## Table of Contents

### SECTION 1  INTRODUCTION

In-Circuit Serial Programming™ (ICSP™) Guide ................................................................. 1-1

### SECTION 2  TECHNICAL BRIEFS

- How to Implement ICSP™ Using PIC12C5XX OTP MCUs ........................................ 2-1
- How to Implement ICSP™ Using PIC16CXXX OTP MCUs ........................................ 2-9
- How to Implement ICSP™ Using PIC17CXXX OTP MCUs ....................................... 2-15
- How to Implement ICSP™ Using PIC16F8X FLASH MCUs ...................................... 2-21

### SECTION 3  PROGRAMMING SPECIFICATIONS

- In-Circuit Serial Programming for PIC12C5XX OTP MCUs ......................................... 3-1
- In-Circuit Serial Programming for PIC12C67X and PIC12CE67X OTP MCUs ............. 3-15
- In-Circuit Serial Programming for PIC14000 OTP MCUs ........................................... 3-27
- In-Circuit Serial Programming for PIC16C5XX OTP MCUs ......................................... 3-39
- In-Circuit Serial Programming for PIC16C6XX/7XX/9XX OTP MCUs ....................... 3-51
- In-Circuit Serial Programming for PIC17C7XX OTP MCUs ........................................ 3-71
- In-Circuit Serial Programming for PIC18CXXX OTP MCUs ........................................ 3-97
- In-Circuit Serial Programming for PIC16F62X FLASH MCUs .................................... 3-135
- In-Circuit Serial Programming for PIC16F8X FLASH MCUs ...................................... 3-149
- In-Circuit Serial Programming for PIC16F8XX FLASH MCUs .................................... 3-165

### SECTION 4  APPLICATION NOTES

- In-Circuit Serial Programming™ (ICSP™) of Calibration Parameters Using a PICmicro® Microcontroller ...... 4-1
SECTION 1
INTRODUCTION
INTRODUCTION

In-Circuit Serial Programming™ (ICSP™) Guide

WHAT IS IN-CIRCUIT SERIAL PROGRAMMING (ICSP)?

In-System Programming (ISP) is a technique where a programmable device is programmed after the device is placed in a circuit board.

In-Circuit Serial Programming (ICSP) is an enhanced ISP technique implemented in Microchip’s PICmicro® One-Time-Programmable (OTP) and FLASH RISC microcontrollers (MCU). Use of only two I/O pins to serially input and output data makes ICSP easy to use and less intrusive on the normal operation of the MCU.

Because they can accommodate rapid code changes in a manufacturing line, PICmicro OTP and FLASH MCUs offer tremendous flexibility, reduce development time and manufacturing cycles, and improve time to market.

In-Circuit Serial Programming enhances the flexibility of the PICmicro even further.

This In-Circuit Serial Programming Guide is designed to show you how you can use ICSP to get an edge over your competition. Microchip has helped its customers implement ICSP using PICmicro MCUs since 1992. Contact your local Microchip sales representative today for more information on implementing ICSP in your product.

ICSP can be activated through a simple 5-pin connector and a standard PICmicro programmer supporting serial programming mode such as Microchip’s PRO MATE® II.

No other MCU has a simpler and less intrusive Serial Programming Mode to facilitate your ICSP needs.

WHAT CAN I DO WITH IN-CIRCUIT SERIAL PROGRAMMING?

ICSP is truly an enabling technology that can be used in a variety of ways including:

- **Reduce Cost of Field Upgrades**
  The cost of upgrading a system’s code can be dramatically reduced using ICSP. With very little effort and planning, a PICmicro OTP- or FLASH-based system can be designed to have code updates in the field.

  For PICmicro FLASH devices, the entire code memory can be rewritten with new code. In PICmicro OTP devices, new code segments and parameter tables can be easily added in program memory areas left blank for update purpose. Often, only a portion of the code (such as a key algorithm) requires update.

- **Reduce Time to Market**
  In instances where one product is programmed with different customer codes, generic systems can be built and inventoried ahead of time. Based on actual mix of customer orders, the PICmicro MCU can be programmed using ICSP, then tested and shipped. The lead-time reduction and simplification of finished goods inventory are key benefits.

- **Calibrate Your System During Manufacturing**
  Many systems require calibration in the final stages of manufacturing and testing. Typically, calibration parameters are stored in Serial EEPROM devices. Using PICmicro MCUs, it is possible to save the additional system cost by programming the calibration parameters directly into the program memory.

- **Add Unique ID Code to Your System During Manufacturing**
  Many products require a unique ID number or a serial number. An example application would be a remote keyless entry device. Each transmitter has a unique “binary key” that makes it very easy to program in the access code at the very end of the manufacturing process and prior to final test.

  Serial number, revision code, date code, manufacturer ID and a variety of other useful information can also be added to any product for traceability. Using ICSP, you can eliminate the need for DIP switches or jumpers.
In fact, this capability is so important to many of our customers that Microchip offers a factory programming service called Serialized Quick Turn Programming (SQTPSM), where each PICmicro MCU device is coded with up to 16 bytes of unique code.

- **Calibrate Your System in the Field**

  Calibration need not be done only in the factory. During installation of a system, ICSP can be used to further calibrate the system to actual operating environment.

  In fact, recalibration can be easily done during periodic servicing and maintenance. In OTP parts, newer calibration data can be written to blank memory locations reserved for such use.

- **Customize and Configure Your System in the Field**

  Like calibration, customization need not done in the factory only. In many situations, customizing a product at installation time is very useful. A good example is home or car security systems where ID code, access code and other such information can be burned in after the actual configuration is determined. Additionally, you can save the cost of DIP switches and jumpers, which are traditionally used.

- **Program Dice When Using Chip-On-Board (COB)**

  If you are using COB, Microchip offers a comprehensive die program. You can get dice that are preprogrammed, or you may want to program the die once the circuit board is assembled. Programming and testing in one single step in the manufacturing process is simpler and more cost effective.

**PROGRAMMING TIME CONSIDERATIONS**

Programming time can be significantly different between OTP and FLASH MCUs. OTP (EPROM) bytes typically program with pulses in the order of several hundred microseconds. FLASH, on the other hand, require several milliseconds or more per byte (or word) to program.

Figure 1 and Figure 2 below illustrate the programming time differences between OTP and FLASH MCUs. Figure 1 shows programming time in an ideal programmer or tester, where the only time spent is actually programming the device. This is only important to illustrate the minimum time required to program such devices, where the programmer or the tester is fully optimized.

Figure 2 is a more realistic programming time comparison, where the “overhead” time for programmer or a tester is built in. The programmer often requires 3 to 5 times the “theoretically” minimum programming time.

**FIGURE 1:** PROGRAMMING TIME FOR FLASH AND OTP MCUS (THEORETICAL MINIMUM TIMES)

![Programming Time Graph](image)

**Note 1:** The programming times shown here only include the total programming time for all memory. Typically, a programmer will have quite a bit of overhead over this “theoretical minimum” programming time.

**Note 2:** In the PIC16CXX MCU (used here for comparison) each word is 14 bits wide. For the sake of simplicity, each word is viewed as “two bytes”.
FIGURE 2: PROGRAMMING TIME FOR FLASH AND OTP MCUS
(TYPICAL PROGRAMMING TIMES ON A PROGRAMMER)

Note 1: The programming times shown are actual programming times on vendor supplied programmers.
2: Microchip OTP programming times are based on PRO MATE II programmer.

Ramifications
The programming time differences between FLASH and OTP MCUs are not particular material for prototyping quantities. However, its impact can be significant in large volume production.

MICROCHIP PROVIDES A COMPLETE SOLUTION FOR ICSP

Products
Microchip offers the broadest line of ICSP-capable MCUs:
- PIC12C5XX OTP, 8-pin Family
- PIC12C67X OTP, 8-pin Family
- PIC12CE67X OTP, 8-pin Family
- PIC16C6XX OTP, Mid-Range Family
- PIC17C7XX OTP High-End Family
- PIC18CXXX OTP, High-End Family
- PIC16F62X FLASH, Mid-Range Family
- PIC16F8X FLASH, Mid-Range Family
- PIC6F8XX FLASH, Mid-Range Family

All together, Microchip currently offers over 40 MCUs capable of ICSP.

Development Tools
Microchip offers a comprehensive set of development tools for ICSP that allow system engineers to quickly prototype, make code changes and get designs out the door faster than ever before.

PRO MATE II Production Programmer – a production quality programmer designed to support the Serial Programming Mode in MCUs up to midvolume production. PRO MATE II runs under DOS in a Command Line Mode, Microsoft® Windows® 3.1, Windows® 95/98, and Windows NT®. PRO MATE II is also capable ofSerialized Quick Turn ProgrammingSM (SQTPSM), where each device can be programmed with up to 16 bytes of unique code.

Microchip offers an ICSP kit that can be used with the Universal Microchip Device Programmer, PRO MATE II. Together these two tools allow you to implement ICSP with minimal effort and use the ICSP capability of Microchip’s PICmicro MCUs.

Technical support
Microchip has been delivering ICSP capable MCUs since 1992. Many of our customers are using ICSP capability in full production. Our field and factory application engineers can help you implement ICSP in your product.
SECTION 2
TECHNICAL BRIEFS

HOW TO IMPLEMENT ICSP™ USING PIC12C5XX OTP MCUS ........................................................... 2-1
HOW TO IMPLEMENT ICSP™ USING PIC16CXXX OTP MCUS .......................................................... 2-9
HOW TO IMPLEMENT ICSP™ USING PIC17CXXX OTP MCUS ......................................................... 2-15
HOW TO IMPLEMENT ICSP™ USING PIC16F8X FLASH MCUS ........................................................ 2-21
INTRODUCTION

The technical brief describes how to implement in-circuit serial programming™ (ICSP) using the PIC12C5XX OTP PICmicro® MCU.

ICSP is a simple way to manufacture your board with an unprogrammed PICmicro MCU and program the device just before shipping the product. Programming the PIC12C5XX MCU in-circuit has many advantages for developing and manufacturing your product.

- **Reduces inventory of products with old firmware.** With ICSP, the user can manufacture product without programming the PICmicro MCU. The PICmicro MCU will be programmed just before the product is shipped.

- **ICSP in production.** New software revisions or additional software modules can be programmed during production into the PIC12C5XX MCU.

- **ICSP in the field.** Even after your product has been sold, a service man can update your program with new program modules.

- **One hardware with different software.** ICSP allows the user to have one hardware, whereas the PIC12C5XX MCU can be programmed with different types of software.

- **Last minute programming.** Last minute programming can also facilitate quick turnarounds on custom orders for your products.

**FIGURE 1: TYPICAL APPLICATION CIRCUIT**

![Typical Application Circuit Diagram]

To implement ICSP into an application, the user needs to consider three main components of an ICSP system: Application Circuit, Programmer and Programming Environment.

**Application Circuit**

During the initial design phase of the application circuit, certain considerations have to be taken into account. Figure 1 shows a typical circuit that addresses the details to be considered during design. In order to implement ICSP on your application board you have to put the following issues into consideration:

1. Isolation of the GP3/MCLR/VPP pin from the rest of the circuit.
2. Isolation of pins GP1 and GP0 from the rest of the circuit.
3. Capacitance on each of the VDD, GP3/MCLR/VPP, GP1, and GP0 pins.
4. Interface to the programmer.
5. Minimum and maximum operating voltage for VDD.
Isolation of the GP3/MCLR/VPP Pin from the Rest of the Circuit

PIC12C5XX devices have two ways of configuring the MCLR pin:

- MCLR can be connected either to an external RC circuit or
- MCLR is tied internally to VDD

When GP3/MCLR/VPP pin is connected to an external RC circuit, the pull-up resistor is tied to VDD, and a capacitor is tied to ground. This circuit can affect the operation of ICSP depending on the size of the capacitor.

Another point of consideration with the GP3/MCLR/VPP pin, is that when the PICmicro MCU is programmed, this pin is driven up to 13V and also to ground. Therefore, the application circuit must be isolated from the voltage coming from the programmer.

When MCLR is tied internally to VDD, the user has only to consider that up to 13V are present during programming of the GP3/MCLR/VPP pin. This might affect other components connected to that pin.

For more information about configuring the GP3/MCLR/VPP pin, is that when the PICmicro MCU is programmed, this pin is driven up to 13V and also to ground. Therefore, the application circuit must be isolated from the voltage coming from the programmer.

Isolation of Pins GP1 and GP0 from the Rest of the Circuit

Pins GP1 and GP0 are used by the PICmicro MCU for serial programming. GP1 is the clock line and GP0 is the data line.

GP1 is driven by the programmer. GP0 is a bidirectional pin that is driven by the programmer when programming and driven by the PICmicro MCU when verifying.

These pins must be isolated from the rest of the application circuit so as not to affect the signals during programming. You must take into consideration the output impedance of the programmer when isolating GP1 and GP0 from the rest of the circuit. This isolation circuit must account for GP1 being an input on the PICmicro MCU and for GP0 being bidirectional pin.

For example, PRO MATE® II has an output impedance of 1 kΩ. If the design permits, these pins should not be used by the application. This is not the case with most designs. As a designer, you must consider what type of circuitry is connected to GP1 and GP0 and then make a decision on how to isolate these pins.

Total Capacitance on VDD, GP3/MCLR/VPP, GP1, and GP0

The total capacitance on the programming pins affects the rise rates of these signals as they are driven out of the programmer. Typical circuits use several hundred microfarads of capacitance on VDD, which helps to dampen noise and improve electromagnetic interference. However, this capacitance requires a fairly strong driver in the programmer to meet the rise rate timings for VDD.

Interface to the Programmer

Most programmers are designed to simply program the PICmicro MCU itself and don’t have strong enough drivers to power the application circuit.

One solution is to use a driver board between the programmer and the application circuit. The driver board needs a separate power supply that is capable of driving the VPP, VDD, GP1, and GP0 pins with the correct ramp rates and also should provide enough current to power-up the application circuit.

The cable length between the programmer and the circuit is also an important factor for ICSP. If the cable between the programmer and the circuit is too long, signal reflections may occur. These reflections can momentarily cause up to twice the voltage at the end of the cable, that was sent from the programmer. This voltage can cause a latch-up. In this case, a termination resistor has to be used at the end of the signal line.

Minimum and Maximum Operating Voltage for VDD

The PIC12C5XX programming specification states that the device should be programmed at 5V. Special considerations must be made if your application circuit operates at 3V only. These considerations may include totally isolating the PICmicro MCU during programming. The other point of consideration is that the device must be verified at minimum and maximum operation voltage of the circuit in order to ensure proper programming margin.

For example, a battery driven system may operate from three 1.5V cells giving an operating voltage range of 2.7V to 4.5V. The programmer must program the device at 5V and must verify the program memory contents at both 2.7V and 4.5V to ensure that proper programming margins have been achieved.
THE PROGRAMMER

PIC12C5XX MCUs only use serial programming and, therefore, all programmers supporting these devices will support the ICSP. One issue with the programmer is the drive capability. As discussed before, it must be able to provide the specified rise rates on the ICSP signals and also provide enough current to power the application circuit. It is recommended that you buffer the programming signals.

Another point of consideration for the programmer is what VDD levels are used to verify the memory contents of the PICmicro MCU. For instance, the PRO MATE II verifies program memory at the minimum and maximum VDD levels for the specified device and is therefore considered a production quality programmer. On the other hand, the PICSTART® Plus only verifies at 5V and is for prototyping use only. The PIC12C5XX programming specifications state that the program memory contents should be verified at both the minimum and maximum VDD levels that the application circuit will be operating. This implies that the application circuit must be able to handle the varying VDD voltages.

There are also several third-party programmers that are available. You should select a programmer based on the features it has and how it fits into your programming environment. The Microchip Development Systems Ordering Guide (DS30177) provides detailed information on all our development tools. The Microchip Third Party Guide (DS00104) provides information on all of our third party development tool developers. Please consult these two references when selecting a programmer. Many options exist including serial or parallel PC host connection, stand-alone operation, and single or gang programmers.

PROGRAMMING ENVIRONMENT

The programming environment will affect the type of programmer used, the programmer cable length, and the application circuit interface. Some programmers are well suited for a manual assembly line while others are desirable for an automated assembly line. A gang programmer should be chosen for programming multiple MCUs at one time. The physical distance between the programmer and the application circuit affects the load capacitance on each of the programming signals. This will directly affect the drive strength needed to provide the correct signal rise rates and current. Finally, the application circuit interface to the programmer depends on the size constraints of the application circuit itself and the assembly line. A simple header can be used to interface the application circuit to the programmer. This might be more desirable for a manual assembly line where a technician plugs the programmer cable into the board.

A different method is the uses spring loaded test pins (often referred as pogo-pins). The application circuit has pads on the board for each of the programming signals. Then there is a movable fixture that has pogo pins in the same configuration as the pads on the board. The application circuit is moved into position and the fixture is moved such that the spring loaded test pins come into contact with the board. This method might be more suitable for an automated assembly line.

After taking into consideration the issues with the application circuit, the programmer, and the programming environment, anyone can build a high quality, reliable manufacturing line based on ICSP.

OTHER BENEFITS

ICSP provides several other benefits such as calibration and serialization. If program memory permits, it would be cheaper and more reliable to store calibration constants in program memory instead of using an external serial EEPROM.

Field Programming of PICmicro OTP MCUs

An OTP device is not normally capable of being reprogrammed, but the PICmicro MCU architecture gives you this flexibility provided the size of your firmware is less than half that of the desired device.

This method involves using jump tables for the reset and interrupt vectors. Example 1 shows the location of a main routine and the reset vector for the first time a device with 0.5K-words of program memory is programmed. Example 2 shows the location of a second main routine and its reset vector for the second time the same device is programmed. You will notice that the GOTO Main that was previously at location 0x0002 is replaced with an NOP. An NOP is a program memory location with all the bits programmed as 0s. When the reset vector is executed, it will execute an NOP and then a GOTO Main1 instruction to the new code.
EXAMPLE 1: LOCATION OF THE FIRST MAIN ROUTINE AND ITS INTERRUPT VECTOR

PROGRAM MEMORY

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>MOVWF OSCAL</td>
<td></td>
</tr>
<tr>
<td>0x001</td>
<td>GOTO MAIN1</td>
<td>RESET VECTOR</td>
</tr>
<tr>
<td>0x040</td>
<td>MAIN1</td>
<td>MAIN1 ROUTINE</td>
</tr>
<tr>
<td>0x080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1FF</td>
<td>MOVlw XX</td>
<td>CALIBRATION VALUE</td>
</tr>
</tbody>
</table>

LEGEND: XX = CALIBRATION VALUE
EXAMPLE 2: LOCATION OF THE SECOND MAIN ROUTINE AND ITS INTERRUPT VECTOR
(AFTER SECOND PROGRAMMING)

LEGEND: XX = CALIBRATION VALUE
Since the program memory of the PIC12C5XX devices is organized in 256 x 12 word pages, placement of such information as look-up tables and CALL instructions must be taken into account. For further information, please refer to application note AN581, Implementing Long Calls and application note AN556, Implementing a Table Read.

CONCLUSION

Microchip Technology Inc. is committed to supporting your ICSP needs by providing you with our many years of experience and expertise in developing in-circuit system programming solutions. Anyone can create a reliable in-circuit system programming station by coupling our background with some forethought to the circuit design and programmer selection issues previously mentioned. Your local Microchip representative is available to answer any questions you have about the requirements for ICSP.
APPENDIX A: SAMPLE DRIVER BOARD SCHEMATIC

Note: The driver board design MUST be tested in the user’s application to determine the effects of the application's circuit on the programming signals timing. Changes may be required if the application places a significant load on VDD, VPP, GP0 or GP1.

