BIT BANGING I²C
FOR
PIC PROCESSORS
(Preview)

BIT BANGING I²C
FOR
PIC PROCESSORS

(Preview)

David W. Hoffman

First Edition

Copyright © 2005 David W. Hoffman

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means without written permission of the author.

I^2C™ is a trademark of Phillips Corporation.
MPLAB IDE™ is a trademark of Microchip.

Contents

INTRODUCTION ... 1

- BIT BANGING – DEFINED ... 1
- CODING CONCERNS WITH THE ... 2
- 16C5X & 16F5X PROCESSORS ... 2
- DEVELOPMENT ENVIRONMENT ... 3
- OBSERVING A PIC IN ACTION. ... 3

SECTION ONE – SERIAL COMMUNICATION CODE DESIGN ... 5

SECTION TWO – BUS CONTENTION ... 9

- METHOD ONE – MULTIPLE I/O LINES ... 10
- METHOD TWO – SINGLE I/O LINE ... 11
- METHOD THREE – MASTER CONTROL DEVICE ... 12

SECTION THREE - BIT BANGING I^2C ... 15

- THE I^2C PROTOCOL ... 16
- I^2C BUS CONTENTION ... 17
- THE HEADER FILE FOR I^2C COMMUNICATION ... 18
- I^2C CODE ... 19
  - ***Call Table*** ... *20*
  - ***I2C_START*** ... *21*

About The Author

David W. Hoffman is an avid electronics enthusiast with a diploma in electronics design and troubleshooting and he enjoys designing electronic circuits. David currently lives in south Florida with his wife Elisabeth.

To my loving wife, Elisabeth.

Introduction

Programming PIC micro-controllers is a very rewarding and challenging endeavor. Adding peripheral devices to PIC projects can be even more challenging when using serial communication. This book tackles one of the most popular serial protocols, I^2C, by providing working code and explaining how that code accomplishes serial communication.

The code presented in this book is fully functional and has been tested to insure that it's usable. However, it's not practical to test this code with every single serial device out there. So modifications and problems are bound to arise. If you ever have issues getting this code to work you're more than welcome to contact me and I will do my best to help resolve the issues you are facing.

This book is intended for the novice PIC programmer not familiar with serial communication. However, it is assumed that you are familiar with assembly, have a PIC programmer or access to one, and that you have a development environment like Microchip's MPLAB IDE. Writing code without such a development environment is sure to cause you frustration. I highly recommend taking the time to download this free development tool from Microchip's website.

You should also locate and download the datasheets mentioned in this book. All the devices mentioned in this book are modern and you shouldn't have any trouble locating the datasheets for these devices on the manufacturers website. Having the datasheets on hand will aid in understanding the code presented in this book.

Bit Banging – Defined

What exactly is "Bit Banging"? Well, not all processors have built in serial support and in these instances designing your own code to implement serial communication is required. This kind of coding is called bit banging. For example, the PIC16C5x and 16F5x devices do not have any kind of serial support built in. So

to add serial devices to a project you have to create code to handle the communication. The advantage to designing your own code is that you can add serial communication to any processor! You will also gain valuable coding experience and this will help you write better programs for future projects! Here are a few more of the advantages and disadvantages:

Advantages:

- You can implement projects using low-cost RISC processors.
- You have complete control over every aspect of the communication process and can customize this process to suite your needs.
- You can mix devices from different protocol families.
- You can implement as many or as few device commands as you like.

Disadvantages:

- You have to code all communication processes and this takes valuable memory.
- You are responsible for all timing considerations. (more on this later)

Coding Concerns with the 16C5x & 16F5x processors

Microchips 16C5x & 16F5x family of RISC based processors are powerful tools for the hobbyist. Providing the ability to reduce project circuitry and adding intelligence to projects. But they do have some shortfalls. Most notably is that the CALL command is limited to the first 256 bytes of memory in each bank, this also applies to ADDWF PC. In fact with the exception of the GOTO command any modification of the PC sets bit 8 to zero. This can present a serious problem when large programs need to be implemented. For this reason all the code presented in this book takes this limitation into account.

To get around this problem most of the code will only use the GOTO command allowing the code to reside anywhere in mem-

ory. To access routines a call table is created and resides in low memory, below the 256-byte limit, providing a bridge to the high memory routines. This will use only a few bytes of precious low memory reserving that space for application code.

