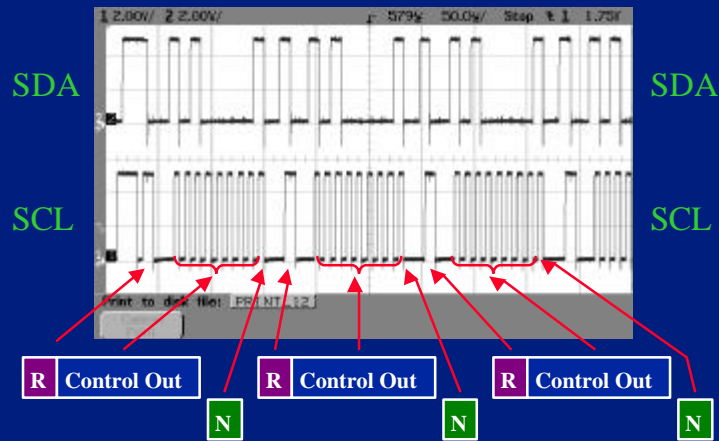


Here is a look at the ACK polling results. Its a bit hard to see from this diagram, but each of the polls has returned a NACK from the EEPROM. Only the one being pointed to returned an ACK, thus ending the polling. The polling when tested went for many cycles, many more than shown here, and will vary. However, ACK polling gives you the fastest possible response from the EEPROM because you can know immediately when it is no longer busy.

Lets take a closer look at the ACK polling now.

I²C - Example Waveforms

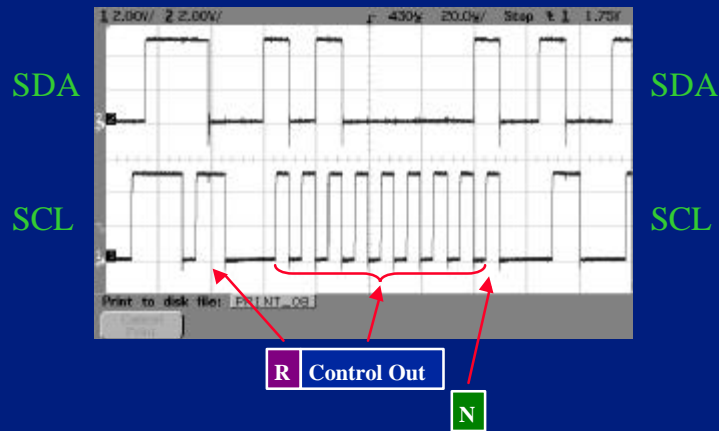
- Close-up of ACK Polling



Here is another look at the ACK polling. Each write of the control out byte is returning a NACK, thus indicating the EEPROM is not ready for new commands at this time.

I²C - Example Waveforms

- ACK Polling - Device Returned NACK

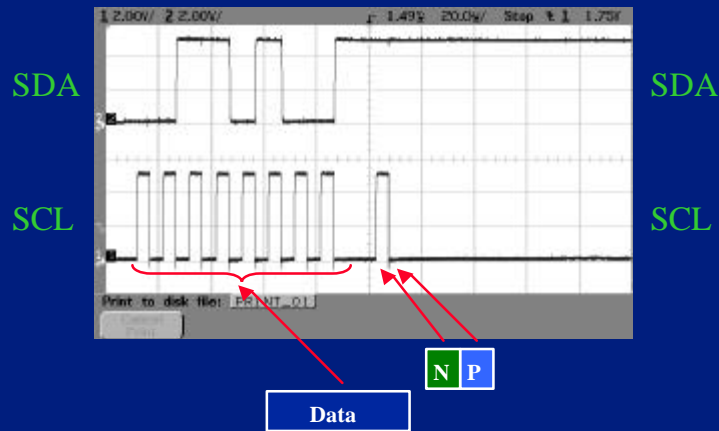


Here we have zoomed in on one loop. You can clearly see the restart, the control out byte, and the NACK. The NACK is now easy to see. Notice that on the 9th clock pulse the SDA line goes high. This indicates the bit is a 1, which signifies a NACK response.

If this bit were low, it would signify a 0, which is an ACK response.

I²C - Example Waveforms

- NACK and STOP at End of Transfer



This slide has zoomed in on the last byte in the transfer. It contains the data that was retrieved from the EEPROM, the NACK sent by the PICmicro and the stop condition. Once these signals have finished, the bus is idle as there is no more data or conditions on the bus. This can be seen from the right of the stop condition. Once the stop condition completes, we have finished communicating with the EEPROM.



I²C - For More Information

- **More I²C Resources:**
 - ***PICmicro Data Sheet***, (example: “PIC16F87x Data Sheet,” MSSP Chapter).
 - ***Reference Manuals*** (MSSP Chapter).
 - **Application Notes *AN734, AN735 and AN578***.

Getting Started: I²C Master Mode

© 2001

Listed here are some more resources that you can take a look at. These items are all available on the Microchip Technology web site. Visit: “<http://www.microchip.com>” to locate these documents.

AN578 - Description: Use of the SSP Module in the IIC Multi-Master Environment

AN734 - Description: Using the PICmicro SSP for Slave I²C Communication

AN735 - Description: Using the PICmicro MSSP Module for [Master Mode] I²C Communications