*see text in technical brief.*

*see text in technical brief.*
INTRODUCTION

In-Circuit Serial Programming™ (ICSP) is a great way to reduce your inventory overhead and time-to-market for your product. By assembling your product with a blank Microchip microcontroller (MCU), you can stock one design. When an order has been placed, these units can be programmed with the latest revision of firmware, tested, and shipped in a very short time. This method also reduces scrapped inventory due to old firmware revisions. This type of manufacturing system can also facilitate quick turnarounds on custom orders for your product.

Most people would think to use ICSP with PICmicro® OTP MCUs only on an assembly line where the device is programmed once. However, there is a method by which an OTP device can be programmed several times depending on the size of the firmware. This method, explained later, provides a way to field upgrade your firmware in a way similar to EEPROM- or Flash-based devices.

HOW DOES ICSP WORK?

Now that ICSP appeals to you, what steps do you take to implement it in your application? There are three main components of an ICSP system: Application Circuit, Programmer and Programming Environment.

Application Circuit

The application circuit must be designed to allow all the programming signals to be directly connected to the PICmicro MCU. Figure 1 shows a typical circuit that is a starting point for when designing with ICSP. The application must compensate for the following issues:

1. Isolation of the MCLR/Vpp pin from the rest of the circuit.
2. Isolation of pins RB6 and RB7 from the rest of the circuit.
3. Capacitance on each of the Vdd, MCLR/Vpp, RB6, and RB7 pins.
4. Minimum and maximum operating voltage for Vdd.
5. PICmicro Oscillator.
6. Interface to the programmer.

The MCLR/Vpp pin is normally connected to an RC circuit. The pull-up resistor is tied to Vdd and a capacitor is tied to ground. This circuit can affect the operation of ICSP depending on the size of the capacitor. It is, therefore, recommended that the circuit in Figure 1 be used when an RC is connected to MCLR/Vpp. The diode should be a Schottky-type device. Another issue with MCLR/Vpp is that when the PICmicro MCU device is programmed, this pin is driven to approximately 13V and also to ground. Therefore, the application circuit must be isolated from this voltage provided by the programmer.
Pins RB6 and RB7 are used by the PICmicro MCU for serial programming. RB6 is the clock line and RB7 is the data line. RB6 is driven by the programmer. RB7 is a bidirectional pin that is driven by the programmer when programming, and driven by the PICmicro MCU when verifying. These pins must be isolated from the rest of the application circuit so as not to affect the signals during programming. You must take into consideration the output impedance of the programmer when isolating RB6 and RB7 from the rest of the circuit. This isolation circuit must account for RB6 being an input on the PICMicro MCU, and for RB7 being bidirectional (can be driven by both the PICmicro MCU and the programmer). For instance, PRO MATE® II has an output impedance of 1kΩ. If the design permits, these pins should not be used by the application. This is not the case with most applications so it is recommended that the designer evaluate whether these signals need to be buffered. As a designer, you must consider what type of circuitry is connected to RB6 and RB7 and then make a decision on how to isolate these pins. Figure 1 does not show any circuitry to isolate RB6 and RB7 on the application circuit because this is very application dependent.

The total capacitance on the programming pins affects the rise rates of these signals as they are driven out of the programmer. Typical circuits use several hundred microfarads of capacitance on VDD which helps to dampen noise and ripple. However, this capacitance requires a fairly strong driver in the programmer to meet the rise rate timings for VDD. Most programmers are designed to simply program the PICmicro MCU itself and don't have strong enough drivers to power the application circuit. One solution is to use a driver board between the programmer and the application circuit. The driver board requires a separate power supply that is capable of driving the VPP and VDD pins with the correct rise rates and should also provide enough current to power the application circuit. RB6 and RB7 are not buffered on this schematic but may require buffering depending upon the application. A sample driver board schematic is shown in Appendix A.

Note: The driver board design MUST be tested in the user's application to determine the effects of the application circuit on the programming signals timing. Changes may be required if the application places a significant load on VDD, VPP, RB6 OR RB7.

The Microchip programming specification states that the device should be programmed at 5V. Special considerations must be made if your application circuit operates at 3V only. These considerations may include totally isolating the PICmicro MCU during programming. The other issue is that the device must be verified at the minimum and maximum voltages at which the application circuit will be operating. For instance, a battery operated system may operate from three 1.5V cells giving an operating voltage range of 2.7V to 4.5V. The programmer must program the device at 5V and must verify the program memory contents at both 2.7V and 4.5V to ensure that proper programming margins have been achieved. This ensures the PICmicro MCU option over the voltage range of the system.

This final issue deals with the oscillator circuit on the application board. The voltage on MCLR/VPP must rise to the specified program mode entry voltage before the device executes any code. The crystal modes available on the PICmicro MCU are not affected by this issue because the Oscillator Start-up Timer waits for 1024 oscillations before any code is executed. However, RC oscillators do not require any startup time and, therefore, the Oscillator Startup Timer is not used. The programmer must drive MCLR/VPP to the program mode entry voltage before the RC oscillator toggles four times. If the RC oscillator toggles four or more times, the program counter will be incremented to some value X. Now when the device enters programming mode, the program counter will not be zero and the programmer will start programming your code at an offset of X. There are several alternatives that can compensate for a slow rise rate on MCLR/VPP. The first method would be to not populate the R, program the device, and then insert the R. The other method would be to have the programming interface drive the OSC1 pin of the PICmicro MCU to ground while programming. This will prevent any oscillations from occurring during programming.

Now all that is left is how to connect the application circuit to the programmer. This depends a lot on the programming environment and will be discussed in that section.

Programmer

The second consideration is the programmer. PIC16CXXX MCUs only use serial programming and therefore all programmers supporting these devices will support ICSP. One issue with the programmer is the drive capability. As discussed before, it must be able to provide the specified rise rates on the ICSP signals and also provide enough current to power the application circuit. Appendix A shows an example driver board. This driver schematic does not show any buffer circuitry for RB6 and RB7. It is recommended that an evaluation be performed to determine if buffering is required. Another issue with the programmer is what VDD levels are used to verify the memory contents of the PICmicro MCU. For instance, the PRO MATE II verifies program memory at the minimum and maximum VDD levels for the specified device and is therefore considered a production quality programmer. On the other hand, the PICSTART® Plus only verifies at 5V and is for prototyping use only. The Microchip programming specifications state that the program memory contents should be verified at both the minimum and maximum VDD levels that the application circuit will be operating. This implies that the application circuit must be able to handle the varying VDD voltages.
There are also several third party programmers that are available. You should select a programmer based on the features it has and how it fits into your programming environment. The *Microchip Development Systems Ordering Guide* (DS30177) provides detailed information on all our development tools. The *Microchip Third Party Guide* (DS00104) provides information on all of our third party tool developers. Please consult these two references when selecting a programmer. Many options exist including serial or parallel PC host connection, stand-alone operation, and single or gang programmers. Some of the third party developers include Advanced Transdata Corporation, BP Microsystems, Data I/O, Emulation Technology and Logical Devices.

**Programming Environment**

The programming environment will affect the type of programmer used, the programmer cable length, and the application circuit interface. Some programmers are well suited for a manual assembly line while others are desirable for an automated assembly line. You may want to choose a gang programmer to program multiple systems at a time.

The physical distance between the programmer and the application circuit affects the load capacitance on each of the programming signals. This will directly affect the drive strength needed to provide the correct signal rise rates and current. This programming cable must also be as short as possible and properly terminated and shielded, or the programming signals may be corrupted by ringing or noise.

Finally, the application circuit interface to the programmer depends on the size constraints of the application circuit itself and the assembly line. A simple header can be used to interface the application circuit to the programmer. This might be more desirable for a manual assembly line where a technician plugs the programmer cable into the board. A different method is the use of spring loaded test pins (commonly referred to as pogo pins). The application circuit has pads on the board for each of the programming signals. Then there is a fixture that has pogo pins in the same configuration as the pads on the board. The application circuit or fixture is moved into position such that the pogo pins come into contact with the board. This method might be more suitable for an automated assembly line.

After taking into consideration the issues with the application circuit, the programmer, and the programming environment, anyone can build a high quality, reliable manufacturing line based on ICSP.

**Other Benefits**

ICSP provides other benefits, such as calibration and serialization. If program memory permits, it would be cheaper and more reliable to store calibration constants in program memory instead of using an external serial EEPROM. For example, your system has a thermistor which can vary from one system to another. Storing some calibration information in a table format allows the microcontroller to compensate in software for external component tolerances. System cost can be reduced without affecting the required performance of the system by using software calibration techniques. But how does this relate to ICSP? The PICmicro MCU has already been programmed with firmware that performs a calibration cycle. The calibration data is transferred to a calibration fixture. When all calibration data has been transferred, the fixture places the PICmicro MCU in programming mode and programs the PICmicro MCU with the calibration data. Application note AN656, *In-Circuit Serial Programming of Calibration Parameters Using a PICmicro Microcontroller*, shows exactly how to implement this type of calibration data programming.

The other benefit of ICSP is serialization. Each individual system can be programmed with a unique or random serial number. One such application of a unique serial number would be for security systems. A typical system might use DIP switches to set the serial number. Instead, this number can be burned into program memory, thus reducing the overall system cost and lowering the risk of tampering.

**Field Programming of PICmicro OTP MCUs**

An OTP device is not normally capable of being reprogrammed, but the PICmicro MCU architecture gives you this flexibility provided the size of your firmware is at least half that of the desired device and the device is not code protected. If your target device does not have enough program memory, Microchip provides a wide spectrum of devices from 0.5K to 8K program memory with the same set of peripheral features that will help meet the criteria.

The PIC16CXXX microcontrollers have two vectors, reset and interrupt, at locations 0x0000 and 0x0004. When the PICmicro MCU encounters a reset or interrupt condition, the code located at one of these two locations in program memory is executed. The first listing of Example 1 shows the code that is first programmed into the PICmicro MCU. The second listing of Example 1 shows the code that is programmed into the PICmicro MCU for the second time.
### EXAMPLE 1: PROGRAMMING CYCLE LISTING FILES

<table>
<thead>
<tr>
<th>First Program Cycle</th>
<th>Second Program Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prog Opcode</strong></td>
<td><strong>Prog Opcode</strong></td>
</tr>
<tr>
<td><strong>Assembly</strong></td>
<td><strong>Assembly</strong></td>
</tr>
<tr>
<td><strong>Mem Instruction</strong></td>
<td><strong>Mem Instruction</strong></td>
</tr>
<tr>
<td>0000 2808 goto Main ;Main loop</td>
<td>0000 0000 nop</td>
</tr>
<tr>
<td>0001 3FFF &lt;blank&gt; ;at 0x0006</td>
<td>0001 2860 goto Main ;Main now</td>
</tr>
<tr>
<td>0002 3FFF &lt;blank&gt;</td>
<td>0002 3FFF &lt;blank&gt; ;at 0x0060</td>
</tr>
<tr>
<td>0003 3FFF &lt;blank&gt;</td>
<td>0003 3FFF &lt;blank&gt;</td>
</tr>
<tr>
<td>0004 2848 goto ISR ;ISR at</td>
<td>0004 0000 nop</td>
</tr>
<tr>
<td>0005 3FFF &lt;blank&gt; ;0x0048</td>
<td>0005 28A8 goto ISR ;ISR now at</td>
</tr>
<tr>
<td>0006 3FFF &lt;blank&gt;</td>
<td>0006 3FFF &lt;blank&gt; ;0x00A8</td>
</tr>
<tr>
<td>0007 3FFF &lt;blank&gt;</td>
<td>0007 3FFF &lt;blank&gt;</td>
</tr>
<tr>
<td>0008 1683 bsf STATUS,RP0</td>
<td>0008 1683 bsf STATUS,RP0</td>
</tr>
<tr>
<td>0009 3007 movlw 0x07</td>
<td>0009 3007 movlw 0x07</td>
</tr>
<tr>
<td>000A 009F movwf ADCON1</td>
<td>000A 009F movwf ADCON1</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0048 1C0C btfss PIR1,RBIF</td>
<td>0048 1C0C btfss PIR1,RBIF</td>
</tr>
<tr>
<td>0049 284E goto EndISR</td>
<td>0049 284E goto EndISR</td>
</tr>
<tr>
<td>004A 1806 btfs PORTB,0</td>
<td>004A 1806 btfs PORTB,0</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>0060 3FFF &lt;blank&gt;</td>
<td>0060 1683 bsf STATUS,RP0</td>
</tr>
<tr>
<td>0061 3FFF &lt;blank&gt;</td>
<td>0061 3005 movlw 0x05</td>
</tr>
<tr>
<td>0062 3FFF &lt;blank&gt;</td>
<td>0062 009F movwf ADCON1</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>00A8 3FFF &lt;blank&gt;</td>
<td>00A8 1C0C btfss PIR1,RBIF</td>
</tr>
<tr>
<td>00A9 3FFF &lt;blank&gt;</td>
<td>00A9 28A8 goto EndISR</td>
</tr>
<tr>
<td>00AA 3FFF &lt;blank&gt;</td>
<td>00AA 1806 btfs PORTB,0</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
The example shows that to program the PICmicro MCU a second time the memory location 0x0000, originally \texttt{goto Main (0x2808)}, is reprogrammed to all 0’s which happens to be a \texttt{nop} instruction. This location cannot be reprogrammed to the new opcode (0x2860) because the bits that are 0’s cannot be reprogrammed to 1’s, only bits that are 1’s can be reprogrammed to 0’s. The next memory location 0x0001 was originally blank (all 1’s) and now becomes a \texttt{goto Main (0x2860)}. When a reset condition occurs, the PICmicro MCU executes the instruction at location 0x0000 which is the \texttt{nop}, a completely benign instruction, and then executes the \texttt{goto Main} to start the execution of code. The example also shows that all program memory locations after 0x005A are blank in the original program so that the second time the PICmicro MCU is programmed, the revised code can be programmed at these locations. The same descriptions can be given for the interrupt vector at location 0x0004.

This method changes slightly for PICmicro MCUs with >2K words of program memory. Each of the \texttt{goto Main} and \texttt{goto ISR} instructions are replaced by the following code segments due to paging on devices with >2K words of program memory.

\begin{verbatim}
movlw <page>  movlw <page>
movwf PCLATH  movwf PCLATH
\texttt{goto Main}  \texttt{goto ISR}
\end{verbatim}

Now your one time programmable PICmicro MCU is exhibiting more EEPROM- or Flash-like qualities.

\section*{CONCLUSION}

Microchip Technology Inc. is committed to supporting your ICSP needs by providing you with our many years of experience and expertise in developing ICSP solutions. Anyone can create a reliable ICSP programming station by coupling our background with some forethought to the circuit design and programmer selection issues previously mentioned. Your local Microchip representative is available to answer any questions you have about the requirements for ICSP.
APPENDIX A: SAMPLE DRIVER BOARD SCHEMATIC

Note:
The driver board design MUST be tested in the user's application to determine the effects of the application circuit on the programming signals timing. Changes may be required if the application places a significant load on Vdd, Vpp, RB6 or RB7.

*see text in technical brief.*
INTRODUCTION

PIC17CXXX microcontroller (MCU) devices can be serially programmed using an RS-232 or equivalent serial interface. As shown in Figure 2, using just three pins, the PIC17CXXX can be connected to an external interface and programmed. In-Circuit Serial Programming (ICSP™) allows for a greater flexibility in an application as well as a faster time to market for the user's product.

This technical brief will demonstrate the practical aspects associated with ICSP using the PIC17CXXX. It will also demonstrate some key capabilities of OTP devices when used in conjunction with ICSP.

Implementation

The PIC17CXXX devices have special instructions, which enables the user to program and read the PIC17CXXX's program memory. The instructions are TABLWT and TLWT which implement the program memory write operation and TABLRD and TLRD which perform the program memory read operation. For more details, please check the In-Circuit Serial Programming for PIC17CXXX OTP Microcontrollers Specification (DS30273), PIC17C4X data sheet (DS30412) and PIC17C75X data sheet (DS30264).

When doing ICSP, the PIC17CXXX runs a boot code, which configures the USART port and receives data serially through the RX line. This data is then programmed at the address specified in the serial data string. A high voltage (about 13V) is required for the EPROM cell to get programmed, and this is usually supplied by the programming header as shown in Figure 2 and Figure 3. The PIC17CXXX's boot code enables and disables the high voltage line using a dedicated I/O line.
ICSP Boot Code

The boot code is normally programmed, into the PIC17CXXX device using a PRO MATE® or PICSTART® Plus or any third party programmer. As depicted in the flowchart in Figure 5, on power-up, or a reset, the program execution always vectors to the boot code. The boot code is normally located at the bottom of the program memory space e.g. 0x700 for a PIC17C42A (Figure 4).

Several methods could be used to reset the PIC17CXXX when the ICSP header is connected to the system board. The simplest method, as shown in Figure 3, is to derive the system 5V, from the 13V supplied by the ICSP header. It is quite common in manufacturing lines, to have system boards programmed with only the boot code ready and available for testing, calibration or final programming. The ICSP header would thus supply the 13V to the system and this 13V would then be stepped down to supply the 5V required to power the system. Please note that the 13V supply should have enough drive capability to supply power to the system as well as maintain the programming voltage of 13V.

The first action of the boot code (as shown in flowchart Figure 5) is to configure the USART to a known baud rate and transmit a request sequence to the ICSP host system. The host immediately responds with an acknowledgment of this request. The boot code then gets ready to receive ICSP data. The host starts sending the data and address byte sequences to the PIC17CXXX. On receiving the address and data information, the 16-bit address is loaded into the TBLPTR registers and the 16-bit data is loaded into the TABLAT registers. The RA2 pin is driven low to enable 13V at MCLR. The PIC17CXXX device then executes a table write instruction. This instruction in turn causes a long write operation, which disables further code execution. Code execution is resumed when an internal interrupt occurs. This delay ensures that the programming pulse width of 1 ms (max.) is met. Once a location is written, RA2 is driven high to disable further writes and a verify operation is done using the Table read instruction. If the result is good, an acknowledge is sent to the host. This process is repeated till all desired locations are programmed.

In normal operation, when the ICSP header is not connected, the boot code would still execute and the PIC17CXXX would send out a request to the host. However it would not get a response from the host, so it would abort the boot code and start normal code execution.
FIGURE 5: FLOWCHART FOR ICSP BOOT CODE

1. Start
2. Goto Boot Code
3. Configure USART and send request
4. Received Host’s ACK?
   - Yes: Prepare to receive ICSP data
   - No: Time-out complete?
      - Yes: Start Code Execution
      - No: Received Address and Data info?
         - Yes: Do Table Write operation
         - No: Interrupt?
            - Yes: Read Program Location
            - No: Last Data/Address sequence?
               - Yes: Signal Programming Error
               - No: Program location verified correctly?
                  - Yes: END
                  - No: Repeat
USING THE ICSP FEATURE ON PIC17CXXX OTP DEVICES

The ICSP feature is a very powerful tool when used in conjunction with OTP devices.

Saving Calibration Information Using ICSP

One key use of ICSP is to store calibration constants or parameters in program memory. It is quite common to interface a PIC17CXXX device to a sensor. Accurate, pre-calibrated sensors can be used, but they are more expensive and have long lead times. Uncalibrated sensors on the other hand are inexpensive and readily available. The only caveat is that these sensors have to be calibrated in the application. Once the calibration constants have been determined, they would be unique to a given system, so they have to be saved in program memory. These calibration parameters/constants can then be retrieved later during program execution and used to improve the accuracy of low cost un-calibrated sensors. ICSP thus offers a cost reduction path for the end user in the application.