The other side benefit to this is that all calls to the serial I/O code will only require a single byte of the two-byte stack. This allows sub-routines in main memory to execute a call to the serial I/O code. As long as the two-byte stack isn't exceeded the program will be able to return from the routines problem free.

Development Environment

The code presented in this book is in assembly and has been developed in the Microchip MPLAB IDE software. This book assumes you have this software (Available for free from Microchip and I highly recommend downloading this fantastic free development tool!) or a development environment similar to MPLAB IDE. This book also assumes you know how to use this software and makes no attempt to explain the use of MPLAB IDE.

Observing a PIC in Action.

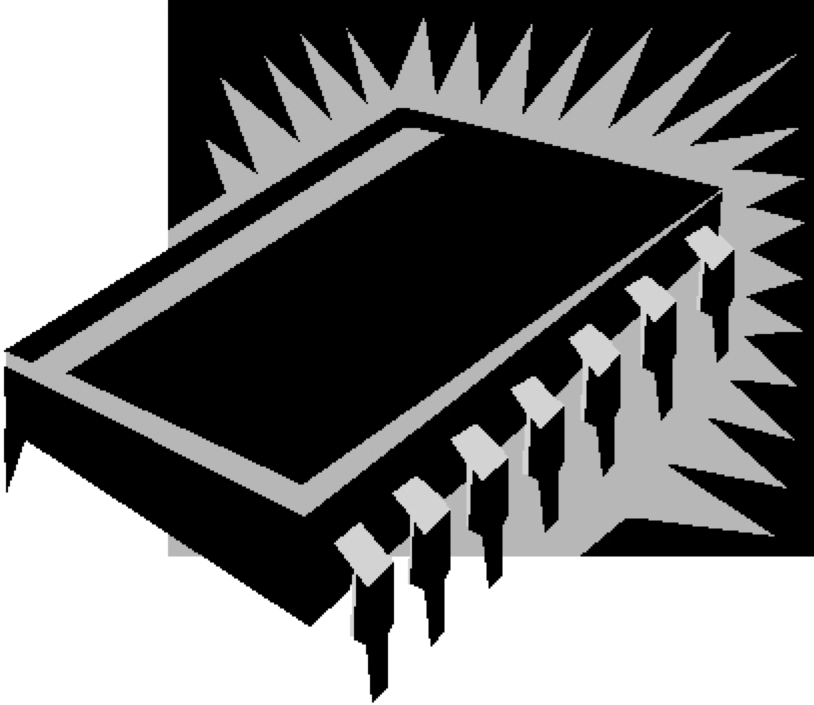
Being able to see your PIC work is of great benefit. It's very useful to see the PIC actually sending and receiving data and also having a way to display the status of the program. The easiest way to accomplish this is to wire up an LED to each pin on the PICs ports. But let me offer some words of caution doing this. According to the datasheet for the PIC16C5x PICs any pin on a port can only provide 20 ma of current and the port can only provide a maximum of 40 ma of current. If you wire in a single LED to any pin on any port it will draw about 7 ma (at 3.33 V with a 470 Ohm resistor). This neither exceeds the pin or port max. But if eight LEDs are wired in when all eight are lit they will draw 56 ma exceeding the port maximum of 40! This can and probably will damage the PIC.

To avoid this there are several solutions available. The most easily implemented is to use an 800-ohm resister to connect the LEDs. According to Ohm's Law this will only require 4 ma of current per LED and a maximum of 33 ma on the port if all 8 LEDs are lit. This doesn't exceed the port maximum of 40 ma but it also doesn't leave much power to run other devices. This method works fine if you're just using that port to observe data.

However, if you are looking to operate more than a few devices from a single port and you still need to watch that data using LEDs then another method needs to be implemented. Some solutions are:

- Using NPN transistors to act as switched. This greatly reduces the current draw on I/O pins and the port as a whole.
- Using a buffer IC such as a 74LS533 Octal D-Type Transparent Latch. This is a non-inverting latch allowing data to flow freely through it and also acts as a buffer. Another benefit this IC has is it can latch the data passing through it at any time allowing you to more closely observe a specific data state.

Proper calculation of current needs is extremely important. If you exceed a devices limits you can and most probably will damage or even destroy that device. Something to seriously consider when working with an EEPROM PIC that costs upwards of $15. One can never be too careful.