Saving Field Calibration Information Using ICSP

Sensors typically tend to drift and lose calibration over time and usage. One expensive solution would be to replace the sensor with a new one. A more cost effective solution however, is to re-calibrated the system and save the new calibration parameter/constants into the PIC17CXXX devices using ICSP. The user program however has to take into account certain issues:

1. Un-programmed or blank locations have to be reserved at each calibration constant location in order to save new calibration parameters/constants.
2. The old calibration parameters/constants are all programmed to 0, so the user program will have to be “intelligent” and differentiate between blank (0xFFFF), zero (0x0000), and programmed locations.

Figure 6 shows how this can be achieved.

Programming Unique Serial Numbers Using ICSP

There are applications where each system needs to have a unique and sometimes random serial number. Example: security devices. One common solution is to have a set of DIP switches which are then set to a unique value during final test. A more cost effective solution however would be to program unique serial numbers into the device using ICSP. The user application can thus eliminate the need for DIP switches and subsequently reduce the cost of the system.

FIGURE 6: FIELD CALIBRATION USING ICSP

![Diagram showing field calibration using ICSP]
Code Updates in the Field Using ICSP

With fast time to market it is not uncommon to see application programs which need to be updated or corrected for either enhancements or minor errors/bugs. If ROM parts were used, updates would be impossible and the product would either become outdated or recalled from the field. A more cost effective solution is to use OTP devices with ICSP and program them in the field with the new updates. Figure 7 shows an example where the user has allowed for one field update to his program.

Here are some of the issues which need to be addressed:

1. The user has to reserve sufficient blank memory to fit his updated code.
2. At least one blank location needs to be saved at the reset vector as well as for all the interrupts.
3. Program all the old "goto" locations (located at the reset vector and the interrupts vectors) to 0 so that these instructions execute as NOPs.
4. Program new "goto" locations (at the reset vector and the interrupt vectors) just below the old "goto" locations.
5. Finally, program the new updated code in the blank memory space.

CONCLUSION

ICSP is a very powerful feature available on the PIC17CXXX devices. It offers tremendous design flexibility to the end user in terms of saving calibration constants and updating code in final production as well as in the field, thus helping the user design a low-cost and fast time-to-market product.

FIGURE 7: CODE UPDATES USING ICSP
INTRODUCTION

In-Circuit Serial Programming™ (ICSP) with PICmicro® FLASH microcontrollers (MCU) is not only a great way to reduce your inventory overhead and time-to-market for your product, but also to easily provide field upgrades of firmware. By assembling your product with a Microchip FLASH-based MCU, you can stock the shelf with one system. When an order has been placed, these units can be programmed with the latest revision of firmware, tested, and shipped in a very short time. This type of manufacturing system can also facilitate quick turnarounds on custom orders for your product. You don’t have to worry about scrapped inventory because of the FLASH-based program memory. This gives you the advantage of upgrading the firmware at any time to fix those “features” that pop up from time to time.

HOW DOES ICSP WORK?

Now that ICSP appeals to you, what steps do you take to implement it in your application? There are three main components of an ICSP system.

These are the: Application Circuit, Programmer and Programming Environment.

Application Circuit

The application circuit must be designed to allow all the programming signals to be directly connected to the PICmicro MCUs. Figure 1 shows a typical circuit that is a starting point for when designing with ICSP. The application must compensate for the following issues:

1. Isolation of the MCLR/VPP pin from the rest of the circuit.
2. Isolation of pins RB6 and RB7 from the rest of the circuit.
3. Capacitance on each of the VDD, MCLR/VPP, RB6, and RB7 pins.
4. Minimum and maximum operating voltage for VDD.
5. PICmicro Oscillator.
6. Interface to the programmer.

The MCLR/VPP pin is normally connected to an RC circuit. The pull-up resistor is tied to VDD and a capacitor is tied to ground. This circuit can affect the operation of ICSP depending on the size of the capacitor. It is, therefore, recommended that the circuit in Figure 1 be used when an RC is connected to MCLR/VPP. The diode should be a Schottky-type device. Another issue with MCLR/VPP is that when the PICmicro MCU device is programmed, this pin is driven to approximately 13V and also to ground. Therefore, the application circuit must be isolated from this voltage provided by the programmer.
Pins RB6 and RB7 are used by the PICmicro MCU for serial programming. RB6 is the clock line and RB7 is the data line. RB6 is driven by the programmer. RB7 is a bidirectional pin that is driven by the programmer when programming, and driven by the PICmicro MCU when verifying. These pins must be isolated from the rest of the application circuit so as not to affect the signals during programming. You must take into consideration the output impedance of the programmer when isolating RB6 and RB7 from the rest of the circuit. This isolation circuit must account for RB6 being an input on the PICmicro MCU and for RB7 being bidirectional (can be driven by both the PICmicro MCU and the programmer). For instance, PRO MATE® II has an output impedance of 1kΩ. If the design permits, these pins should not be used by the application. This is not the case with most applications so it is recommended that the designer evaluate whether these signals need to be buffered. As a designer, you must consider what type of circuitry is connected to RB6 and RB7 and then make a decision on how to isolate these pins. Figure 1 does not show any circuitry to isolate RB6 and RB7 on the application circuit because this is very application dependent.

The total capacitance on the programming pins affects the rise rates of these signals as they are driven out of the programmer. Typical circuits use several hundred microfarads of capacitance on VDD which helps to dampen noise and ripple. However, this capacitance requires a fairly strong driver in the programmer to meet the rise rate timings for VDD. Most programmers are designed to simply program the PICmicro MCU itself and don’t have strong enough drivers to power the application circuit. One solution is to use a driver board between the programmer and the application circuit. The driver board requires a separate power supply that is capable of driving the VPP and VDD pins with the correct rise rates and should also provide enough current to power the application circuit. RB6 and RB7 are not buffered on this schematic but may require buffering depending upon the application. A sample driver board schematic is shown in Appendix A.

Note: The driver board design MUST be tested in the user’s application to determine the effects of the application circuit on the programming signals timing. Changes may be required if the application places a significant load on Vdd, VPP, RB6 or RB7.

The Microchip programming specification states that the device should be programmed at 5V. Special considerations must be made if your application circuit operates at 3V only. These considerations may include totally isolating the PICmicro MCU during programming. The other issue is that the device must be verified at the minimum and maximum voltages at which the application circuit will be operating. For instance, a battery operated system may operate from three 1.5V cells giving an operating voltage range of 2.7V to 4.5V. The programmer must program the device at 5V and must verify the program memory contents at both 2.7V and 4.5V to ensure that proper programming margins have been achieved. This ensures the PICmicro MCU option over the voltage range of the system.

This final issue deals with the oscillator circuit on the application board. The voltage on MCLR/VPP must rise to the specified program mode entry voltage before the device executes any code. The crystal modes available on the PICmicro MCU are not affected by this issue because the Oscillator Start-up Timer waits for 1024 oscillations before any code is executed. However, RC oscillators do not require any startup time and, therefore, the Oscillator Startup Timer is not used. The programmer must drive MCLR/VPP to the program mode entry voltage before the RC oscillator toggles four times. If the RC oscillator toggles four or more times, the program counter will be incremented to some value X. Now when the device enters programming mode, the program counter will not be zero and the programmer will start programming your code at an offset of X. There are several alternatives that can compensate for a slow rise rate on MCLR/VPP. The first method would be to not populate the R, program the device, and then insert the R. The other method would be to have the programming interface drive the OSC1 pin of the PICmicro MCU to ground while programming. This will prevent any oscillations from occurring during programming.

Now all that is left is how to connect the application circuit to the programmer. This depends a lot on the programming environment and will be discussed in that section.

Programmer

The second consideration is the programmer. PIC16F8X MCUs only use serial programming and therefore all programmers supporting these devices will support ICSP. One issue with the programmer is the drive capability. As discussed before, it must be able to provide the specified rise rates on the ICSP signals and also provide enough current to power the application circuit. Appendix A shows an example driver board. This driver schematic does not show any buffer circuitry for RB6 and RB7. It is recommended that an evaluation be performed to determine if buffering is required. Another issue with the programmer is what VDD levels are used to verify the memory contents of the PICmicro MCU. For instance, the PRO MATE II verifies program memory at the minimum and maximum VDD levels for the specified device and is therefore considered a production quality programmer. On the other hand, the PICSTART® Plus only verifies at 5V and is for prototyping use only. The Microchip programming specifications state that the program memory contents should be verified at both the minimum and maximum VDD levels that the application circuit will be operating. This implies that the application circuit must be able to handle the varying VDD voltages.
There are also several third party programmers that are available. You should select a programmer based on the features it has and how it fits into your programming environment. The Microchip Development Systems Ordering Guide (DS30177) provides detailed information on all our development tools. The Microchip Third Party Guide (DS00104) provides information on all of our third party tool developers. Please consult these two references when selecting a programmer. Many options exist including serial or parallel PC host connection, stand-alone operation, and single or gang programmers. Some of the third party developers include Advanced Transdata Corporation, BP Microsystems, Data I/O, Emulation Technology and Logical Devices.

Programming Environment

The programming environment will affect the type of programmer used, the programmer cable length, and the application circuit interface. Some programmers are well suited for a manual assembly line while others are desirable for an automated assembly line. You may want to choose a gang programmer to program multiple systems at a time.

The physical distance between the programmer and the application circuit affects the load capacitance on each of the programming signals. This will directly affect the drive strength needed to provide the correct signal rise rates and current. This programming cable must also be as short as possible and properly terminated and shielded or the programming signals may be corrupted by ringing or noise.

Finally, the application circuit interface to the programmer depends on the size constraints of the application circuit itself and the assembly line. A simple header can be used to interface the application circuit to the programmer. This might be more desirable for a manual assembly line where a technician plugs the programmer cable into the board. A different method is the use of spring loaded test pins (commonly referred to as pogo pins). The application circuit has pads on the board for each of the programming signals. Then there is a fixture that has pogo pins in the same configuration as the pads on the board. The application circuit or fixture is moved into position such that the pogo pins come into contact with the board. This method might be more suitable for an automated assembly line.

After taking into consideration the issues with the application circuit, the programmer, and the programming environment, anyone can build a high quality, reliable manufacturing line based on ICSP.

Other Benefits

ICSP provides other benefits, such as calibration and serialization. If program memory permits, it would be cheaper and more reliable to store calibration constants in program memory instead of using an external serial EEPROM. For example, your system has a thermometer which can vary from one system to another. Storing some calibration information in a table format allows the microcontroller to compensate in software for external component tolerances. System cost can be reduced without affecting the required performance of the system by using software calibration techniques. But how does this relate to ICSP? The PICmicro MCU has already been programmed with firmware that performs a calibration cycle. The calibration data is transferred to a calibration fixture. When all calibration data has been transferred, the fixture places the PICmicro MCU in programming mode and programs the PICmicro MCU with the calibration data. Application note AN656, In-Circuit Serial Programming of Calibration Parameters Using a PICmicro Microcontroller, shows exactly how to implement this type of calibration data programming.

The other benefit of ICSP is serialization. Each individual system can be programmed with a unique or random serial number. One such application of a unique serial number would be for security systems. A typical system might use DIP switches to set the serial number. Instead, this number can be burned into program memory thus reducing the overall system cost and lowering the risk of tampering.

Field Programming of FLASH PICmicro MCUs

With the ISP interface circuitry already in place, these FLASH-based PICmicro MCUs can be easily reprogrammed in the field. These FLASH devices allow you to reprogram them even if they are code protected. A portable ISP programming station might consist of a laptop computer and programmer. The technician plugs the ISP interface cable into the application circuit and downloads the new firmware into the PICmicro MCU. The next thing you know the system is up and running without those annoying “bugs”. Another instance would be that you want to add an additional feature to your system. All of your current inventory can be converted to the new firmware and field upgrades can be performed to bring your installed base of systems up to the latest revision of firmware.

CONCLUSION

Microchip Technology Inc. is committed to supporting your ICSP needs by providing you with our many years of experience and expertise in developing ICSP solutions. Anyone can create a reliable ICSP programming station by coupling our background with some forethought to the circuit design and programmer selection issues previously mentioned. Your local Microchip representative is available to answer any questions you have about the requirements for ICSP.
APPENDIX A:  SAMPLE DRIVER BOARD SCHEMATIC

Note:  The driver board design MUST be tested in the user's application to determine the effects of the application circuit on the programming signals timing. Changes may be required if the application places a significant load on Vdd, Vpp, RB6 or RB7.

*see text in technical brief.*see text in technical brief.
SECTION 3
PROGRAMMING SPECIFICATIONS

IN-CIRCUIT SERIAL PROGRAMMING FOR PIC12C5XX OTP MCUs .................................................. 3-1
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC12C67X AND PIC12CE67X OTP MCUs ......................... 3-15
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC14000 OTP MCUs .................................................. 3-27
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC16C55X OTP MCUs .................................................. 3-39
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC16C6XX/7XX/9XX OTP MCUs .................................. 3-51
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC17C7XX OTP MCUs .................................................. 3-71
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC18CXXX OTP MCUs .................................................. 3-97
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC16F62X FLASH MCUs ............................................. 3-135
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC16F8X FLASH MCUs .................................................. 3-149
IN-CIRCUIT SERIAL PROGRAMMING FOR PIC16F8XX FLASH MCUs ............................................. 3-165
This document includes the programming specifications for the following devices:

- PIC12C508
- PIC12C508A
- PIC12CE518
- PIC12C509
- PIC12C509A
- PIC12CE519

1.0 PROGRAMMING THE PIC12C5XX

The PIC12C5XX can be programmed using a serial method. Due to this serial programming, the PIC12C5XX can be programmed while in the user’s system increasing design flexibility. This programming specification applies to PIC12C5XX devices in all packages.

1.1 Hardware Requirements

The PIC12C5XX requires two programmable power supplies, one for VDD (2.0V to 6.5V recommended) and one for VPP (12V to 14V). Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC12C5XX allows programming of user program memory, special locations used for ID, and the configuration word for the PIC12C5XX.
2.0 PROGRAM MODE ENTRY

The program/verify test mode is entered by holding pins DB0 and DB1 low while raising MCLR pin from VIL to VIHH. Once in this test mode the user program memory and the test program memory can be accessed and programmed in a serial fashion. The first selected memory location is the fuses. GP0 and GP1 are Schmitt trigger inputs in this mode.

Incrementing the PC once (using the increment address command) selects location 0x000 of the regular program memory. Afterwards all other memory locations from 0x001-01FF (PIC12C508/CE518), 0x001-03FF (PIC12C509/CE519) can be addressed by incrementing the PC.

If the program counter has reached the last user program location and is incremented again, the on-chip special EPROM area will be addressed. (See Figure 2-2 to determine where the special EPROM area is located for the various PIC12C5XX devices).

2.1 Programming Method

The programming technique is described in the following section. It is designed to guarantee good programming margins. It does, however, require a variable power supply for VCC.

2.1.1 PROGRAMMING METHOD DETAILS

Essentially, this technique includes the following steps:

1. Perform blank check at VDD = VDDmin. Report failure. The device may not be properly erased.
2. Program location with pulses and verify after each pulse at VDD = VDDP:
   where VDDP = VDD range required during programming (4.5V - 5.5V).
   a) Programming condition:
      VPP = 13.0V to 13.25V
      VDD = VDDP = 4.5V to 5.5V
      VPP must be ≥ VDD + 7.25V to keep "programming mode" active.
   b) Verify condition:
      VDD = VDDP
      VPP ≥ VDD + 7.5V but not to exceed 13.25V
      If location fails to program after "N" pulses, (suggested maximum program pulses of 8) then report error as a programming failure.

   Note: Device must be verified at minimum and maximum specified operating voltages as specified in the data sheet.

3. Once location passes "Step 2", apply 11X over programming, i.e., apply 11 times the number of pulses that were required to program the location. This will guarantee a solid programming margin. The over programming should be made "software programmable" for easy updates.
4. Program all locations.
5. Verify all locations (using speed verify mode) at VDD = VDDmin
6. Verify all locations at VDD = VDDmax
   VDDmin is the minimum operating voltage spec. for the part. VDDmax is the maximum operating voltage spec. for the part.

2.1.2 SYSTEM REQUIREMENTS

Clearly, to implement this technique, the most stringent requirements will be that of the power supplies:

VPP: VPP can be a fixed 13.0V to 13.25V supply. It must not exceed 14.0V to avoid damage to the pin and should be current limited to approximately 100mA.
VDD: 2.0V to 6.5V with 0.25V granularity. Since this method calls for verification at different VDD values, a programmable VDD power supply is needed.

Current Requirement: 40mA maximum

Microchip may release devices in the future with different VDD ranges which make it necessary to have a programmable VDD.

It is important to verify an EPROM at the voltages specified in this method to remain consistent with Microchip's test screening. For example, a PIC12C5XX specified for 4.5V to 5.5V should be tested for proper programming from 4.5V to 5.5V.

2.1.3 SOFTWARE REQUIREMENTS

Certain parameters should be programmable (and therefore easily modified) for easy upgrade.

a) Pulse width
b) Maximum number of pulses, present limit 8.
   c) Number of over-programming pulses: should be = (A • N) + B, where N = number of pulses required in regular programming. In our current algorithm A = 11, B = 0.

2.2 Programming Pulse Width

Program Memory Cells: When programming one word of EPROM, a programming pulse width (TPW) of 100µs is recommended.

The maximum number of programming attempts should be limited to 8 per word.

After the first successful verify, the same location should be over-programmed with 11X over-programming.

Configuration Word: The configuration word for oscillator selection, WDT (watchdog timer) disable and code protection, and MCLR enable, requires a programming pulse width (TPWF) of 10ms. A series of 100µs pulses is preferred over a single 10ms pulse.
FIGURE 2-1: PROGRAMMING METHOD FLOWCHART

Start

Blank Check @ Vdd = Vddmin

Pass? Yes

No

Report Possible Erase Failure

Continue Programming at user’s option

Program 1 Location @ Vpp = 13.0V to 13.25V

Vdd = Vddp

N > 8? Yes

No

N = N + 1
(N = # of program pulses)

Pass? Yes

No

Apply 11N additional program pulses

Increment PC to point to next location, N = 0

All locations done?

Yes

Pass? Yes

No

Report Programming Failure

Now program Configuration Word

Verify Configuration Word @ Vddmax & Vddmin

Done

Verify all locations @ Vdd = Vddmin

Pass? Yes

No

Report verify failure @ Vddmin

Verify all locations @ Vdd = Vddmax

Pass? Yes

No

Report verify failure @ Vddmax
FIGURE 2-2: PIC12C5XX SERIES PROGRAM MEMORY MAP IN PROGRAM/VERIFY MODE

Address (Hex) 000

User Program Memory (NNN + 1) x 12 bit

0 0 0 ID0
0 0 0 ID1
0 0 0 ID2
0 0 0 ID3

For Customer Use
(4 x 4 bit usable)

For Factory Use

Configuration Word 5 bits

NNN Highest normal EPROM memory address. NNN = 0x1FF for PIC12C508/CE518. NNN = 0x3FF for PIC12C509/CE519. Note that some versions will have an oscillator calibration value programmed at NNN.

TTT Start address of special EPROM area and ID locations.
2.3 Special Memory Locations

The highest address of program memory space is reserved for the internal RC oscillator calibration value. This location should not be overwritten except when this location is blank, and it should be verified, when programmed, that it is a \texttt{MOVLW XX} instruction.

The ID Locations area is only enabled if the device is in programming/verify mode. Thus, in normal operation mode only the memory location 0x000 to 0xNNN will be accessed and the Program Counter will just roll over from address 0xNNN to 0x000 when incremented.

The configuration word can only be accessed immediately after MCLR going from \textit{Vil} to \textit{VIH}. The Program Counter will be set to all ‘1’s upon MCLR = \textit{Vil}. Thus, it has the value “0xFFF” when accessing the configuration EPROM. Incrementing the Program Counter once causes the Program Counter to roll over to all ‘0’s. Incrementing the Program Counter 4K times after reset (MCLR = \textit{Vil}) does not allow access to the configuration EPROM.

2.3.1 CUSTOMER ID CODE LOCATIONS

Per definition, the first four words (address TTT to TTT+3) are reserved for customer use. It is recommended that the customer use only the four lower order bits (bits 0 through 3) of each word and filling the eight higher order bits with ‘0’s.

A user may want to store an identification code (ID) in the ID locations and still be able to read this code after the code protection bit was programmed.

EXAMPLE 2-1: CUSTOMER CODE 0xD1E2

The Customer ID code “0xD1E2” should be stored in the ID locations 0x200-0x203 like this (PIC12C508/508A/CE518):

\begin{verbatim}
  200: 0000 0000 1101
  201: 0000 0000 0001
  202: 0000 0000 1110
  203: 0000 0000 0010
\end{verbatim}

Reading these four memory locations, even with the code protection bit programmed would still output on GP0 the bit sequence “1101”, “0001”, “1110”, “0010” which is “0xD1E2”.

\begin{footnotesize}
\begin{tabular}{|l|}
\hline
\textbf{Note:} All other locations in PICmicro® MCU configuration memory are reserved and should not be programmed. \\
\hline
\end{tabular}
\end{footnotesize}

2.4 Program/Verify Mode

The program/verify mode is entered by holding pins GP1 and GP0 low while raising MCLR pin from \textit{Vil} to \textit{VIH} (high voltage). Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial. GP0 and GP1 are Schmitt Trigger inputs in this mode.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at \textit{Vil}). This means that all I/O are in the reset state (High impedance inputs).

\begin{footnotesize}
\begin{tabular}{|l|}
\hline
\textbf{Note:} The MCLR pin should be raised from \textit{Vil} to \textit{VIH} within 9 ms of \textit{VDD} rise. This is to ensure that the device does not have the PC incremented while in valid operation range. \\
\hline
\end{tabular}
\end{footnotesize}
2.4.1 PROGRAM/VERIFY OPERATION

The GP1 pin is used as a clock input pin, and the GP0 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (GP1) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin GP0 is required to have a minimum setup and hold time (see AC/DC specs) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1 µs between the command and the data. After this delay the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first. Therefore, during a read operation the LSB will be transmitted onto pin GP0 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1 µs delay is also specified between consecutive commands.

All commands are transmitted LSB first. Data words are also transmitted LSB first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1 µs is required between a command and a data word (or another command).

The commands that are available are listed in Table 2-1.

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB ... LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Data</td>
<td>0 0 0 0 1 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Read Data</td>
<td>0 0 0 1 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>0 0 1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>Begin programming</td>
<td>0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>End Programming</td>
<td>0 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The clock must be disabled during in-circuit programming.
2.4.1.1 LOAD DATA

After receiving this command, the chip will load in a 14-bit "data word" when 16 cycles are applied, as described previously. Because this is a 12 bit core, the two msb's of the data word are ignored. A timing diagram for the load data command is shown in Figure 5-1.

2.4.1.2 READ DATA

After receiving this command, the chip will transmit data bits out of the memory currently accessed starting with the second rising edge of the clock input. The GP0 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. Because this is a 12-bit core, the two MSB's of the data are unused and read as '0'. A timing diagram of this command is shown in Figure 5-2.

2.4.1.3 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.4.1.4 BEGIN PROGRAMMING

A load data command must be given before every begin programming command. Programming of the appropriate memory (test program memory or user program memory) will begin after this command is received and decoded. Programming should be performed with a series of 100µs programming pulses. A programming pulse is defined as the time between the begin programming command and the end programming command.

2.4.1.5 END PROGRAMMING

After receiving this command, the chip stops programming the memory (configuration program memory or user program memory) that it was programming at the time.

2.5 Programming Algorithm Requires Variable Vdd

The PIC12C5XX uses an intelligent algorithm. The algorithm calls for program verification at Vddmin as well as Vddmax. Verification at Vddmin guarantees good "erase margin". Verification at Vddmax guarantees good "program margin".

The actual programming must be done with Vdd in the Vddp range (4.75 - 5.25V).

Vddp = Vcc range required during programming.
Vdd min. = minimum operating Vdd spec for the part.
Vddmax = maximum operating Vdd spec for the part.

Programmers must verify the PIC12C5XX at its specified Vddmax and Vddmin levels. Since Microchip may introduce future versions of the PIC12C5XX with a broader Vdd range, it is best that these levels are user selectable (defaults are ok).

Note: Any programmer not meeting these requirements may only be classified as "prototype" or "development" programmer but not a "production" quality programmer.
3.0 CONFIGURATION WORD

The PIC12C5XX family members have several configuration bits. These bits can be programmed (reads '0') or left unprogrammed (reads '1') to select various device configurations. Figure 3-1 provides an overview of configuration bits.

FIGURE 3-1: CONFIGURATION WORD BIT MAP

<table>
<thead>
<tr>
<th>Bit Number:</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C5XX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 11-5:</td>
<td>Reserved, '–' write as '0' for PIC12C5XX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 4:</td>
<td>MCLRE, Master Clear pin Enable Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = MCLR internally connected to Vdd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = MCLR pin enabled</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 3:</td>
<td>CP, Code Protect Enable Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Code Memory Unprotected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Code Memory Protected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 2:</td>
<td>WDTE, WDT Enable Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = WDT enabled</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = WDT disabled</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bit 1-0:</td>
<td>FOSC&lt;1:0&gt;, Oscillator Selection Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11: ExtRC oscillator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10: IntRC oscillator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01: XT oscillator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00: LP oscillator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.0 CODE PROTECTION

The program code written into the EPROM can be protected by writing to the CP bit of the configuration word. In PIC12C5XX, it is still possible to program and read locations 0x000 through 0x03F, after code protection. Once code protection is enabled, all protected segments read '0's (or "garbage values") and are prevented from further programming. All unprotected segments, including ID locations and configuration word, read normally. These locations can be programmed.

Once code protection is enabled, all code protected locations read 0's. All unprotected segments, including the internal oscillator calibration value, ID, and configuration word read as normal.

4.1 Embedding Configuration Word and ID Information in the Hex File

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

| TABLE 4-1: CODE PROTECTION |

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read Enabled, Write Enabled</td>
<td>Read Enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read Enabled, Write Enabled</td>
<td>Read Enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x1FF]</td>
<td>Read Disabled (all 0's), Write Disabled</td>
<td>Read Enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x200 : 0x203)</td>
<td>Read Enabled, Write Enabled</td>
<td>Read Enabled, Write Enabled</td>
</tr>
</tbody>
</table>

| PIC12C509 |

To code protect:

- (CP enable pattern: XXXXXXXX0XXX)

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x1FE]</td>
<td>Read disabled (all 0's), Write Disabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>0x1FF Oscillator Calibration Value</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x200 : 0x203)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
</tbody>
</table>

| PIC12C509A |

To code protect:

- (CP enable pattern: XXXXXXXX0XXX)

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x3FF]</td>
<td>Read disabled (all 0's), Write Disabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x400 : 0x403)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
</tbody>
</table>

| PIC12C509 |

To code protect:

- (CP enable pattern: XXXXXXXX0XXX)

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x3FF]</td>
<td>Read disabled (all 0's), Write Disabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x400 : 0x403)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
</tbody>
</table>
PIC12C509A
To code protect:
- (CP enable pattern: XXXXXXXX0XXX)

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x3FE]</td>
<td>Read disabled (all 0’s), Write Disabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>0x3FF Oscillator Calibration Value</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x400 : 0x403)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
</tbody>
</table>

PIC12CE518
To code protect:
- (CP enable pattern: XXXXXXXX0XXX)

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x1FE]</td>
<td>Read disabled (all 0’s), Write Disabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>0x1FF Oscillator Calibration Value</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x200 : 0x203)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
</tbody>
</table>

PIC12CE519
To code protect:
- (CP enable pattern: XXXXXXXX0XXX)

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0xFFF)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x00:0x3F]</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>[0x40:0x3FF]</td>
<td>Read disabled (all 0’s), Write Disabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x400 : 0x403)</td>
<td>Read enabled, Write Enabled</td>
<td>Read enabled, Write Enabled</td>
</tr>
</tbody>
</table>
4.2 Checksum

4.2.1 CHECKSUM CALCULATIONS

Checksum is calculated by reading the contents of the PIC12C5XX memory locations and adding up the opcodes up to the maximum user addressable location, (not including the last location which is reserved for the oscillator calibration value) e.g., 0x1FE for the PIC12C508/CE518. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC12C5XX family is shown in Table 4-2.

The checksum is calculated by summing the following:

- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

The oscillator calibration value location is not used in the above checksums.

**TABLE 4-2: CHECKSUM COMPUTATION**

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x723 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C508</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x01F</td>
<td>EE20</td>
<td>DC68</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x33F] + CFGW &amp; 0x01F + SUM(IDS)</td>
<td>EDF7</td>
<td>D363</td>
</tr>
<tr>
<td>PIC12C508A</td>
<td>OFF</td>
<td>SUM[0x000:0x1EE] + CFGW &amp; 0x01F</td>
<td>EE20</td>
<td>DC68</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x03F] + CFGW &amp; 0x01F + SUM(IDS)</td>
<td>EDF7</td>
<td>D363</td>
</tr>
<tr>
<td>PIC12C509</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x01F</td>
<td>EC20</td>
<td>DA68</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x33F] + CFGW &amp; 0x01F + SUM(IDS)</td>
<td>EBF7</td>
<td>D163</td>
</tr>
<tr>
<td>PIC12C509A</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x01F</td>
<td>EC20</td>
<td>DA68</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x33F] + CFGW &amp; 0x01F + SUM(IDS)</td>
<td>EBF7</td>
<td>D163</td>
</tr>
<tr>
<td>PIC12CE518</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x01F</td>
<td>EE20</td>
<td>DC68</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x33F] + CFGW &amp; 0x01F + SUM(IDS)</td>
<td>EDF7</td>
<td>D363</td>
</tr>
<tr>
<td>PIC12CE519</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x01F</td>
<td>EC20</td>
<td>DA68</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x33F] + CFGW &amp; 0x01F + SUM(IDS)</td>
<td>EBF7</td>
<td>D163</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word
SUM[0:a:b] = [Sum of locations a through b inclusive]
SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.
*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
+ = Addition
& = Bitwise AND
### 5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

**TABLE 5-1: AC/DC CHARACTERISTICS**

**TIMING REQUIREMENTS FOR PROGRAM/VERIFY MODE**

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD1</td>
<td>VDDP</td>
<td>Supply voltage during programming</td>
<td>4.75</td>
<td>5.0</td>
<td>5.25</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PD2</td>
<td>IDD</td>
<td>Supply current (from VDD) during programming</td>
<td></td>
<td></td>
<td>20</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD3</td>
<td>VDDV</td>
<td>Supply voltage during verify</td>
<td>VDDmin</td>
<td></td>
<td>VDDmax</td>
<td>V</td>
<td>Note 1</td>
</tr>
<tr>
<td>PD4</td>
<td>VIH1</td>
<td>Voltage on MCLR/VPP during programming</td>
<td>12.75</td>
<td>13.25</td>
<td></td>
<td>V</td>
<td>Note 2</td>
</tr>
<tr>
<td>PD5</td>
<td>VIH2</td>
<td>Voltage on MCLR/VPP during verify</td>
<td>VDD + 4.0</td>
<td>50</td>
<td></td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD6</td>
<td>IPP</td>
<td>Programming supply current (from VPP)</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD9</td>
<td>VIH</td>
<td>(GP1, GP0) input high level</td>
<td>0.8 VDD</td>
<td></td>
<td></td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>PD8</td>
<td>VIL</td>
<td>(GP1, GP0) input low level</td>
<td>0.2 VDD</td>
<td></td>
<td></td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
</tbody>
</table>

**Serial Program Verify**

| P1           | TR  | MCLR/VPP rise time (VSS to VHH) | 8.0   |      |      | µs     |            |
| P2           | TF  | MCLR Fall time | 8.0   |      |      | µs     |            |
| P3           | Tset1 | Data in setup time before clock ↓ | 100   |     |      | ns     |            |
| P4           | Thld1 | Data in hold time after clock ↓ | 100   |     |      | ns     |            |
| P5           | Tdly1 | Data input not driven to next clock input (delay required between command/data or command/command) | 1.0   |     |      | µs     |            |
| P6           | Tdly2 | Delay between clock ↓ to clock ↑ of next command or data | 1.0   |     |      | µs     |            |
| P7           | Tdly3 | Clock ↑ to date out valid (during read data) | 200   |     |      | ns     |            |
| P8           | Thld0 | Hold time after MCLR ↑ | 2     |     |      | µs     |            |

**Note 1:** Program must be verified at the minimum and maximum VDD limits for the part.

**Note 2:** VIHH must be greater than VDD + 4.5V to stay in programming/verify mode.
FIGURE 5-1: LOAD DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-2: READ DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-3: INCREMENT ADDRESS COMMAND (PROGRAM/VERIFY)
1.0 PROGRAMMING THE PIC12C67X AND PIC12CE67X

The PIC12C67X and PIC12CE67X can be programmed using a serial method. In serial mode the PIC12C67X and PIC12CE67X can be programmed while in the users system. This allows for increased design flexibility.

1.1 Hardware Requirements

The PIC12C67X and PIC12CE67X requires two programmable power supplies, one for VDD (2.0V to 6.0V recommended) and one for VPP (12V to 14V). Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC12C67X and PIC12CE67X allows programming of user program memory, special locations used for ID, and the configuration word for the PIC12C67X and PIC12CE67X.
2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The user memory space extends from 0x0000 to 0x1FFF (8K). Table 2-1 shows actual implementation of program memory in the PIC12C67X family.

<table>
<thead>
<tr>
<th>Device</th>
<th>Program Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C671/</td>
<td>0x000 - 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC12CE673</td>
<td></td>
</tr>
<tr>
<td>PIC12C672/</td>
<td>0x000 - 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC12CE674</td>
<td></td>
</tr>
</tbody>
</table>

When the PC reaches the last location of the implemented program memory, it will wrap around and address a location within the physically implemented memory (see Figure 2-1).

In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000-0x1FFF) being user program memory and the second half (0x2000-0x3FFF) being configuration memory. The PC will increment from 0x0000 to 0x1FFF and wrap to 0x000 or 0x2000 to 0x3FFF and wrap around to 0x2000 (not to 0x0000). Once in configuration memory, the highest bit of the PC stays a ‘1’, thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode, as described in Section 2.2.

The last location of the program memory space holds the factory programmed oscillator calibration value. This location should not be programmed except when blank (a non-blank value should not cause the device to fail a blank check). If blank, the programmer should program it to a RETLW XX statement where “XX” is the calibration value.

In the configuration memory space, 0x2000-0x20FF are utilized. When in configuration memory, as in the user memory, the 0x2000-0x2XFF segment is repeatedly accessed as the PC exceeds 0x2XFF (see Figure 2-1).

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000:0x2003].

**Note 1:** All other locations in PICmicro® MCU configuration memory are reserved and should not be programmed.

**Note 2:** Due to the secure nature of the on-board EEPROM memory in the PIC12CE673/674, it can be accessed only by the user program.
FIGURE 2-1: PROGRAM MEMORY MAPPING
2.2 Program/Verify Mode

The program/verify mode is entered by holding pins GP1 and GP0 low while raising MCLR pin from VIL to VIH (high voltage). VDD is then raised from VIL to VIH. Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program memory. GP1 is a Schmitt Trigger input in this mode.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at VIL). This means that all I/O are in the reset state (High impedance inputs).

2.2.1 PROGRAM/VERIFY OPERATION

The GP1 pin is used as a clock input pin, and the GP0 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (GP1) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin GP0 is required to have a minimum setup and hold time (see AC/DC specs) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1 µs between the command and the data. After this delay the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first. Therefore, during a read operation the LSB will be transmitted onto pin GP0 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1 µs delay is also specified between consecutive commands.

All commands are transmitted LSB first. Data words are also transmitted LSB first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1 µs is required between a command and a data word (or another command).

The commands that are available are listed in Table 1-1.

2.2.1.1 LOAD CONFIGURATION

After receiving this command, the program counter (PC) will be set to 0x2000. By then applying 16 cycles to the clock pin, the chip will load 14-bits a “data word” as described above, to be programmed into the configuration memory. A description of the memory mapping schemes for normal operation and configuration mode operation is shown in Figure 2-1. After the configuration memory is entered, the only way to get back to the user program memory is to exit the program/verify test mode by taking MCLR low (VIL).

Note 1: The MCLR pin must be raised from VIL to VIH before VDD is applied. This is to ensure that the device does not have the PC incremented while in valid operation range.

Note 2: Do not power GP2, GP4 or GP5 before VDD is applied.

### TABLE 1-1: COMMAND MAPPING

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB ... LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>000000</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Load Data</td>
<td>000001</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Read Data</td>
<td>000010</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>000011</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Begin programming</td>
<td>000100</td>
<td></td>
</tr>
<tr>
<td>End Programming</td>
<td>000110</td>
<td></td>
</tr>
</tbody>
</table>
**FIGURE 2-2: PROGRAM FLOW CHART - PIC12C67X AND PIC12CE67X PROGRAM MEMORY**

- Start
  - Set $V_{PP} = V_{IHH1}$
  - Set $V_{DD} = V_{DDP}$
  - $N = 0$
  - Program Cycle
    - Read Data Command
    - Data Correct?
      - Yes
        - Increment Address Command
      - No
        - Apply 3N Additional Program Cycles
        - All Locations Done?
          - Yes
            - Verify all Locations @ $V_{DD_{MIN}}$, $V_{PP} = V_{IHH2}$
          - No
            - Data Correct?
              - Yes
                - Report Verify @ $V_{DD_{MIN}}$ Error
              - No
                - Verify all Locations @ $V_{DD_{MAX}}$, $V_{PP} = V_{IHH2}$
        - $N = N + 1$
        - $N = \#$ of Program Cycles
        - $N > 25$
          - Yes
            - Report Programming Failure
          - No
            - All Locations Done?

- Program Cycle
  - Load Data Command
  - Begin Programming Command
  - Wait 100 $\mu$s
  - End Programming Command

- $V_{DDP} = V_{DD}$ range for programming (typically 4.75V - 5.25V).
- $V_{DD_{MIN}} = \text{Minimum } V_{DD}$ for device operation.
- $V_{DD_{MAX}} = \text{Maximum } V_{DD}$ for device operation.
FIGURE 2-3: PROGRAM FLOW CHART - PIC12C67X AND PIC12CE67X CONFIGURATION WORD & ID LOCATIONS

Start

Set $V_{PP} = V_{IH1}$

Load Configuration Command

$N = 0$

Program ID Loc?

Yes

Program Cycle

$N = N + 1$

$N = \# \text{ of Program Cycles}$

Data Correct?

Yes

Read Data Command

No

Apply 3N Program Cycles

Increment Address Command

Address = 2004

Yes

Increment Address Command

Increment Address Command

Increment Address Command

Increment Address Command

Increment Address Command

Program Cycle 100 Cycles

Data Correct?

Yes

Read Data Command

No

Data Correct?

Yes

Set $V_{DD} = V_{DDmax}$

Report Program ID/Config. Error

No

Data Correct?

Yes

Set $V_{PP} = V_{IH2}$

Yes

Read Data Command

Set $V_{DD} = V_{DDmin}$

Data Correct?

Yes

Read Data Command

Set $V_{PP} = V_{IH2}$

No

Done

Yes

Data Correct?
2.2.1.2 LOAD DATA

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 5-1.

2.2.1.3 READ DATA

After receiving this command, the chip will transmit data bits out of the memory currently accessed starting with the second rising edge of the clock input. The GP0 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 5-2.