Section One – Serial Communication Code Design

Serial communication has tremendous advantages over parallel communication. Of course there is the obvious disadvantage of data transfer rate but parallel data transfer has many dedicated I/O pins used for memory addressing and data transfer. This can quickly turn a simple project into a major one just in the amount of additional wiring required. For example, a 512-byte parallel EEPROM required 8 transmission lines for the address and 8 transmission lines for data. That's a total of 16 wires going from your PIC to your EEPROM. And this isn't even considering the control lines that are required.

Serial communication reduces the number of connections to just a few leaving I/O bits free for other uses. Serial devices available include EEPROMs with Write/Erase cycles as high 1 million. Port expanders, AD Converters and much more. These are only a few of the many advantages serial communication has over traditional parallel communication. Learning how to

implement serial devices in your circuits will add functionality taking your projects in directions you probably would have never otherwise considered.

Implementing serial communication in a processor that does not have built in support can be challenging. Many issues have to be considered such as timing, transmitting the correct number of bits, receiving the correct number of bits, and starting and stopping serial communication sessions correctly. These are many of the issues that need to be considered and handled but certainly not all of them. This book is intended to provide you with the knowledge and tools needed to implement I^{2}C serial communication in any processor that does not have built in serial support.

It's also imperative to design the code so that the implementation of various serial devices is possible. It's just not practical to design the code to support a single device and then have to make major changes to support additional devices. And considering the protocol covered here, flexibility is of great importance. There are many types of I^{2}C peripheral devices available on the market. And this makes I^{2}C a very attractive protocol to implement in projects.

However, the code presented in this book is designed around serial EEPROMs. But since every single device using I^{2}C will permit the code, developed here, to be easily integrated into using other devices. In other words this code will easily support other devices in addition to EEPROMs. The code is designed around the implementation of several steps to complete an I^{2}C communication session, these are:

- Send the start bit to the device.
- Send the required command byte(s).
- Send the address (if required).
- Send or receive the data byte.
- Send the stop bit to the device.

Starting from these basic building blocks the code is broken down into three major files, they are:

- **Header File** – The header file contains all the declarations for variables and labels. This allows changes to the code to take place by simply altering the value of a variable. Take for example the variable I2C_SDA, which holds a value from 0 to 7. This number is used to indicate which bit on the I/O port is to be used to send and receive data. Just as the variable SDA_CLK is used to specify which bit on the I/O port is used to send out clock pulses. By changing the values of these variables you can quickly change which bit or port the core code uses to communicate with serial devices.
- **I^{2}C Core Code** – Consists of a number of routines designed to handle communication with serial devices. These routines are in their own file so that they can easily be imported into projects.
- **I^{2}C Implementation Code** – is the code required to implement the core routines in your projects.

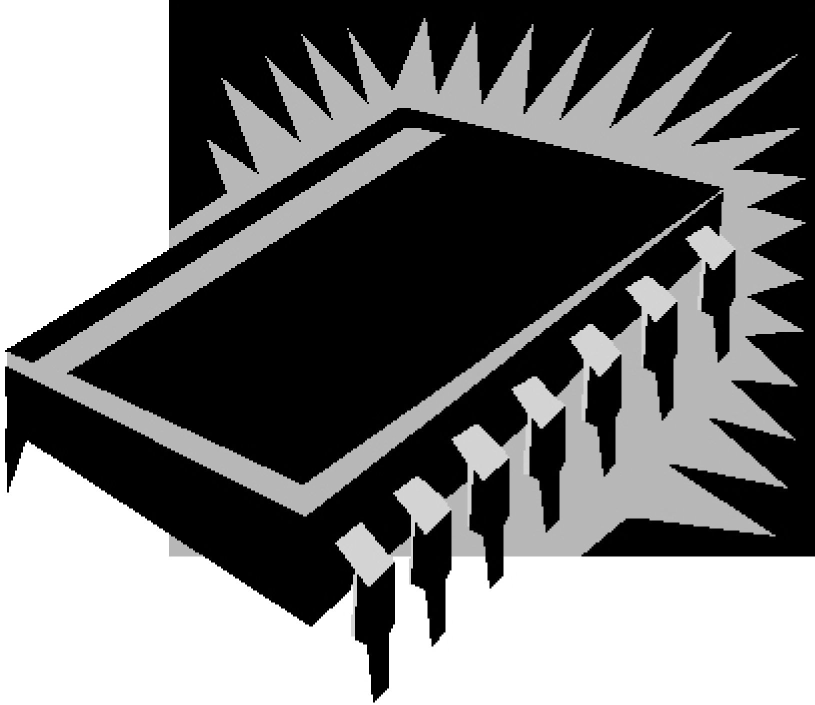
The core routines are designed to simply send and receive an eight-bit word. It's up to the main code, your code, to decide what is being sent. This allows you to communicate with devices that require a single or several commands. This in turns allows you to implement any size EEPROM required.