2.2.1.4 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.2.1.5 BEGIN PROGRAMMING

A load command (load configuration or load data) must be given before every begin programming command. Programming of the appropriate memory (test program memory or user program memory) will begin after this command is received and decoded. Programming should be performed with a series of 100µs programming pulses. A programming pulse is defined as the time between the begin programming command and the end programming command.

2.2.1.6 END PROGRAMMING

After receiving this command, the chip stops programming the memory (configuration program memory or user program memory) that it was programming at the time.

2.3 Programming Algorithm Requires Variable Vdd

The PIC12C67X and PIC12CE67X uses an intelligent algorithm. The algorithm calls for program verification at VDDmin as well as VDDmax. Verification at VDDmin guarantees good “erase margin”. Verification at VDDmax guarantees good “program margin”.

The actual programming must be done with VDD in the VDDP range (4.75 - 5.25V).

VDDP = VCC range required during programming.

VDD min. = minimum operating VDD spec for the part.

VDD max. = maximum operating VDD spec for the part.

Programmers must verify the PIC12C67X and PIC12CE67X at its specified VDDmax and VDDmin levels. Since Microchip may introduce future versions of the PIC12C67X and PIC12CE67X with a broader VDD range, it is best that these levels are user selectable (defaults are ok).

Note: Any programmer not meeting these requirements may only be classified as “prototype” or “development” programmer but not a “production” quality programmer.
3.0 CONFIGURATION WORD

The PIC12C67X and PIC12CE67X family members have several configuration bits. These bits can be programmed (reads '0') or left unprogrammed (reads '1') to select various device configurations. Figure 3-1 provides an overview of configuration bits.

FIGURE 3-1: CONFIGURATION WORD

<table>
<thead>
<tr>
<th>Bit Number:</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP1</td>
<td>CP0</td>
<td>CP1</td>
<td>CP0</td>
<td>CP1</td>
<td>CP0</td>
<td>MCLRE</td>
<td>CP1</td>
<td>CP0</td>
<td>PWRTE</td>
<td>WDTE</td>
<td>FOSC2</td>
<td>FOSC1</td>
<td>FOSC0</td>
<td>Register: CONFIG</td>
</tr>
<tr>
<td>Address</td>
<td>2007h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bit 13-8, 6-5: CP1:CP0: Code Protection bits (1) (2)

- 11 = Code protection off
- 10 = 0400h-07FFh code protected;
- 01 = 0200h-07FFh code protected;
- 00 = 0000h-07FFh code protected;

bit 7: MCLRE: GP3/MCLR pin function select

- 1 = GP3/MCLR pin function is MCLR
- 0 = GP3/MCLR pin function is digital I/O, MCLR internally tied to Vdd

bit 4: PWRTE: Power-up Timer Enable bit (1)

- 1 = PWRT disabled
- 0 = PWRT enabled

bit 3: WDTE: Watchdog Timer Enable bit

- 1 = WDT enabled
- 0 = WDT disabled

bit 2-0: FOSC2:FOSC0: Oscillator Selection bits

- 111 = EXTRC oscillator / CLKOUT function on GP4/OSC2/CLKOUT pin
- 110 = EXTRC oscillator / GP4 function on GP4/OSC2/CLKOUT pin
- 101 = INTRC oscillator / CLKOUT function on GP4/OSC2/CLKOUT pin
- 100 = INTRC oscillator / GP4 function on GP4/OSC2/CLKOUT pin
- 011 = invalid selection
- 010 = HS oscillator
- 001 = XT oscillator
- 000 = LP oscillator

3: All of the CP1:CP0 pairs have to be given the same value to enable the code protection scheme listed.

4: 07FFh is always uncode protected on the 12C672 and 03FFh is always uncode protected on the 12C671. This location contains the RETLW xx calibration instruction for the INTRC.
4.0 CODE PROTECTION

The program code written into the EPROM can be protected by writing to the CP0 & CP1 bits of the configuration word.

For PIC12C67X and PIC12CE67X devices, once code protection is enabled, all protected segments read '0's (or "garbage values") and are prevented from further programming. All unprotected segments, including ID and configuration word locations, and calibration word location read normally and can be programmed.

4.1 Embedding Configuration Word and ID Information in the Hex File

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

<table>
<thead>
<tr>
<th>TABLE 1-2: CONFIGURATION WORD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PIC12C671, PIC12CE673</strong></td>
</tr>
<tr>
<td>To code protect:</td>
</tr>
<tr>
<td>• Protect all memory 00 0000 X00X XXXX</td>
</tr>
<tr>
<td>• Protect 0200h-07FFh 01 0101 X01X XXXX</td>
</tr>
<tr>
<td>• No code protection 11 1111 X11X XXXX</td>
</tr>
<tr>
<td>Program Memory Segment</td>
</tr>
<tr>
<td>Configuration Word (0x2007)</td>
</tr>
<tr>
<td>Unprotected memory segment</td>
</tr>
<tr>
<td>Protected memory segment</td>
</tr>
<tr>
<td>ID Locations (0x2000 : 0x2003)</td>
</tr>
<tr>
<td>INTRC Calibration Word (0X3FF)</td>
</tr>
</tbody>
</table>

| **PIC12C672, PIC12CE674** |
| To code protect: |
| • Protect all memory 00 0000 X00X XXXX |
| • Protect 0200h-07FFh 01 0101 X01X XXXX |
| • Protect 0400h-07FFh 10 1010 X10X XXXX |
| • No code protection 11 1111 X11X XXXX |
| Program Memory Segment | R/W in Protected Mode | R/W in Unprotected Mode |
| Configuration Word (0x2007) | Read Unscrambled, Write Enabled | Read Unscrambled, Write Enabled |
| Unprotected memory segment | Read Unscrambled, Write Enabled | Read Unscrambled, Write Enabled |
| Protected memory segment | Read All 0’s, Write Disabled | Read Unscrambled, Write Enabled |
| ID Locations (0x2000 : 0x2003) | Read Unscrambled, Write Enabled | Read Unscrambled, Write Enabled |
| INTRC Calibration Word (0X7FF) | Read Unscrambled, Write Enabled | Read Unscrambled, Write Enabled |
4.2 Checksum

4.2.1 CHECKSUM CALCULATIONS

Checksum is calculated by reading the contents of the PIC12C67X and PIC12CE67X memory locations and adding the opcodes up to the maximum user addressable location, excluding the oscillator calibration location in the last address, e.g., 0x3FE for the PIC12C671/CE673. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC12C67X and PIC12CE67X devices is shown in Table 4-1.

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

**TABLE 4-1: CHECKSUM COMPUTATION**

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>Ox25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC12C671</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x3FFF</td>
<td>3B3F</td>
<td>070D</td>
</tr>
<tr>
<td>PIC12C671</td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3FFF + SUM_ID</td>
<td>4E5E</td>
<td>0013</td>
</tr>
<tr>
<td>PIC12C671</td>
<td>ALL</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>3B4E</td>
<td>071C</td>
</tr>
<tr>
<td>PIC12CE673</td>
<td>OFF</td>
<td>SUM[0x000:0x3FE] + CFGW &amp; 0x3FFF</td>
<td>3B3F</td>
<td>070D</td>
</tr>
<tr>
<td>PIC12CE673</td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3FFF + SUM_ID</td>
<td>4E5E</td>
<td>0013</td>
</tr>
<tr>
<td>PIC12CE673</td>
<td>ALL</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>3B4E</td>
<td>071C</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word
SUM[a:b] = [Sum of locations a through b inclusive]
SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble.
For example,
ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.
*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
+ = Addition
& = Bitwise AND
### 5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

**TABLE 1-3: AC/DC CHARACTERISTICS**

**TIMING REQUIREMENTS FOR PROGRAM/VERIFY TEST MODE**

Standard Operating Conditions
- Operating Temperature: \( +10^\circ C \leq T_A \leq +40^\circ C \), unless otherwise stated, \((25^\circ C \text{ is recommended})\)
- Operating Voltage: \( 4.5V \leq V_{DD} \leq 5.5V \), unless otherwise stated.

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD1</td>
<td>VDDP</td>
<td>Supply voltage during programming</td>
<td>4.75</td>
<td>5.0</td>
<td>5.25</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PD2</td>
<td>IDD</td>
<td>Supply current (from VDD) during programming</td>
<td></td>
<td></td>
<td>20</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD3</td>
<td>VDDV</td>
<td>Supply voltage during verify</td>
<td>VDDmin</td>
<td></td>
<td>VDDmax</td>
<td>V</td>
<td>Note 1</td>
</tr>
<tr>
<td>PD4</td>
<td>VIHH</td>
<td>Voltage on MCLR/VPP during programming</td>
<td>12.75</td>
<td>13.25</td>
<td></td>
<td>V</td>
<td>Note 2</td>
</tr>
<tr>
<td>PD5</td>
<td>VIHH2</td>
<td>Voltage on MCLR/VPP during verify</td>
<td>VDD + 4.0</td>
<td></td>
<td>13.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD6</td>
<td>IPP</td>
<td>Programming supply current (from VPP)</td>
<td></td>
<td></td>
<td>50</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD9</td>
<td>VIH</td>
<td>(GP0, GP1) input high level</td>
<td>0.8 VDD</td>
<td></td>
<td></td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>PD8</td>
<td>Vil</td>
<td>(GP0, GP1) input low level</td>
<td>0.2 VDD</td>
<td></td>
<td></td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
</tbody>
</table>

**Serial Program Verify**

| P1 | TR | MCLR/VPP rise time (VSS to VIHH) for test mode entry | 8.0 | \( \mu s \) |
| P2 | Tf | MCLR Fall time | 8.0 | \( \mu s \) |
| P3 | Tset1 | Data in setup time before clock ↓ | 100 | ns |
| P4 | Thld1 | Data in hold time after clock ↓ | 100 | ns |
| P5 | Tdly1 | Data input not driven to next clock input (delay required between command/data or command/command) | 1.0 | \( \mu s \) |
| P6 | Tdly2 | Delay between clock ↓ to clock ↑ of next command or data | 1.0 | \( \mu s \) |
| P7 | Tdly3 | Clock ↑ to data out valid (during read data) | 200 | ns |
| P8 | Thld0 | Hold time after VDD↑ | 2 | \( \mu s \) |
| P9 | TPDP | Hold time after VPP↑ | 5 | \( \mu s \) |

**Note 1:** Program must be verified at the minimum and maximum VDD limits for the part.
**Note 2:** VIHH must be greater than VDD + 4.5V to stay in programming/verify mode.
FIGURE 5-1: LOAD DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-2: READ DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-3: INCREMENT ADDRESS COMMAND (PROGRAM/VERIFY)
This document includes the programming specifications for the following devices:

• PIC14000

1.0 PROGRAMMING THE PIC14000

The PIC14000 can be programmed using a serial method. In serial mode the PIC14000 can be programmed while in the user's system. This allows for increased design flexibility. This programming specification applies to PIC14000 devices in all packages.

1.1 Hardware Requirements

The PIC14000 requires two programmable power supplies, one for VDD (2.0V to 6.5V recommended) and one for VPP (12V to 14V).

1.2 Programming Mode

The programming mode for the PIC14000 allows programming of user program memory, configuration word, and calibration memory.
2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The program and calibration memory space extends from 0x000 to 0xFFF (4096 words). Table 2-1 shows actual implementation of program memory in the PIC14000.

TABLE 2-1: IMPLEMENTATION OF PROGRAM AND CALIBRATION MEMORY IN THE PIC14000P

<table>
<thead>
<tr>
<th>Area</th>
<th>Memory Space</th>
<th>Access to Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>0x000-0xFB0</td>
<td>PC&lt;12:0&gt;</td>
</tr>
<tr>
<td>Calibration</td>
<td>0xFC0-0xFFF</td>
<td>PC&lt;12:0&gt;</td>
</tr>
</tbody>
</table>

When the PC reaches address 0xFFF, it will wrap around and address a location within the physically implemented memory (see Figure 2-1).

In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000-0x1FFF) being user program memory and the second half (0x2000-0x3FFF) being configuration memory. The PC will increment from 0x0000 to 0x1FFF and wrap to 0x0000, or 0x2000 to 0x3FFF and wrap around to 0x2000 (not to 0x0000). Once in configuration memory, the highest bit of the PC stays a ‘1’, thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode, as described in Section 2.2.

In the configuration memory space, 0x2000-0x20FF are utilized. When in configuration memory, as in the user memory, the 0x2000-0x2XFF segment is repeatedly accessed as PC exceeds 0x2XFF (Figure 2-1).

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000: 0x2003]. All other locations are reserved and should not be programmed.

The ID locations read out normally, even after code protection. To understand how the devices behave, refer to Table 4-1.

To understand the scrambling mechanism after code protection, refer to Section 4.1.
FIGURE 2-1: PROGRAM MEMORY MAPPING
2.2 Program/Verify Mode

The program/verify mode is entered by holding pins RC6 and RC7 low while raising MCLR pin from VIL to VIH (high voltage). Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program memory. RC6 and RC7 are both Schmitt Trigger inputs in this mode.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at VIL). This means that all I/O are in the reset state (High impedance inputs).

The RB6 pin is used as a clock input pin, and the RB7 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (RC6) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin RC7 is required to have a minimum setup and hold time (see AC/DC specs) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1µs between the command and the data. After this delay the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first. Therefore, during a read operation the LSB will be transmitted onto pin RC7 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1µs delay is also specified between consecutive commands.

All commands are transmitted LSB first. Data words are also transmitted LSB first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1µs is required between a command and a data word (or another command).

The commands that are available are listed in Table 2-1. Note: The CPU clock must be disabled during in-circuit programming (to avoid incrementing the PC).

### TABLE 2-1: COMMAND MAPPING

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB ... LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>0 0 0 0 0 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Load Data</td>
<td>0 0 0 0 0 1 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Read Data</td>
<td>0 0 0 1 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>0 0 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>Begin programming</td>
<td>0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>End Programming</td>
<td>0 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Note: The MCLR pin should be raised as quickly as possible from VIL to VIHH. This is to ensure that the device does not have the PC incremented while in valid operation range.
FIGURE 2-2: PROGRAM FLOW CHART - PIC14000 PROGRAM MEMORY AND CALIBRATION

Start

Set VDD = VDDP*

N = 0

Program Cycle

Read Data Command

Data Correct?

No

Apply 3N Additional Program Cycles

No

All Locations Done?

Yes

Verify all Locations @ VDD min.*
VPP = VIHH2

Data Correct?

No

Report Verify @ VDD min. Error

Yes

Verify all Locations @ VDD max.
VPP = VIHH2

Data Correct?

No

Report Verify @ VDD max. Error

Yes

Done

No

N > 25

Yes

Report Programming Failure

Yes

N = N + 1

N = # of Program Cycles

Program Cycle

Load Data Command

Begin Programming Command

Wait 100 µs

End Programming Command

Increment Address Command
FIGURE 2-3: PROGRAM FLOW CHART - PIC14000 CONFIGURATION WORD & ID LOCATIONS

Start

Load Configuration Command

N = 0

Program ID Loc?

Program Cycle

N = N + 1 N = # of Program Cycles

Data Correct?

Report ID Configuration Error

Apply 3N Program Cycles

Increment Address Command

Address = 2004

Increment Address Command

Increment Address Command

Increment Address Command

Read Data Command

Program Cycle 100 Cycles

Data Correct?

Report Program ID/Config. Error

Set VDD = VDDmin

Read Data Command

Set VDD = VDDmax

Read Data Command

Set VPP = VIHH2

Done

Data Correct?

No

Yes

Yes

Yes

Yes

No

Yes
2.2.1.2 LOAD DATA

After receiving this command, the chip will load in a 14-bit "data word" when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 5-1.

2.2.1.3 READ DATA

After receiving this command, the chip will transmit data bits out of the memory currently accessed starting with the second rising edge of the clock input. The RC7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 5-2.

2.2.1.4 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.2.1.5 BEGIN PROGRAMMING

A load command (load configuration or load data) must be given before every begin programming command. Programming of the appropriate memory (test program memory or user program memory) will begin after this command is received and decoded. Programming should be performed with a series of 100µs programming pulses. A programming pulse is defined as the time between the begin programming command and the end programming command.

2.2.1.6 END PROGRAMMING

After receiving this command, the chip stops programming the memory (configuration program memory or user program memory) that it was programming at the time.

2.3 Programming Algorithm Requires Variable VDD

The PIC14000 uses an intelligent algorithm. The algorithm calls for program verification at VDDmin as well as VDDmax. Verification at VDDmin guarantees good "erase margin". Verification at VDDmax guarantees good "program margin".

The actual programming must be done with VDD in the VDDP range (4.75 - 5.25V).

VDDP = VCC range required during programming.
VDDmin = minimum operating VDD spec for the part.
VDDmax = maximum operating VDD spec for the part.

Programmers must verify the PIC14000 at its specified VDDmax and VDDmin levels. Since Microchip may introduce future versions of the PIC14000 with a broader VDD range, it is best that these levels are user selectable (defaults are ok).

Note: Any programmer not meeting these requirements may only be classified as "prototype" or "development" programmer but not a "production" quality programmer.
3.0 CONFIGURATION WORD

The PIC14000 has several configuration bits. These bits can be programmed (reads '0') or left unprogrammed (reads '1') to select various device configurations. Figure 3-1 provides an overview of configuration bits.

FIGURE 3-1: CONFIGURATION WORD BIT MAP

<table>
<thead>
<tr>
<th>Bit Number:</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC14000 CPC</td>
<td>CPP1</td>
<td>CPP0</td>
<td>CPP0</td>
<td>CPP1</td>
<td>CPC</td>
<td>CPC</td>
<td>F</td>
<td>CPP1</td>
<td>CPP0</td>
<td>PWRTE</td>
<td>WDTE</td>
<td>F</td>
<td>FOSC</td>
<td></td>
</tr>
</tbody>
</table>

CPP<1:0>
- 11: All Unprotected
- 10: N/A
- 01: N/A
- 00: All Protected

bit 1,6: F Internal trim, factory programmed. DO NOT CHANGE! Program as ‘1’. Note 1.

bit 3: PWRTE, Power Up Timer Enable Bit
- 0 = Power up timer enabled
- 1 = Power up timer disabled (unprogrammed)

bit 2: WDTE, WDT Enable Bit
- 0 = WDT disabled
- 1 = WDT enabled (unprogrammed)

bit 0: FOSC<1:0>, Oscillator Selection Bit
- 0: HS oscillator (crystal/resonator)
- 1: Internal RC oscillator (unprogrammed)

Note 1: See Section 4.1.2 for cautions.
4.0 CODE PROTECTION

The memory space in the PIC14000 is divided into two areas: program space (0-0xFBF) and calibration space (0xFC0-0xFFF).

For program space or user space, once code protection is enabled, all protected segments read ‘0’s (or “garbage values”) and are prevented from further programming. All unprotected segments, including ID locations and configuration word, read normally. These locations can be programmed.

4.1 Calibration Space

The calibration space can contain factory-generated and programmed values. For non-JW devices, the CPC bits in the configuration word are set to ‘0’ at the factory, and the calibration data values are write-protected; they may still be read out, but not programmed. JW devices contain the factory values, but DO NOT have the CPC bits set.

Microchip does not recommend setting code protect bits in windowed devices to ‘0’. Once code-protected, the device cannot be reprogrammed.

4.1.1 CALIBRATION SPACE CHECKSUM

The data in the calibration space has its own checksum. When properly programmed, the calibration memory will always checksum to 0x0000. When this checksum is 0x0000, and the checksum of memory [0x0000:0x2FBF] is 0x2FBF, the part is effectively blank, and the programmer should indicate such.

If the CPC bits are set to ‘1’, but the checksum of the calibration memory is 0x0000, the programmer should NOT program locations in the calibration memory space, even if requested to do so by the operator. This would be the case for a new JW device.

If the CPC bits are set to ‘1’, and the checksum of the calibration memory is NOT 0x0000, the programmer is allowed to program the calibration space as directed by the operator.