For instance, say you are communicating with a serial EEPROM that requires only a single address byte. You'll only need to send the command word, followed by the address byte and then send or receive the data byte. Using this method allows you control over how much data is sent to the serial device. All you need to do is make changes to your main code. Rarely will you need to make changes to the serial communication core code.

However, with every rule there is an exception and in this case this holds true. You will need to make changes to the core code when implementing devices that use different commands than those presented in this book. Although nearly every EEPROM, regardless of size, uses the same commands to send and receive data this isn't true for other devices. These other devices use different commands and this means that changes will need to be made to the code so that these devices can be

properly implemented. But don't fret, this topic is covered in greater detail later in the book.

The last element that needs introduction in the core code is the call table. The call table is implemented due to a limitation of the PIC 16C5x series processors, discussed in the previous section. The call table is a chunk of code that exists at and below the 255-byte barrier. This code only consists of some simple configuration commands and GOTO commands that jump the program to the appropriate code higher in memory. This allows the majority of the serial core code to exist in high memory saving low memory for application code and data lookup tables. It's important to note that only the last command being called needs to exist at address 255 (0xFF Hex) as the RETLW command will properly return to the calling routine regardless of where RETLW exists in memory.



Section Two – Bus Contention

One of the most challenging problems you'll face is the possibility of bus contention when using more than one processor in a circuit. Bus contention occurs when both processors try to access the serial bus at the same time. This is a serious problem with any protocol and can cause chaos with serial access. It comes down to you to make sure that your multiple processor environment is free from the possibility of bus contention.

Imagine that you have two processors in a circuit each serving a specific task. Both processors are capable of accessing a single EEPROM for storing and retrieving information. Since both processors are not equipped with built in serial support neither has a way to know when the other processor is accessing the serial bus. The result is that both processors can access the bus at the same time causing miscommunications and disrupting proper operation throughout the circuit. Certainly not a situation any designer wants to face.

There are several possibilities for resolving this issue. The first relies on the use of multiple I/O line on a PIC port to inform and be informed when the serial bus is in use. The second relies on a single I/O line to inform and be informed of bus activity. They both have pros and cons.

Method One – Multiple I/O Lines

The first solution uses extra I/O lines to communicate a processor's state to other processors. One I/O line is used for each processor to indicate its status. So if there are two processors in the circuit each would have two I/O lines dedicated to this communication. Each processor will indicate when it's using the bus by bringing its I/O line low. When it's not using the bus it will bring its I/O line high. But either processor can only use the bus after checking the state of the *other* processor first. For example:

Let's call our two I/O lines P1 and P2 and are assigned to each processor respectively. Both lines are connected to both processors so they can communicate their state. If processor one wishes to transmit or receive data on the serial bus it must first check the state of the P2 line. And if P2 is low then processor two is currently using the serial bus. Once P2 come high processor one will be allowed to access the bus. Once processor one detects P2 high it immediately brings it's own control line P1 low so that processor two will know that the serial bus is in use.

Although this design seems to eliminate bus contention it is still possible, although remotely so, for both processors to fall into accessing the serial bus at the same time. Take for example the instance where both processors are near to accessing the serial bus at the same time. Processor one checks P2 and finds it high but before processor one can bring P1 low processor two checks P1 and finds it high. Both will now think that the serial bus is free when in fact it's not.

One possible solution is, for example, processor one to re-sample P2 several times after bringing P1 low. If during this period P2 goes low then P1 knows that P2 is confused and that

P1 needs to relinquish control of the serial bus. Repeated sampling of the control lines is probably the only way to insure that both processors won't fall into this rare possibility.

Another solution to this possibility is to constantly check the state of P1 and P2 and if at some point one goes low when it shouldn't you can have both processors abort the sequence and re-negotiate who has control. Another is to actually sample P1 or P2 multiple times to insure that it the serial bus is indeed free. Perhaps even re-sampling P1 or P2 after claiming control of the bus to be sure that there isn't contention between the two processors.

The main advantage to this method is that it requires less code than the next method presented. The disadvantage is that it increases the number of connections in the circuit. If you're only using two processors this method should work fine but more than two and the number of additional wires can quickly defeat the purpose of using serial devices.

Method Two – Single I/O Line

With this second method the multiple control lines are replaced with a single line. A pull-up resister is used to keep the line high when not in use. Since this single data bit is tied to all the processors in the circuit the amount of wiring is greatly reduced over the previous method. The drawback is more coding is required to implement this method and a close look at how it will be implemented brings up the possibility of still facing bus contention.

For any processor to take control of the serial bus it must first check the state of the control line to see if it's high. If so then the processor brings the line low, by putting a low on that bit. This indicates to all other processors that the serial bus is in use. This works in theory but the possibility of bus contention still exists. Take for example P1 looking at the control line and seeing it high. This processor then has to execute a number of instructions to change the I/O state of that bit and bring the control line low. During this period of time another processor

may look at the control line and see that it is still high. This second processor then starts executing code to bring the control line low. So this method alone is still not a solution.

To resolve this possibility each processor needs to enter a queue and this can be accomplished by forcing each processor to check the control line multiple times. And if the control line remained high during that period of time the processor may then take control of the bus. A simple loop will accomplish this:

```
            CLRF COUNTER
LOOP        BTFSS CONTROL_BIT
            GOTO B_BUSY
            INCFSZ COUNTER
            GOTO LOOP
            ;CODE HERE TO TAKE CONTROL OF THE BUS
B_BUSY      ;BUS IS BUSY WAIT FOR BUS TO BE FREE AND
            ;THEN TRY AGAIN.
```

With this code the possibility of multiple processors taking control of the bus is greatly reduced but not completely eliminated. To eliminate bus contention completely one processor has to be declared the master. Someone has to have control over who has access to the serial bus and who doesn't and this can be accomplished in one of two ways:

Method Three – Master Control Device

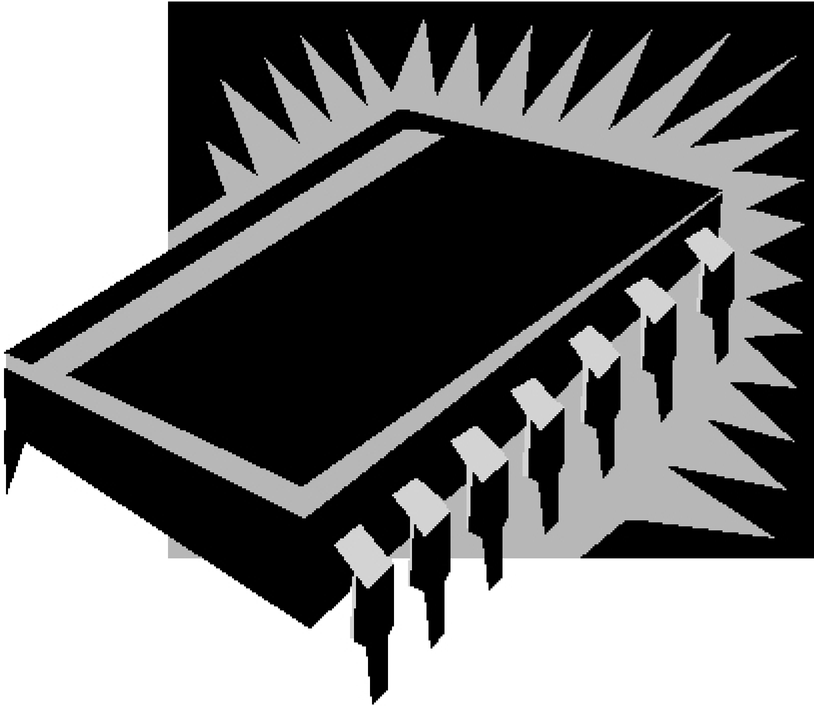
By assigning one processor master control all others will have to bend to the master's will. The master processor uses an I/O bit to communicate to other processors who is in control. Using this method requires a substantial increase in wiring. Every single processor has a control line going to it from the master. The master continuously cycles through the other processors giving each a chance to take control of the serial bus. Once the master detects activity on the serial bus whatever processor initiated that activity has control. Once activity ceases on the

serial bus the master then give the next processor a chance to use the serial bus and so on…

Since the master device has ultimate say of which processor has control and when there is no chance of bus contention. The drawback: Increased wiring is required and a PIC has to have at least a portion of the available resources dedicated to the control code. There is however another option.