The calibration space contains specially coded data values used for device parameter calibration. The programmer may wish to read these values and display them for the operator’s convenience. For further information on these values and their coding, refer to AN621 (DS00621B).

4.1.2 REPROGRAMMING CALIBRATION SPACE

The operator should be allowed to read and store the data in the calibration space, for future reprogramming of the device. This procedure is necessary for reprogramming a windowed device, since the calibration data will be erased along with the rest of the memory. When saving this data, Configuration Word <1,6> must also be saved, and restored when the calibration data is reloaded.

4.2 Embedding Configuration Word and ID Information in the Hex File

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

---

### TABLE 4-1: CODE PROTECT OPTIONS

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Unprotected memory segment</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protected memory segment</td>
<td>Read All 0’s, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protected calibration memory</td>
<td>Read Unscrambled, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x2000 : 0x2003)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

Legend: X = Don’t care
4.3 Checksum

4.3.1 CHECKSUM CALCULATIONS

Checksum is calculated by reading the contents of the PIC14000 memory locations and adding up the opcodes up to the maximum user addressable location, 0xFBF. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for the PIC14000 device is shown in Table 4-2:

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

### TABLE 4-2: CHECKSUM COMPUTATION

<table>
<thead>
<tr>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>SUM[0000:0FBF] + CFGW &amp; 0x3FBD</td>
<td>0x2FFD</td>
<td>0xFB8B</td>
</tr>
<tr>
<td>OFF OTP</td>
<td>SUM[0000:0FBF] + CFGW &amp; 0x3FBD</td>
<td>0x0E7D</td>
<td>0xDA4B</td>
</tr>
<tr>
<td>ON</td>
<td>CFGW &amp; 0x3FBD + SUM(IDs)</td>
<td>0x300A</td>
<td>0xFB8D</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word  
SUM[A:B] = [Sum of locations a through b inclusive]  
SUM(ID) = ID locations masked by 0x7F then made into a 28-bit value with ID0 as the most significant byte  
*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]  
+ = Addition  
& = Bitwise AND
## 5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

### TABLE 5-1: AC/DC CHARACTERISTICS

#### AC/DC TIMING REQUIREMENTS FOR PROGRAM/VERIFY MODE

**Standard Operating Conditions**
Operating Temperature: $+10^\circ C \leq T_A \leq +40^\circ C$, unless otherwise stated, (25°C recommended)
Operating Voltage: $4.5V \leq V_{DD} \leq 5.5V$, unless otherwise stated.

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD1</td>
<td>VDDP</td>
<td>Supply voltage during programming</td>
<td>4.75</td>
<td>5.0</td>
<td>5.25</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PD2</td>
<td>IDD</td>
<td>Supply current (from VDD) during programming</td>
<td>–</td>
<td>–</td>
<td>20</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD3</td>
<td>VDDV</td>
<td>Supply voltage during verify</td>
<td>VDDmin</td>
<td>VDDmax</td>
<td>V</td>
<td>Note 1</td>
<td></td>
</tr>
<tr>
<td>PD4</td>
<td>VIHH1</td>
<td>Voltage on MCLR/VPP during programming</td>
<td>12.75</td>
<td>–</td>
<td>13.25</td>
<td>V</td>
<td>Note 2</td>
</tr>
<tr>
<td>PD5</td>
<td>VIHH2</td>
<td>Voltage on MCLR/VPP during verify</td>
<td>VDD + 4.0</td>
<td>13.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD6</td>
<td>IPP</td>
<td>Programming supply current (from VPP)</td>
<td>–</td>
<td>–</td>
<td>50</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD9</td>
<td>VIH1</td>
<td>(RC6, RC7) input high level</td>
<td>0.8</td>
<td>VDD</td>
<td>–</td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>PD8</td>
<td>VIL1</td>
<td>(RC6, RC7) input low level</td>
<td>0.2</td>
<td>VDD</td>
<td>–</td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td><strong>Serial Program Verify</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>TR</td>
<td>MCLR/VPP rise time (VSS to VHH) for test mode entry</td>
<td>–</td>
<td>–</td>
<td>8.0</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>TF</td>
<td>MCLR Fall time</td>
<td>–</td>
<td>–</td>
<td>8.0</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>TSET1</td>
<td>Data in setup time before clock ↓</td>
<td>100</td>
<td>–</td>
<td>–</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>THLD1</td>
<td>Data in hold time after clock ↓</td>
<td>100</td>
<td>–</td>
<td>–</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>TDLY1</td>
<td>Data input not driven to next clock input (delay required between command/data or command/command)</td>
<td>1.0</td>
<td>–</td>
<td>–</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>TDLY2</td>
<td>Delay between clock ↓ to clock ↑ of next command or data</td>
<td>1.0</td>
<td>–</td>
<td>–</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>TDLY3</td>
<td>Clock ↑ to date out valid (during read data)</td>
<td>200</td>
<td>–</td>
<td>–</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>THLD0</td>
<td>Hold time after MCLR ↑</td>
<td>2</td>
<td>–</td>
<td>–</td>
<td>µs</td>
<td></td>
</tr>
</tbody>
</table>

**Note 1:** Program must be verified at the minimum and maximum VDD limits for the part.
**Note 2:** VIHH must be greater than VDD + 4.5V to stay in programming/verify mode.
This document includes the programming specifications for the following devices:

- PIC16C554
- PIC16C556
- PIC16C558

1.0 PROGRAMMING THE PIC16C55X

The PIC16C55X can be programmed using a serial method. In serial mode the PIC16C55X can be programmed while in the user's system. This allows for increased design flexibility.

1.1 Hardware Requirements

The PIC16C55X requires two programmable power supplies, one for VDD (2.0V to 6.5V recommended) and one for VPP (12V to 14V). Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC16C55X allows programming of user program memory, special locations used for ID, and the configuration word for the PIC16C55X.
PIC16C55X

2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The user memory space extends from 0x0000 to 0x1FFF (8K). Table 2-1 shows actual implementation of program memory in the PIC16C55X family.

<table>
<thead>
<tr>
<th>Device</th>
<th>Program Memory Size</th>
<th>Access to Program Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C554</td>
<td>0x0000 - 0x1FF (0.5K)</td>
<td>PC&lt;8:0&gt;</td>
</tr>
<tr>
<td>PIC16C556</td>
<td>0x0000 - 0x3FF (1K)</td>
<td>PC&lt;9:0&gt;</td>
</tr>
<tr>
<td>PIC16C558</td>
<td>0x0000 - 0x7FF (2K)</td>
<td>PC&lt;10:0&gt;</td>
</tr>
</tbody>
</table>

When the PC reaches the last location of the implemented program memory, it will wrap around and address a location within the physically implemented memory (see Figure 2-1).

In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000-0x1FFF) being user program memory and the second half (0x2000-0x3FFF) being configuration memory. The PC will increment from 0x0000 to 0x1FFF and wrap to 0x000 or 0x2000 to 0x3FFF and wrap around to 0x2000 (not to 0x0000). Once in configuration memory, the highest bit of the PC stays a '1', thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode, as described in Section 2.2.

In the configuration memory space, 0x2000-0x20FF are utilized. When in a configuration memory, as in the user memory, the 0x2000-0x2XFF segment is repeatedly accessed as the PC exceeds 0x2XXF (see Figure 2-1).

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000:0x2003]. It is recommended that the user use only the four least significant bits of each ID location. In some devices, the ID locations read-out in a scrambled fashion after code protection is enabled. For these devices, it is recommended that ID location is written as "11111100 bbbb" where 'bbbb' is ID information.

Note: All other locations are reserved and should not be programmed.

In other devices, the ID locations read out normally, even after code protection. To understand how the devices behave, refer to Table 4-1.

To understand the scrambling mechanism after code protection, refer to Section 4.1.
FIGURE 2-1: PROGRAM MEMORY MAPPING

The diagram shows the program memory mapping for the PIC16C55X microcontroller. The memory space is divided into 0.5KW, 1KW, and 2KW segments, with the ID Location, ID Location, and Configuration Word sections marked as reserved. The 0.5KW segment contains implemented locations at 0000H-1FFE H, with the remaining space reserved. The 1KW segment has implemented locations at 2000H-3FFFH, and the 2KW segment starts from 4000H, followed by an implemented location at 4008H and reserved locations thereafter.

<table>
<thead>
<tr>
<th>Segment</th>
<th>0000H-1FFE H</th>
<th>2000H-3FFFH</th>
<th>4000H-6FFFH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5KW</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>1KW</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Reserved</td>
</tr>
<tr>
<td>2KW</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Reserved locations are indicated by shaded areas in the diagram.
## 2.2 Program/Verify Mode

The program/verify mode is entered by holding pins RB6 and RB7 low while raising MCLR pin from VIL to VIHH (high voltage). Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program and configuration memory. RB6 is a Schmitt Trigger input in this mode.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at VIL). This means that all I/O are in the reset state (High impedance inputs).

### 2.2.1 PROGRAM/VERIFY OPERATION

The RB6 pin is used as a clock input pin, and the RB7 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (RB6) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin RB7 is required to have a minimum setup and hold time (see AC/DC specs) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1μs between the command and the data. After this delay the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first. Therefore, during a read operation the LSB will be transmitted onto pin RB7 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1μs delay is also specified between consecutive commands.

The commands that are available are listed in Table 2-1.

### 2.2.1.1 LOAD CONFIGURATION

After receiving this command, the program counter (PC) will be set to 0x2000. By then applying 16 cycles to the clock pin, the chip will load 14-bits a “data word” as described above, to be programmed into the configuration memory. A description of the memory mapping schemes for normal operation and configuration mode operation is shown in Figure 2-1. After the configuration memory is entered, the only way to get back to the user program memory is to exit the program/verify test mode by taking MCLR low (VIL).

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB ... LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>0 0 0 0 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Load Data</td>
<td>0 0 0 0 1 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Read Data</td>
<td>0 0 0 1 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>0 0 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>Begin programming</td>
<td>0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>End Programming</td>
<td>0 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The CPU clock must be disabled during in-circuit programming.
FIGURE 2-2: PROGRAM FLOW CHART - PIC16C55X PROGRAM MEMORY

* VDDP = VDD range for programming (typically 4.75V - 5.25V).
VDDmin = Minimum VDD for device operation.
VDDmax = Maximum VDD for device operation.

---

* VDDP = VDD range for programming (typically 4.75V - 5.25V).
VDDmin = Minimum VDD for device operation.
VDDmax = Maximum VDD for device operation.
FIGURE 2-3: PROGRAM FLOW CHART - PIC16C55X CONFIGURATION WORD & ID LOCATIONS

Start

Load Configuration Command

N = 0

Program ID Loc? Yes

Program Cycle

N = N + 1 N ≠ # of Program Cycles

N > 25

Data Correct? Yes

ID/Configuration Error

Program Cycle

100 Cycles

Read Data Command

Data Correct? Yes

Report Program ID/Config. Error

Data Correct? No

Set VDD = VDDmin

Read Data Command

Set VDD = VDDmax

Data Correct? Yes

Set VPP = VIHH2

Data Correct? No

Set VPP = VIHH2

Read Data Command

Done

Address = 2004

Increment Address Command

Increment Address Command

Increment Address Command

Increment Address Command

No

Yes
2.2.1.2 LOAD DATA
After receiving this command, the chip will load in a 14-bit "data word" when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 5-1.

2.2.1.3 READ DATA
After receiving this command, the chip will transmit data bits out of the memory currently accessed starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 5-2.

2.2.1.4 INCREMENT ADDRESS
The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.2.1.5 BEGIN PROGRAMMING
A load command (load configuration or load data) must be given before every begin programming command. Programming of the appropriate memory (test program memory or user program memory) will begin after this command is received and decoded. Programming should be performed with a series of 100µs programming pulses. A programming pulse is defined as the time between the begin programming command and the end programming command.

2.2.1.6 END PROGRAMMING
After receiving this command, the chip stops programming the memory (configuration program memory or user program memory) that it was programming at the time.

2.3 Programming Algorithm Requires Variable Vdd
The PIC16C55X uses an intelligent algorithm. The algorithm calls for program verification at Vddmin as well as Vddmax. Verification at Vddmin guarantees good "erase margin". Verification at Vddmax guarantees good "program margin".

The actual programming must be done with Vdd in the Vddop range (4.75 - 5.25V).

Vddop = Vcc range required during programming.
Vdd min. = minimum operating Vdd spec for the part.
Vdd max. = maximum operating Vdd spec for the part.

Programmers must verify the PIC16C55X at its specified Vddmax and Vddmin levels. Since Microchip may introduce future versions of the PIC16C55X with a broader Vdd range, it is best that these levels are user selectable (defaults are ok).

Note: Any programmer not meeting these requirements may only be classified as "prototype" or "development" programmer but not a "production" quality programmer.
3.0 CONFIGURATION WORD

The PIC16C55X family members have several configuration bits. These bits can be programmed (reads '0') or left unprogrammed (reads '1') to select various device configurations. Figure 3-1 provides an overview of configuration bits.

FIGURE 3-1: CONFIGURATION WORD BIT MAP

| Bit Number: 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PIC16C554/556/558 | CP1 | CP0 | CP1 | CP0 | CP1 | CP0 | PWRTE | WDTE | FOSC1 | FOSC0 |

bit 7: Reserved for future use
bit 6: Set to 0
bit 5-4: CP1:CP0, Code Protect
bit 8-13

<table>
<thead>
<tr>
<th>Device</th>
<th>CP1</th>
<th>CP0</th>
<th>Code Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C554</td>
<td>0</td>
<td>0</td>
<td>All memory protected</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Do not use</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Do not use</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Code protection off</td>
</tr>
<tr>
<td>PIC16C556</td>
<td>0</td>
<td>0</td>
<td>All memory protected</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Upper 1/2 memory protected</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Do not use</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Code protection off</td>
</tr>
<tr>
<td>PIC16C558</td>
<td>0</td>
<td>0</td>
<td>All memory protected</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Upper 3/4 memory protected</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Upper 1/2 memory protected</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Code protection off</td>
</tr>
</tbody>
</table>

bit 3: PWRTE, Power Up Timer Enable Bit
- PIC16C554/556/558:
  - 0 = Power up timer enabled
  - 1 = Power up timer disabled

bit 2: WDTE, WDT Enable Bit
- 1 = WDT enabled
- 0 = WDT disabled

bit 1-0: FOSC<1:0>, Oscillator Selection Bit
- 11: RC oscillator
- 10: HS oscillator
- 01: XT oscillator
- 00: LP oscillator
4.0 CODE PROTECTION

The program code written into the EPROM can be protected by writing to the CP0 & CP1 bits of the configuration word.

4.1 Programming Locations 0x0000 to 0x03F after Code Protection

For PIC16C55X devices, once code protection is enabled, all protected segments read '0's (or "garbage values") and are prevented from further programming. All unprotected segments, including ID locations and configuration word, read normally. These locations can be programmed.

4.2 Embedding Configuration Word and ID Information in the Hex File

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

### TABLE 4-1: CONFIGURATION WORD

<table>
<thead>
<tr>
<th>PIC16C554</th>
<th>0000001000XXXX</th>
<th>1111111011XXXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protect all memory</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>No code protection</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protected memory segment</td>
<td>Read All 0's, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x2000 : 0x2003)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIC16C556</th>
<th>0000001000XXXX</th>
<th>1111111011XXXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protect all memory</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protect upper 1/2 memory</td>
<td>0101011101XXXX</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>No code protection</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protected memory segment</td>
<td>Read All 0's, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x2000 : 0x2003)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIC16C558</th>
<th>0000001000XXXX</th>
<th>1111111011XXXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protect all memory</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protect upper 3/4 memory</td>
<td>0101011101XXXX</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protect upper 1/2 memory</td>
<td>1010101010XXXX</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>No code protection</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>Protected memory segment</td>
<td>Read All 0's, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations (0x2000 : 0x2003)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>
4.3 Checksum

4.3.1 CHECKSUM CALCULATIONS

Checksum is calculated by reading the contents of the PIC16C55X memory locations and adding up the opcodes up to the maximum user addressable location, e.g., 0x1FF for the PIC16C74. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC16C55X devices is shown in Table.

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C554</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F3F</td>
<td>3D3F</td>
<td>090D</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>SUM_ID + CFGW &amp; 0x3F3F</td>
<td>3D4E</td>
<td>091C</td>
</tr>
<tr>
<td>PIC16C556</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F3F</td>
<td>3B3F</td>
<td>070D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F3F + SUM_ID</td>
<td>4E5E</td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F3F + SUM_ID</td>
<td>3B4E</td>
<td>071C</td>
</tr>
<tr>
<td>PIC16C558</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F3F</td>
<td>373F</td>
<td>030D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F3F + SUM_ID</td>
<td>5D6E</td>
<td>0F23</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F3F + SUM_ID</td>
<td>4A5E</td>
<td>FC13</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F3F + SUM_ID</td>
<td>374E</td>
<td>031C</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word
SUM[a:b] = \([\text{sum of locations a through b inclusive}]\)
SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble.

For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.

*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
+ = Addition
& = Bitwise AND
5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

### TABLE 5-1: AC/DC CHARACTERISTICS

TIMING REQUIREMENTS FOR PROGRAM/VERIFY TEST MODE

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD1</td>
<td>VDDP</td>
<td>Supply voltage during programming</td>
<td>4.75</td>
<td>5.0</td>
<td>5.25</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PD2</td>
<td>IDDp</td>
<td>Supply current (from VDD) during programming</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD3</td>
<td>VDDV</td>
<td>Supply voltage during verify</td>
<td>VDDmin</td>
<td>-</td>
<td>VDDmax</td>
<td>V</td>
<td>Note 1</td>
</tr>
<tr>
<td>PD4</td>
<td>VHH1</td>
<td>Voltage on MCLR/VPP during programming</td>
<td>12.75</td>
<td>-</td>
<td>13.25</td>
<td>V</td>
<td>Note 2</td>
</tr>
<tr>
<td>PD5</td>
<td>VHH2</td>
<td>Voltage on MCLR/VPP during verify</td>
<td>VDD + 4.0</td>
<td>-</td>
<td>13.5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PD6</td>
<td>IPP</td>
<td>Programming supply current (from VPP)</td>
<td>-</td>
<td>-</td>
<td>50</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD9</td>
<td>Vih</td>
<td>(RB6, RB7) input high level</td>
<td>0.8 VDD</td>
<td>-</td>
<td>-</td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>PD8</td>
<td>VIL</td>
<td>(RB6, RB7) input low level</td>
<td>0.2 VDD</td>
<td>-</td>
<td>-</td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
</tbody>
</table>

#### Serial Program Verify

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>TR</td>
<td>MCLR/VPP rise time (VSS to VHH) for test mode entry</td>
<td>-</td>
<td>-</td>
<td>8.0</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>TF</td>
<td>MCLR Fall time</td>
<td>-</td>
<td>-</td>
<td>8.0</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>TSET1</td>
<td>Data in setup time before clock ↓</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>TH1</td>
<td>Data in hold time after clock ↓</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>TDL1</td>
<td>Data input not driven to next clock input (delay required between command/data or command/command)</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>TDL2</td>
<td>Delay between clock ↓ to clock ↑ of next command or data</td>
<td>1.0</td>
<td>-</td>
<td>-</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>TDL3</td>
<td>Clock ↑ to date out valid (during read data)</td>
<td>200</td>
<td>-</td>
<td>-</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>TH0</td>
<td>Hold time after MCLR ↑</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Tpw</td>
<td>Programming Pulse Width</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>µs</td>
<td></td>
</tr>
</tbody>
</table>

**Note 1:** Program must be verified at the minimum and maximum VDD limits for the part.

**Note 2:** VHH must be greater than VDD + 4.5V to stay in programming/verify mode.
FIGURE 5-1: LOAD DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-2: READ DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-3: INCREMENT ADDRESS COMMAND (PROGRAM/VERIFY)
1.0 PROGRAMMING THE PIC16C6XX/7XX/9XX

The PIC16C6XX/7XX/9XX can be programmed using a serial method. In serial mode the PIC16C6XX/7XX/9XX can be programmed while in the users system. This allows for increased design flexibility. This programming specification applies to PIC16C6XX/7XX/9XX devices in all packages.