Combining a binary counter with a binary to decimal decoder can produce the required control circuitry. For a PIC to take control of the bus it must first check the control line it is assigned and see if it's low. If so it can then bring low another line, let's call it BUS_BUSY, that is wired to an AND gate. This AND gate controls the clock signal going to the counter. Once BUS_BUSY is low the processor then checks the control line to be sure it is indeed still low. This tells the processor that it took control of the serial bus in time and that that control didn't slip by while it was executing code. If the control line is high then the processor knows it missed it's chance and has to bring BUS_BUSH high again. This allows the clock signal through the AND gate and the polling continues. The processor will have another shot at the bus shortly once its control line is activated again.

Having a device designated as the controller of serial bus activity is probably the safest way to implement a multiple processor circuit. The only drawback is the additional circuitry and code required to implement multiple processor serial access. In fact, you may find it easier to just use PIC devices with built in serial support. These problems are eliminated with these processors and make multiple processor environments easier to implement.



Section Three - Bit Banging I^2C

Coding from scratch to take advantage of I^2C devices isn't that hard. In fact, once the basic code is developed it's very easy to modify it to take advantage of other I^2C devices. This chapter will cover all the required code for communicating with I^2C devices.

I^2C, developed by Philips Semiconductors, uses a simple communication protocol to access peripheral devices on the bus. There is a master device and a slave device. This exercise assumes that the master device is always the PIC16C55 and that the slave device is a Microchip 24C01C serial EEPROM (1k bit EEPROM configured in a 128 x 8 matrix).

The I^2C Protocol

I^2C uses just two wires to implement serial communication between one or more master devices and one or more slave devices. Communication is initiated with a START bit (Which is a START condition generated by bringing SDA low while CLK is high) and then is followed with a Control Byte (Figure 1). The control byte consists of the control code, device address, and the read/write bit. The control code will always remain the same for the device being accessed and in this case is 1010 for the serial EEPROM. The device address can range from 000 to 111 allowing up to seven different devices on the same bus. The read / write bit tells the device which operation is being performed. Device addressing is accomplished by tying pins high or low on the device itself. There are three pins (A0 – A2) on the 24C01 serial EEPROM that allow us to determine its address on the bus. Tie all the pins low and the devices address is 000. Now Tie pin A0 high and keep the other two low and the address becomes 001. And 010 if pin A1 is high and pins A0 and A2 are low. Using this method any address between 000 and 111 can be assigned to any device. And this allows multiple devices on the same bus.

For example, let's say you need to send a write command to device 010. The control byte would look like this 10100100. Once the START bit is sent the control byte is transmitted on the bus. Only the device with the address 010 will acknowledge receiving the control byte with an ACK. A complete write session (Figure 2) consists of a START bit, command byte, ACK, address byte, ACK, data byte, ACK, and the STOP bit.

There are two ways you can read a byte from the device. The first consists of sending a control byte with the read/write bit set. This will allow the retrieval of the data byte located at the address of the last operation + 1. For example, if you write a data byte to address 00001010 inside the EEPROM doing a read will return the contents located at address 00001011.

To read from a random address you will first have to set the address pointer inside the EEPROM to the address you wish to read from. This is accomplished by first sending out a write

command byte followed by the address you wish to read. Next a START bit is generated and then the read control byte is sent out. Now the data byte is read from the device at the address specified in the write part of this session (Figure 4). Note: An ACK is not generated after reading the data byte.

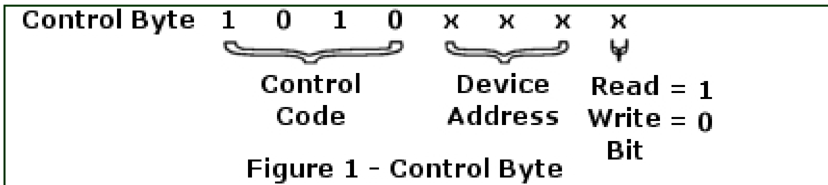


Figure 1 - Control Byte

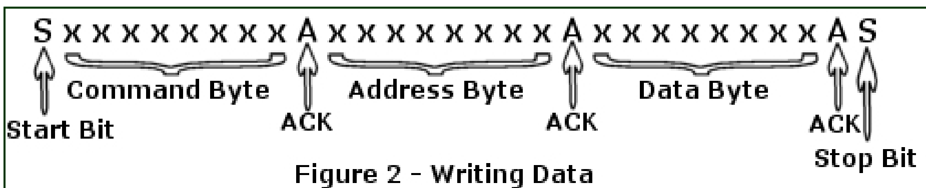


Figure 2 - Writing Data

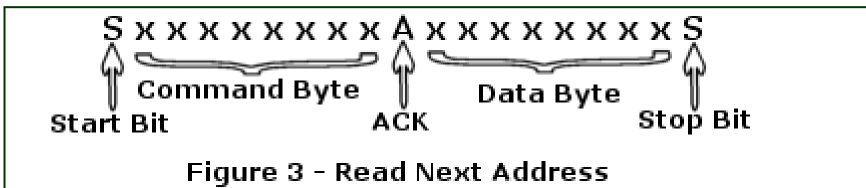
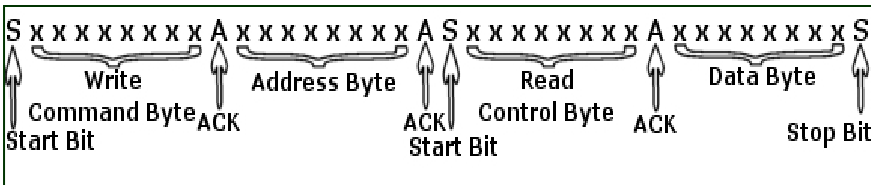


Figure 3 - Read Next Address



I^{2}C Bus Contention

If you're planning on using more than one master on a bus then it is highly recommend using processors that have built in I^{2}C support. When working with a single processor you will always know who is in control of the bus, the one PIC in the circuit. However, if you have two processors then the possibility of bus contention arises as discussed in section one. But if using processors with built in serial support is not possible then using one of the methods discussed in section one is certainly recommended.

The Header File for I^{2}C Communication

Go ahead and create a new project in MPLAB and then create a new file. This will become the header file for the I^{2}C code. Place the code listed below in the header file and call it "i2c_128b.h". The code listed in this chapter required the use of six registers and is designed to be as configurable as possible. After all, this is cut and paste code, and is intended to be inserted into many PIC projects.

```
I2C_EEPROM_W    EQU   0xA0
I2C_EEPROM_R    EQU   0xA1
I2C_CLK         EQU   0x00
I2C_SDA         EQU   0x01
I2C_PORT        EQU   0x06
I2C_TRIS        EQU   0x1B
I2C_DEVICE      EQU   0x1C
I2C_COUNT       EQU   0x1D
I2C_STATUS      EQU   0x1E
I2C_IO          EQU   0x1F
```

Let's break these down each label and see exactly what they are for:

- **I2C_EEPROM_W** Write command byte for EEPROM
- **I2C_EEPROM_R** Read command byte for EEPROM.
- **I2C_CLK** Bit on port I2C_PORT used to send clock pulses.
- **I2C_SDA** Bit on port I2C_PORT used to send and receive data.
- **I2C_PORT** IO port used to communicate with serial devices.
- **I2C_TRIS** Stores the input / output states of other bits on I2C_PORT.
- **I2C_DEVICE** Register used to store the device address.

- **I2C_COUNT** Register used to keep track of what bit is being sent or received.
- **I2C_STATUS** Register used to keep track of internal modes.
- **I2C_IO** Register used to store the date being sent or received.

You may be asking yourself why a register dedicated to the port state is required. Simply put this register is being used to keep track of the I/O state of the other bits on the port being used to communicate with the serial EEPROM. Since ports on the PIC are either 4 bits wide or 8 bits wide this leaves other bits that can be used for other tasks. And during a read or write process to the serial device the TRIS command will be used to change the SDA bit from data out to data in and back again. I2C_TRIS is used to preserve the states of these other bits. For example: Say bits 0 and 1 on PORT B are being used to communicate with the serial EEPROM. And bits 2 – 7 for other tasks. Some are input and some are output. Whenever a TRIS is used to change bit 1 to an input or output state during a communication session you don't want to change the state of the other port bits. They need to remain the same. By storing the state of the port in I2C_TRIS whenever a TRIS on I2C_PORT is done the state of the other bits will be preserved.

I²C Code

Here is the complete code you'll need to implement I²C communication using a PIC 16C5x processor. The total number of core commands consists of 81 instructions. This does not include the commands required to access the core routines. This code can easily be adapted for use in other PIC processors. Later in this chapter will discuss modifying this code to add support for other I2C devices. But first, here's a look at the code of each routine in detail.