1.1 Hardware Requirements

The PIC16C6XX/7XX/9XX requires two programmable power supplies, one for Vdd (2.0V to 6.5V recommended) and one for VPP (12V to 14V). Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC16C6XX/7XX/9XX allows programming of user program memory, special locations used for ID, and the configuration word for the PIC16C6XX/7XX/9XX.
Pin Diagrams (Con’t)
2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The user memory space extends from 0x0000 to 0xFFFF (8K). Table 2-1 shows actual implementation of program memory in the PIC16C6XX/7XX/9XX family.

**TABLE 2-1: IMPLEMENTATION OF PROGRAM MEMORY IN THE PIC16C6XX/7XX/9XX**

<table>
<thead>
<tr>
<th>Device</th>
<th>Program Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C61</td>
<td>0x000 – 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC16C62/620A</td>
<td>0x000 – 0x1FF (0.5K)</td>
</tr>
<tr>
<td>PIC16C621/621A</td>
<td>0x000 – 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC16C622/622A</td>
<td>0x000 – 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC16C62/62A/62B</td>
<td>0x000 – 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC16C63/63A</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
<tr>
<td>PIC16C64/64A</td>
<td>0x000 – 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC16C65/65A/65B</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
<tr>
<td>PIC16CE623</td>
<td>0x000 – 0x1FF (0.5K)</td>
</tr>
<tr>
<td>PIC16CE624</td>
<td>0x000 – 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC16CE625</td>
<td>0x000 – 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC16C71</td>
<td>0x000 – 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC16C710</td>
<td>0x000 – 0x1FF (0.5K)</td>
</tr>
<tr>
<td>PIC16C711</td>
<td>0x000 – 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC16C712</td>
<td>0x000 – 0x3FF (1K)</td>
</tr>
<tr>
<td>PIC16C716</td>
<td>0x000 – 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC16C72/72A</td>
<td>0x000 – 0x7FF (2K)</td>
</tr>
<tr>
<td>PIC16C73/73A/73B</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
<tr>
<td>PIC16C74/74A/74B</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
<tr>
<td>PIC16C66</td>
<td>0x000 – 0x1FFF (8K)</td>
</tr>
<tr>
<td>PIC16C67</td>
<td>0x000 – 0x1FFF (8K)</td>
</tr>
<tr>
<td>PIC16C76</td>
<td>0x000 – 0x1FFF (8K)</td>
</tr>
<tr>
<td>PIC16C77</td>
<td>0x000 – 0x1FFF (8K)</td>
</tr>
<tr>
<td>PIC16C745</td>
<td>0x000 – 0x1FFF (8K)</td>
</tr>
<tr>
<td>PIC16C765</td>
<td>0x000 – 0x1FFF (8K)</td>
</tr>
<tr>
<td>PIC16C773</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
<tr>
<td>PIC16C774</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
<tr>
<td>PIC16C923/924</td>
<td>0x000 – 0xFFF (4K)</td>
</tr>
</tbody>
</table>

When the PC reaches the last location of the implemented program memory, it will wrap around and address a location within the physically implemented memory (see Figure 2-1).

Once in configuration memory, the highest bit of the PC stays a ‘1’, thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode, as described in Section 2.2.

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000:0x2003]. It is recommended that the user use only the four least significant bits of each ID location. In some devices, the ID locations read-out in a scrambled fashion after code protection is enabled. For these devices, it is recommended that ID location is written as “11 1111 1bbb bbbb” where ‘bbbb’ is ID information.

Note: All other locations are reserved and should not be programmed.

In other devices, the ID locations read out normally, even after code protection. To understand how the devices behave, refer to Table 4-1.

To understand the scrambling mechanism after code protection, refer to Section 3.1.
FIGURE 2-1: PROGRAM MEMORY MAPPING

<table>
<thead>
<tr>
<th></th>
<th>ID Location</th>
<th>ID Location</th>
<th>ID Location</th>
<th>ID Location</th>
<th>ID Location</th>
<th>Configuration Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Reserved</td>
</tr>
<tr>
<td>2001h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2002h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2003h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2004h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2005h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2006h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2007h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>0.5K words</th>
<th>1K words</th>
<th>2K words</th>
<th>4K words</th>
<th>8K words</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000h-1FFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2001h-3FFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2002h-7FFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2003h-8FFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2004h-FFFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2005h-1000h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2006h-1FFFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2007h-3FFFh</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
</tbody>
</table>

| 2000h-2007h     | Reserved    | Reserved   | Reserved   | Reserved   | Reserved   |
| 2100h-3FFFh     | Reserved    | Reserved   | Reserved   | Reserved   | Reserved   |
2.2 Program/Verify Mode

The program/verify mode is entered by holding pins RB6 and RB7 low while raising MCLR pin from Vss to the appropriate VIH (high voltage). Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program memory. RB6 is a Schmitt Trigger input in this mode.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at Vss). This means that all I/O are in the reset state (High impedance inputs).

2.2.1 PROGRAM/VERIFY OPERATION

The RB6 pin is used as a clock input pin, and the RB7 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (RB6) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSb) of the command being input first. The data on pin RB7 is required to have a minimum setup and hold time (see AC/DC specs) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1 μs between the command and the data. After this delay the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSb first. Therefore, during a read operation the LSb will be transmitted onto pin RB7 on the rising edge of the second cycle, and during a load operation the LSb will be latched on the falling edge of the second cycle. A minimum 1 μs delay is also specified between consecutive commands.

All commands are transmitted LSb first. Data words are also transmitted LSb first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1 μs is required between a command and a data word (or another command).

The commands that are available are listed in Table 2-2.

2.2.1.1 LOAD CONFIGURATION

After receiving this command, the program counter (PC) will be set to 0x2000. By then applying 16 cycles to the clock pin, the chip will load 14-bits a “data word” as described above, to be programmed into the configuration memory. A description of the memory mapping schemes for normal operation and configuration mode operation is shown in Figure 2-1. After the configuration memory is entered, the only way to get back to the user program memory is to exit the program/verify test mode by taking MCLR low (VIH).

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSb... LSb)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>0 0 0 0 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Load Data</td>
<td>0 0 0 0 1 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Read Data</td>
<td>0 0 0 1 0 0</td>
<td>0, data(14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>0 0 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>Begin programming</td>
<td>0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>End Programming</td>
<td>0 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Note: The MCLR pin should be raised as quickly as possible from VIH to VIH, this is to ensure that the device does not have the PC incremented while in valid operation range.

2: Do not power any pin before VDD is applied.

Note 2: The clock must be disabled during In-Circuit Serial Programming.
FIGURE 2-2: PROGRAM FLOW CHART - PIC16C6XX/7XX/9XX PROGRAM MEMORY

Start

Set VDD = VDDP*

Set VPP = VIHH1

N = 1

Program Cycle

Read Data Command

Data correct?

Yes

Apply 3N Additional Program Cycles

All locations done?

Yes

Verify all locations @ VDD min.*
VPP = VIHH2

Data correct?

No

Report verify @ VDD min. Error

Yes

Verify all locations @ VDD max.*
VPP = VIHH2

Data correct?

No

Report verify @ VDD max. Error

Yes

Done

N > 25?

No

Yes

Report programming failure

N = N + 1

Increment Address Command

Program Cycle

Load Data Command

Begin Programming Command

Wait 100 µs

End Programming Command

* VDDP = VDD range for programming (typically 4.75V - 5.25V).
VDDmin = Minimum VDD for device operation.
VDDmax = Maximum VDD for device operation.
FIGURE 2-3: PROGRAM FLOW CHART - PIC16C6XX/7XX/9XX CONFIGURATION WORD & ID LOCATIONS

Start

Set VDD = VDD^*

Set VPP = VHH1

Load Configuration Command

N = 1

Program ID Loc?

Yes

Program Cycle

N = N + 1 N = # of Program Cycles

No

Data Correct?

Yes

Apply 3N Program Cycles

No

Data Correct?

Yes

Report ID Configuration Error

No

Increment Address Command

Yes

Address = 2004

Increment Address Command

Increment Address Command

Increment Address Command

Increment Address Command

Program Cycle 100 Cycles

Read Data Command

No

Data Correct?

Yes

Set VDD = VDDmin

Set VPP = VHH2

Read Data Command

No

Data Correct?

Yes

Report Program ID/Config. Error

No

Set VDD = VDDmax

Set VPP = VHH2

Read Data Command

Yes

Data Correct?

Done

VDD^* = VDD Range for programming (Typically 4.25V – 5.25V)

VDDMIN = minimum VDD for device operation

VDDMAX = maximum VDD for device operation
2.2.1.2 LOAD DATA

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 4-1.

2.2.1.3 READ DATA

After receiving this command, the chip will transmit data bits out of the memory currently accessed starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 4-2.

2.2.1.4 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 4-3.

2.2.1.5 BEGIN PROGRAMMING

A load command (load configuration or load data) must be given before every begin programming command. Programming of the appropriate memory (test program memory or user program memory) will begin after this command is received and decoded. Programming should be performed with a series of 100µs programming pulses. A programming pulse is defined as the time between the begin programming command and the end programming command.

2.2.1.6 END PROGRAMMING

After receiving this command, the chip stops programming the memory (configuration program memory or user program memory) that it was programming at the time.

2.3 Programming Algorithm Requires Variable VDD

The PIC16C6XX/7XX/9XX uses an intelligent algorithm. The algorithm calls for program verification at VDDmin as well as VDDmax. Verification at VDDmin guarantees good “erase margin”. Verification at VDDmax guarantees good “program margin”.

The actual programming must be done with VDD in the VDDP range (4.75 - 5.25V).

VDDP = VCC range required during programming.

VDD min. = minimum operating VDD spec for the part.

VDD max. = maximum operating VDD spec for the part.

Programmers must verify the PIC16C6XX/7XX/9XX at its specified VDDmax and VDDmin levels. Since Microchip may introduce future versions of the PIC16C6XX/7XX/9XX with a broader VDD range, it is best that these levels are user selectable (defaults are ok).

Note: Any programmer not meeting these requirements may only be classified as “prototype” or “development” programmer but not a “production” quality programmer.
3.0 CONFIGURATION WORD
The PIC16C6XX/7XX/9XX family members have several configuration bits. These bits can be programmed (reads '0') or left unprogrammed (reads '1') to select various device configurations. Figure 3-1 and Figure 3-2 provides an overview of configuration bits.
# FIGURE 3-1: CONFIGURATION WORD BIT MAP

<table>
<thead>
<tr>
<th>Bit Number:</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C61/71</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>CP0</td>
</tr>
<tr>
<td>PIC16C62/64/65/73/74</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>CP0</td>
</tr>
</tbody>
</table>

| PIC16C64A/64A/65A/65B/66/67/72/72A/73A/73B/74A/74B/76/77 | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |

| PIC16C65A/65A/65B | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |
| PIC16C66/67/72/72A | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |
| PIC16C73/73A/73B | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |
| PIC16C74A/74B/74B/76/77 | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |
| PIC16C745/765 | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |
| PIC16C9XX | —  | —  | —  | —  | — | — | — | — | — | — | — | — | — | CP0 |

<table>
<thead>
<tr>
<th>Device</th>
<th>CP1</th>
<th>CP0</th>
<th>Code Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C622/622A</td>
<td>0</td>
<td>0</td>
<td>All memory protected</td>
</tr>
<tr>
<td>PIC16C62/62A/62B</td>
<td>0</td>
<td>1</td>
<td>Upper 3/4 memory protected</td>
</tr>
<tr>
<td>PIC16C63/63A</td>
<td>1</td>
<td>0</td>
<td>Upper 1/2 memory protected</td>
</tr>
<tr>
<td>PIC16C64/64A/712/716</td>
<td>1</td>
<td>1</td>
<td>Code protection off</td>
</tr>
<tr>
<td>PIC16C65/65A/65B</td>
<td>0</td>
<td>0</td>
<td>All memory protected</td>
</tr>
<tr>
<td>PIC16C66/67/72/72A</td>
<td>0</td>
<td>1</td>
<td>Do not use</td>
</tr>
<tr>
<td>PIC16C73/73A/73B</td>
<td>1</td>
<td>0</td>
<td>Do not use</td>
</tr>
<tr>
<td>PIC16C74/74A/74B/76/77</td>
<td>1</td>
<td>1</td>
<td>Code protection off</td>
</tr>
<tr>
<td>PIC16C745/765</td>
<td>0</td>
<td>1</td>
<td>All memory protected</td>
</tr>
<tr>
<td>PIC16C9XX</td>
<td>1</td>
<td>1</td>
<td>Upper 1/2 memory protected</td>
</tr>
</tbody>
</table>

**Note 1:** Enabling Brown-out Reset automatically enables the Power-up Timer (PWRT) regardless of the value of bit PWRT. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.
FIGURE 3-2: CONFIGURATION WORD FOR PIC16C773/774 DEVICE

<table>
<thead>
<tr>
<th>CP1</th>
<th>CP0</th>
<th>BORV1</th>
<th>BORV0</th>
<th>CP1</th>
<th>CP0</th>
<th>-</th>
<th>BODEN</th>
<th>CP1</th>
<th>CP0</th>
<th>PWRT</th>
<th>WDTE</th>
<th>FOSC1</th>
<th>FOSC0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>bit0</td>
</tr>
</tbody>
</table>

Register: CONFIG
Address: 2007h

**CP <1:0>: Code Protection bits (2)**

<table>
<thead>
<tr>
<th>Device</th>
<th>CP1</th>
<th>CP0</th>
<th>Code Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C773/774</td>
<td>0</td>
<td>0</td>
<td>All memory protected</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>Upper 3/4 memory protected</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Upper 1/2 memory protected</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>Code protection off</td>
</tr>
</tbody>
</table>

bit 11-10: **BORV <1:0>:** Brown-out Reset Voltage bits
11 = V<sub>BOR</sub> set to 2.5V
10 = V<sub>BOR</sub> set to 2.7V
01 = V<sub>BOR</sub> set to 4.2V
00 = V<sub>BOR</sub> set to 4.5V

bit 7: **Unimplemented,** Read as '1'

bit 6: **BODEN:** Brown-out Reset Enable bit (1)
1 = Brown-out Reset enabled
0 = Brown-out Reset disabled

bit 3: **PWRT:** Power-up Timer Enable bit (1)
1 = PWRT disabled
0 = PWRT enabled

bit 2: **WDTE:** Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0: **FOSC <1:0>:** Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

Note 1: Enabling Brown-out Reset automatically enables the Power-up Timer (PWRT) regardless of the value of bit PWRT. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

2: All of the CP <1:0> pairs have to be given the same value to enable the code protection scheme listed.
FIGURE 3-3: CONFIGURATION WORD, PIC16C710/711

<table>
<thead>
<tr>
<th>CP0</th>
<th>CP0</th>
<th>CP0</th>
<th>CP0</th>
<th>CP0</th>
<th>CP0</th>
<th>BODEN</th>
<th>CP0</th>
<th>CP0</th>
<th>PWRTE</th>
<th>WDTE</th>
<th>FOSC1</th>
<th>FOSC0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Register: CONFIG
Address 2007h

bit 13-7 **CP0**: Code protection bits (2)

5-4: 1 = Code protection off
      0 = All memory is code protected, but 00h - 3Fh is writable

bit 6: **BODEN**: Brown-out Reset Enable bit (1)

1 = BOR enabled
0 = BOR disabled

bit 3: **PWRTE**: Power-up Timer Enable bit (1)

1 = PWRT disabled
0 = PWRT enabled

bit 2: **WDTE**: Watchdog Timer Enable bit

1 = WDT enabled
0 = WDT disabled

bit 1-0: **FOSC <1:0>**: Oscillator Selection bits

11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

Note 1: Enabling Brown-out Reset automatically enables Power-up Timer (PWRT) regardless of the value of bit PWRTE.
      Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

2: All of the CP0 bits have to be given the same value to enable the code protection scheme listed.
3.1 **Embedding Configuration Word and ID Information in the Hex File.**

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Microchip Technology Inc. feels strongly that this feature is beneficial to the end customer.
3.2 Checksum

3.2.1 CHECKSUM CALCULATIONS

Checksum is calculated by reading the contents of the PIC16C6XX/7XX/9XX memory locations and adding up the opcodes up to the maximum user addressable location, e.g., 0x1FF for the PIC16C74. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC16C6XX/7XX/9XX devices is shown in Table 3-1.

The checksum is calculated by summing the following:

- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

---

TABLE 3-1: CHECKSUM COMPUTATION

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum* Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C61</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x001F + 0x3FE0</td>
<td>0x3BFF</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM_XNOR7[0x000:0x3FF] + (CFGW &amp; 0x001F</td>
<td>0xFC6F</td>
</tr>
<tr>
<td>PIC16C620</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM_ID + CFGW &amp; 0x3F7F</td>
<td>0x099C</td>
</tr>
<tr>
<td>PIC16C620A</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM_ID + CFGW &amp; 0x3F7F</td>
<td>0x099C</td>
</tr>
<tr>
<td>PIC16C621</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td>1/2</td>
<td>ALL</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x094D</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x099C</td>
</tr>
<tr>
<td>PIC16C621A</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td>1/2</td>
<td>ALL</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x094D</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x099C</td>
</tr>
<tr>
<td>PIC16C622</td>
<td>OFF</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td>1/2</td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x094D</td>
</tr>
<tr>
<td>3/4</td>
<td>ALL</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x099C</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x099C</td>
</tr>
<tr>
<td>PIC16C622A</td>
<td>OFF</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td>1/2</td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x094D</td>
</tr>
<tr>
<td>3/4</td>
<td>ALL</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x099C</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x099C</td>
</tr>
<tr>
<td>PIC16C623</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F</td>
<td>0x3D7F</td>
</tr>
<tr>
<td>ON</td>
<td>ALL</td>
<td>SUM_ID + CFGW &amp; 0x3F7F</td>
<td>0x094D</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word
SUM[a:b] = [Sum of locations a through b inclusive]
SUM_XNOR7[a:b] = XOR of the seven high order bits of memory location with the seven low order bits summed over locations a through b inclusive. For example, XOR(0xF3C1) = 0x078 XOR 0x031 = 0x0036.
SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.