Call Table

```
I2C_SREAD    BSF I2C_STATUS,7
             GOTO I2C_START
I2C_SWRTE    BCF I2C_STATUS,7
             GOTO I2C_START
I2C_READ     GOTO I2C_RWORD
I2C_WRITE    GOTO I2C_WWORD
I2C_STOP     GOTO I2C_STPB
I2C_WAIT     GOTO I2C_WAITA
```

This is the call table and consists of five commands. They are:

- **I2C_SREAD** is used to start a read process and consists of two commands. The first is setting bit 7 of I2C_STATUS and the second is jumping to the I2C_START routine.
- **I2C_SWRTE** is the same as I2C_READ except bit 7 in I2C_STATUS is cleared.
- **I2C_READ** jumps to the read routine I2C_RWORD and reads a data word.
- **I2C_WRITE** jumps to the routine I2C_WWORD and writes a data word.
- **I2C_STOP** jumps to I2C_STPB and generates the stop condition.
- **I2C_WAIT** jumps to I2C_WAITA and is used to delay further I2C operations until a write sequence is complete (more on this later).

The main purpose of the call table is to allow the majority of the code to exist above address 0x0FF. As discussed in chapter one on the PIC16C5x processors the CALL command can only access the first 256 bytes of memory in any bank but the GOTO command is not subject to this limitation. With this call table only 7 bytes of memory are used in the precious 256 low memory section. This reserves most of this memory area for other code or data lookup tables.

I2C_START

```
I2C_START       BSF I2C_PORT,I2C_SDA
                BSF I2C_PORT,I2C_CLK
                BCF I2C_TRIS,I2C_SDA
                MOVF I2C_TRIS,0
                TRIS I2C_PORT
                BCF I2C_PORT,I2C_SDA
                BCF I2C_PORT,I2C_CLK
                MOVLW I2C_EEPROM_W
                BTFSC I2C_STATUS,7
                MOVLW I2C_EEPROM_R
                IORWF I2C_DEVICE,0
```

I2C_START is responsible for generating the start condition, setting up the I/O port for data transmission and configuring the control byte and sending the control byte out to the I^{2}C device. Let’s look at each command in more detail.

- **BSF I2C_PORT,I2C_SDA** – Makes sure that I2C_SDA is high. I2C_SDA must be high for a successful execution of a start bit. I2C_SDA must be set high before I2C_CLK to in-sure a start or stop bit isn’t sent by accident.
- **BSF I2C_PORT,I2C_CLK** – Makes sure that CLK is high. CLK must be high for a successful execution of a start bit.
- **BCF I2C_TRIS,I2C_SDA** – Clear bit I2C_SDA in I2C_TRIS. This is done so that only the I/O state of I2C_SDA is changed when using the TRIS command I2C_PORT.
- **MOVF I2C_TRIS,0** – Move the contents of I2C_TRIS into the W register.
- **TRIS I2C_PORT** – TRIS I2C_PORT so that bit I2C_SDA is in output mode.
- **BCF I2C_PORT,I2C_SDA** – By bringing I2C_SDA low the start bit is generated.
- **BCF I2C_PORT,I2C_CLK** – Bring I2C_CLK low. I2C_SDA can only safely be changed when I2C_CLK is low.

- **MOVLW I2C_EEPROM_W** – First command of configuring the command byte. b’10100000’ is moved into the W register.
- **BTFSC I2C_STATUS,7** – Check the state of bit 7 in I2C_STATUS and if clear skip the next command.
- **MOVLW I2C_EEPROM_R** – This command is executed if bit 7 in I2C_STATUS is set and this means that a read operation is taking place. b’10100001’ is moved into the W register.
- **IORWF I2C_DEVICE,0** – The command byte is finished by merging the contents of the w register with the address of the device being accessed. This is accomplished by using the IORWF (Inclusive OR) command. The use of the IORWF command dictates that I2C_DEVICE be free of any set bits except for the device address bits (bits 3,2, and 1). All other bits in this register should always be clear! For example: If I2C_DEVICE has the value of 0x06 (b’00000110’) this means device 0x03 on the bus is being accessed. After the inclusive OR command the W register will contain the value b’10100110’ for a write command byte and b’10100111’ for a read command byte. If any other bits in I2C_DEVICE are set it’s possible to corrupt the command byte and cause a failure in the session.

END OF PREVIEW