*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
+ = Addition
& = Bitwise AND
| = Bitwise OR
<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value 0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16CE624</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F</td>
<td>0x3B7F 0x074D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4EDE 0x0093</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3BCE 0x079C</td>
</tr>
<tr>
<td>PIC16CE625</td>
<td>OFF</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x377F 0x034D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + SUM_XNOR7[0x400:0x7FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x0F3A</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3AF9 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x379F 0x1059</td>
</tr>
<tr>
<td>PIC16C62</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x377F 0x034D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37AF 0x1069</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + SUM_XNOR7[0x200:0x7FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x379F 0x1059</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>SUM_XNOR7[0x000:0x7FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x378F 0x3735</td>
</tr>
<tr>
<td>PIC16C62A</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x377F 0x034D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + SUM_XNOR7[0x400:0x7FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x0F3A</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
<tr>
<td>PIC16C62B</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x377F 0x034D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x1FF] + SUM_XNOR7[0x400:0x7FF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x0F3A</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
<tr>
<td>PIC16C63</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x378F 0x3735</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + SUM_XNOR7[0x800:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x1069</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
<tr>
<td>PIC16C63A</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x378F 0x3735</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + SUM_XNOR7[0x800:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x1069</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
<tr>
<td>PIC16C64</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x378F 0x3735</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + SUM_XNOR7[0x800:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x1069</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
<tr>
<td>PIC16C64A</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x378F 0x3735</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + SUM_XNOR7[0x800:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x1069</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
<tr>
<td>PIC16C65</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x378F 0x3735</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + SUM_XNOR7[0x800:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x5DEE 0x1069</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x4ADE 0x1069</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x37CE 0x039C</td>
</tr>
</tbody>
</table>

Legend:  
- CFGW = Configuration Word  
- SUM[ab] = [Sum of locations a through b inclusive]  
- SUM_XNOR7[ab] = XNOR of the seven high order bits of memory location with the seven low order bits summed over locations a through b inclusive. For example, XNOR(0x3C31)=0x78 XNOR 0c31 = 0x0036.  
- SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.  
- *Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]  
  + = Addition  
  & = Bitwise AND  
  | = Bitwise OR
### TABLE 3-1: CHECKSUM COMPUTATION (CONTINUED)

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C65A</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFD4B</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xFD9C</td>
</tr>
<tr>
<td>PIC16C65B</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFD4B</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xFD9C</td>
</tr>
<tr>
<td>PIC16C66</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xFD9C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x1F7F</td>
<td>0xEB9C</td>
</tr>
<tr>
<td>PIC16C67</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x7FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xFD9C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x1F7F</td>
<td>0xEB9C</td>
</tr>
<tr>
<td>PIC16C710</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUM[0x000:0x07FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td>PIC16C711</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SUM[0x000:0x07FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td>PIC16C712</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td>PIC16C716</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x01FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td>PIC16C72</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td>PIC16C72A</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td>PIC16C73</td>
<td>OFF</td>
<td>SUM[0x000:0x1FFF] + CFGW &amp; 0x3F7F</td>
<td>0x51EE</td>
<td>0xA3D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3E0E</td>
<td>0xEF3C</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word

SUM[ab] = [Sum of locations a through b inclusive]

SUM_XNOR7[ab] = XNOR of the seven high order bits of memory location with the seven low order bits summed over locations a through b inclusive. For example, XNOR(0x3C31)=0x78 XNOR 0c31 = 0x0036.

SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.

*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]

+ = Addition
& = Bitwise AND
I = Bitwise OR
### TABLE 3-1: CHECKSUM COMPUTATION (CONTINUED)

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C73B</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xF84D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0xF3A3</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xF89C</td>
</tr>
<tr>
<td>PIC16C74</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x2F8F</td>
<td>0xF88D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + SUM_XNOR7[0x800:FFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x2FAF</td>
<td>0xF159</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x2F9F</td>
<td>0xF155</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>SUM_XNOR7[0x000:0xFFF] + CFGW &amp; 0x003F + 0x3F80</td>
<td>0x2F8F</td>
<td>0xF235</td>
</tr>
<tr>
<td>PIC16C74A</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xF84D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0xF3A3</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xF89C</td>
</tr>
<tr>
<td>PIC16C74B</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xF84D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0xF3A3</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF293</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2FCE</td>
<td>0xF89C</td>
</tr>
<tr>
<td>PIC16C76</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x1F7F</td>
<td>0xEB4D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x39EE</td>
<td>0xEBA3</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2CDE</td>
<td>0xDE93</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x1FCE</td>
<td>0xEB9C</td>
</tr>
<tr>
<td>PIC16C77</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFB4D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x55EE</td>
<td>0xFB13</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x48DE</td>
<td>0xFA93</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3BCE</td>
<td>0x709C</td>
</tr>
<tr>
<td>PIC16C773</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFB4D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x55EE</td>
<td>0xFB13</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x48DE</td>
<td>0xFA93</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3BCE</td>
<td>0x709C</td>
</tr>
<tr>
<td>PIC16C774</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFB4D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x55EE</td>
<td>0xFB13</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x48DE</td>
<td>0xFA93</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x3BCE</td>
<td>0x709C</td>
</tr>
<tr>
<td>PIC16C923</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFB4D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0x3032</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF213</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2F4E</td>
<td>0xFB1C</td>
</tr>
<tr>
<td>PIC16C924</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFB4D</td>
</tr>
<tr>
<td></td>
<td>1/2</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x51EE</td>
<td>0x3032</td>
</tr>
<tr>
<td></td>
<td>3/4</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x40DE</td>
<td>0xF213</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>0x2F4E</td>
<td>0xFB1C</td>
</tr>
<tr>
<td>PIC16C745</td>
<td>OFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F</td>
<td>0x2F7F</td>
<td>0xFB4D</td>
</tr>
<tr>
<td></td>
<td>1000:1FFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>1F3F</td>
<td>EB0D</td>
</tr>
<tr>
<td></td>
<td>800:1FFF</td>
<td>SUM[0x000:0xFFF] + CFGW &amp; 0x3F7F + SUM_ID</td>
<td>39E</td>
<td>EB23</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3F7F + SUM_ID</td>
<td>1F4E</td>
<td>EB1C</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word
SUM[a:b] = [Sum of locations a through b inclusive]
SUM_XNOR7[a:b] = XNOR of the seven high order bits of memory location with the seven low order bits summed over locations a through b inclusive. For example, XNOR(0x3C31)=0x78 XNOR 0x31 = 0x0036.
SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM_ID = 0x2746.

*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
+ = Addition
& = Bitwise AND
| = Bitwise OR
<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16C765</td>
<td>OFF</td>
<td>$\text{SUM}(0000:1FFF) + \text{CFGW} &amp; 0x3F3F$</td>
<td>1F3F</td>
<td>EB0D</td>
</tr>
<tr>
<td>1000:1FF</td>
<td>SUM(0000:0FFF) + \text{CFGW} &amp; 0x3F3F + \text{SUM}_\text{ID}</td>
<td>396E</td>
<td>EB23</td>
<td></td>
</tr>
<tr>
<td>800:1FF</td>
<td>SUM(0000:07FF) + \text{CFGW} &amp; 0x3F3F + \text{SUM}_\text{ID}</td>
<td>2C5E</td>
<td>DE13</td>
<td></td>
</tr>
<tr>
<td>ALL</td>
<td>\text{CFGW} * 0x3F3F + \text{SUM}_\text{ID}</td>
<td>1F4E</td>
<td>EB1C</td>
<td></td>
</tr>
</tbody>
</table>

Legend: 
\text{CFGW} = Configuration Word  
SUM[a:b] = [Sum of locations a through b inclusive]  
SUM\_XNOR7[a:b] = XNOR of the seven high order bits of memory location with the seven low order bits summed over locations a through b inclusive. For example, XNOR(0x3C31)=0x78 \ XOR \ 0c31 = 0x0036.  
SUM\_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x12, ID1 = 0x37, ID2 = 0x4, ID3 = 0x26, then SUM\_ID = 0x2746.  
*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]  
+ = Addition  
\& = Bitwise AND  
| = Bitwise OR
# 4.0 PROGRAM/VERIFY MODE

## TABLE 4-1: AC/DC CHARACTERISTICS

### TIMING REQUIREMENTS FOR PROGRAM/VERIFY TEST MODE

**Standard Operating Conditions**

Operating Temperature: \( +10^\circ C \leq T_A \leq +40^\circ C \), unless otherwise stated, \((20^\circ C \text{ recommended})\)

Operating Voltage: \( 4.5V \leq V_{DD} \leq 5.5V \), unless otherwise stated.

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD1</td>
<td>VDDP</td>
<td>Supply voltage during programming</td>
<td>4.75</td>
<td>5.0</td>
<td>5.25</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PD2</td>
<td>IDD</td>
<td>Supply current (from VDD) during programming</td>
<td>–</td>
<td>–</td>
<td>20</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD3</td>
<td>VDDV</td>
<td>Supply voltage during verify</td>
<td>(V_{DD}\text{min} )</td>
<td>–</td>
<td>(V_{DD}\text{max} )</td>
<td>V</td>
<td>Note 1</td>
</tr>
<tr>
<td>PD4</td>
<td>V_IHH1</td>
<td>Voltage on MCLR/VPP during programming</td>
<td>12.75</td>
<td>–</td>
<td>13.25</td>
<td>V</td>
<td>Note 2</td>
</tr>
<tr>
<td>PD5</td>
<td>V_IHH2</td>
<td>Voltage on MCLR/VPP during verify</td>
<td>(V_{DD} + 4.5)</td>
<td>–</td>
<td>13.25</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>PD6</td>
<td>IPP</td>
<td>Programming supply current (from VPP)</td>
<td>–</td>
<td>–</td>
<td>50</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>PD9</td>
<td>V_H</td>
<td>(RB6, RB7) input high level</td>
<td>0.8 (V_{DD})</td>
<td>–</td>
<td>–</td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>PD8</td>
<td>V_L</td>
<td>(RB6, RB7) input low level</td>
<td>0.2 (V_{DD})</td>
<td>–</td>
<td>–</td>
<td>V</td>
<td>Schmitt Trigger input</td>
</tr>
</tbody>
</table>

### Serial Program Verify

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sym.</th>
<th>Characteristic</th>
<th>Min.</th>
<th>Typ.</th>
<th>Max.</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>TR</td>
<td>MCLR/VPP rise time (VSS to VHH) for test mode entry</td>
<td>–</td>
<td>–</td>
<td>8.0</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>Tf</td>
<td>MCLR Fall time</td>
<td>–</td>
<td>–</td>
<td>8.0</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>Tset1</td>
<td>Data in setup time before clock ↓</td>
<td>100</td>
<td>–</td>
<td>–</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>Thld1</td>
<td>Data in hold time after clock ↓</td>
<td>100</td>
<td>–</td>
<td>–</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>Tdy1</td>
<td>Data input not driven to next clock input (delay required between command/data or command/command)</td>
<td>1.0</td>
<td>–</td>
<td>–</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>Tdy2</td>
<td>Delay between clock ↓ to clock ↑ of next command or data</td>
<td>1.0</td>
<td>–</td>
<td>–</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>Tdy3</td>
<td>Clock ↑ to date out valid (during read data)</td>
<td>200</td>
<td>–</td>
<td>–</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>Thld0</td>
<td>Hold time after MCLR ↑</td>
<td>2</td>
<td>–</td>
<td>–</td>
<td>µs</td>
<td></td>
</tr>
</tbody>
</table>

**Note 1:** Program must be verified at the minimum and maximum VDD limits for the part.

**2:** \(V_IHH\) must be greater than \(V_{DD} + 4.5V\) to stay in programming/verify mode.
FIGURE 4-1: LOAD DATA COMMAND (PROGRAM/VERIFY)

FIGURE 4-2: READ DATA COMMAND (PROGRAM/VERIFY)

FIGURE 4-3: INCREMENT ADDRESS COMMAND (PROGRAM/VERIFY)
1.0 PROGRAMMING THE PIC17C7XX

The PIC17C7XX is programmed using the TABLWT instruction. The table pointer points to the internal EPROM location start. Therefore, a user can program an EPROM location while executing code (even from internal EPROM). This programming specification applies to PIC17C7XX devices in all packages.

For the convenience of a programmer developer, a “program & verify” routine is provided in the on-chip test program memory space. The program resides in ROM and not EPROM, therefore, it is not erasable. The “program/verify” routine allows the user to load any address, program a location, verify a location or increment to the next location. It allows variable programming pulse width.

The PIC17C7XX group of the High End Family has added a feature that allows the serial programming of the device. This is very useful in applications where it is desirable to program the device after it has been manufactured into the users system (In-circuit Serial Programming (ISP)). This allows the product to be shipped with the most current version of the firmware, since the microcontroller can be programmed just before final test as opposed to before board manufacture. Devices may be serialized to make the product unique, “special” variants of the product may be offered, and code updates are possible. This allows for increased design flexibility.

1.1 Hardware Requirements

Since the PIC17C7XX under programming is actually executing code from “boot ROM,” a clock must be provided to the part. Furthermore, the PIC17C7XX under programming may have any oscillator configuration (EC, XT, LF or RC). Therefore, the external clock driver must be able to overdrive pulldown in RC mode. CMOS drivers are required since the OSC1 input has a Schmitt trigger input with levels (typically) of 0.2 VDD and 0.8 VDD. See the PIC17C7XX data sheet (DS30289) for exact specifications.

The PIC17C7XX requires two programmable power supplies, one for VDD (3.0V to 5.5V recommended) and one for VPP (13 ± 0.25V). Both supplies should have a minimum resolution of 0.25V.

The PIC17C7XX uses an intelligent algorithm. The algorithm calls for program verification at VDDmin as well as VDDmax. Verification at VDDmin guarantees good “erase margin”. Verification at VDDmax guarantees good “program margin.” Three times (3X) additional pulses will increase program margin beyond VDDmax and insure safe operation in user system.

The actual programming must be done with VDD in the VDDP range (Parameter PD1).

VDDP = VDD range required during programming.
VDDmin. = minimum operating VDD spec. for the part.
VDDmax. = maximum operating VCC spec for the part.

Programmers must verify the PIC17C7XX at its specified VDDmax and VDDmin levels. Since Microchip may introduce future versions of the PIC17C7XX with a broader VDD range, it is best that these levels are user selectable (defaults are ok). Blank checks should be performed at VDDMIN.

Note: Any programmer not meeting these requirements may only be classified as “prototype” or “development” programmer but not a “production” quality programmer.
TABLE 1-1: PIN DESCRIPTIONS (DURING PROGRAMMING IN PARALLEL MODE): PIC17C7XX

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Pin Name</th>
<th>Pin Type</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA4:RA0</td>
<td>RA4:RA0</td>
<td>I</td>
<td>Necessary in programming mode</td>
</tr>
<tr>
<td>TEST</td>
<td>TEST</td>
<td>I</td>
<td>Must be set to “high” to enter programming mode</td>
</tr>
<tr>
<td>PORTB&lt;7:0&gt;</td>
<td>DAD15:DAD8</td>
<td>I/O</td>
<td>Address &amp; data: high byte</td>
</tr>
<tr>
<td>PORTC&lt;7:0&gt;</td>
<td>DAD7:DAD0</td>
<td>I/O</td>
<td>Address &amp; data: low byte</td>
</tr>
<tr>
<td>MCLR/VPP</td>
<td>VPP</td>
<td>P</td>
<td>Programming Power</td>
</tr>
<tr>
<td>VDD</td>
<td>VDD</td>
<td>P</td>
<td>Power Supply</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>P</td>
<td>Ground</td>
</tr>
</tbody>
</table>

Legend: I = Input, O = Output, P = Power
2.0 PARALLEL MODE PROGRAM ENTRY

To execute the programming routine, the user must hold TEST pin high, RA2, RA3 must be low and RA4 must be high (after power-up) while keeping MCLR low and then raise MCLR pin from VIL to VDD or VPP. This will force FFE0h in the program counter and execution will begin at that location (the beginning of the boot code) following reset.

Note: The Oscillator must not have 72 OSC clocks while the device MCLR is between VIL and VIHH.

All unused pins during programming are in hi-impedance state.

PORTB (RB pins) has internal weak pull-ups which are active during the programming mode. When the TEST pin is high, the Power-up timer (PWRT) and Oscillator Start-up Timers (OST) are disabled.

2.1 Program/Verify Mode

The program/verify mode is intended for full-feature programmers. This mode offers the following capabilities:

- Load any arbitrary 16-bit address to start program and/or verify at that location.
- Increment address to program/verify the next location.
- Allows arbitrary length programming pulse width.
- Following a “verify” allows option to program the same location or increment and verify the next location.
- Following a “program” allows options to program the same location again, verify the same location or to increment and verify the next location.

FIGURE 2-1: PROGRAMMING/VERIFY STATE DIAGRAM
2.1.1 LOADING NEW ADDRESS

The program allows new address to be loaded right out of reset. A 16-bit address is presented on ports B (high byte) and C (low byte) and the RA1 is pulsed (0 → 1, then 1 → 0). The address is latched on the rising edge of RA1. See timing diagrams for details. After loading an address, the program automatically goes into a “verify cycle.” To load a new address at any time, the PIC17C7XX must be reset and the programming mode re-entered.

2.1.2 VERIFY (OR READ) MODE

“Verify mode” can be entered from “Load address” mode, “program mode” or “verify mode.” In verify mode pulsing RA1 will turn on PORTB and PORTC output drivers and output the 16-bit value from the current location. Pulsing RA1 again will increment location count and be ready for the next verify cycle. Pulsing RA0 will begin a program cycle.

2.1.3 PROGRAM CYCLE

“Program cycle” is entered from “verify cycle” or program cycle itself. After a verify, pulsing RA0 will begin a program cycle. 16-bit data must be presented on PORTB (high byte) and PORTC (low byte) before RA0 is raised.

The data is sampled 3 TCY cycles after the rising edge of RA0, Programming continues for the duration of RA0 pulse.

At the end of programming, the user can choose one of three different routes. If RA1 is kept low and RA0 is pulsed again, the same location will be programmed again. This is useful for applying over programming pulses. If RA1 is raised before RA0 falling edge, then a verify cycle is started without address increment. Raising RA1 after RA0 goes low will increment address and begin verify cycle on the next address.

FIGURE 2-2: PIC17C7XX PROGRAM MEMORY MAP
3.0 PARALLEL MODE PROGRAMMING SPECIFICATIONS

FIGURE 3-1: PROGRAMMING ROUTINE FLOWCHART

- RA2 = 0
- RA3 = 0
- RA4 = 1

- MCLR = 1
- Bport = 0xE1 (hold for 10 Tcy)

- Present address on ports RB, RC
  hold TCY after RA1 changes to 1

- RA1 = 0
- RA1 = 1

- Stop driving address on ports

- B port = MSB of Data
  C port = LSB of Data

- B port tristate, should be forced by user

- Min RA + high or low = 10 Tcy
FIGURE 3-2: RECOMMENDED PROGRAMMING ALGORITHM FOR USER EPROM

- Start
  - Load new address
    - Pulse-count = 0
  - Set VDD = VDDMIN
  - Verify blank
    - Pulse Blank Check?
      - YES
      - Load new data
      - Set VDD to VDDP
      - Program using 100 µs pulse increment pulse-count
      - Verify location for correct date
        - Pass?
          - YES
          - Set VDD = VDDMIN
            - Program error message
            - Issue error message
            - “Fail verify @ VDDMIN/MAX”
          - NO
          - NO
          - NO
          - NO
          - NO
          - NO
          - NO
          - NO
          - Location fails programming issue error message “Unable to programming location”
      - NO
        - Pulse-Count >25
          - YES
          - Location fails programming issue error message “Unable to programming location”
      - Issue “Blank check fail” error message
  - NO

- Pass?
  - YES
  - Set VDD = VDDMIN
    - verify location(s)
  - NO
  - NO
  - NO
  - NO
  - NO
  - NO
  - NO
  - NO
  - NO
  - NO
  - NO
FIGURE 3-3: RECOMMENDED PROGRAMMING ALGORITHM FOR CONFIGURATION WORDS

Start

Load new address
Pulse-count = 0

Set VDD = VDDmin

Verify blank

Pass? Blank check?

NO Issue “blank check fail” error message

YES

Load new data

Set VDD = VDDP

Set VDD = VDDmin

Program using 100 μs pulse increment pulse-count

Pulse count <100

YES

Verify location for correct data

Pass?

YES

NO Location fails programming, issue error message “Unable to program location”

Pass?

YES

Set VDD = VDDMIN

Set VDD = VDDMIN

Verifying location

NO

YES Pulse count <100

Programming error: Issue error message “Fail verify @ VDDmin/max”

Set VDD = VDDmax

Verify location(s)

Pass?

YES

NO
4.0 SERIAL MODE PROGRAM ENTRY

4.1 Hardware Requirements

Certain design criteria must be taken into account for ISP. Seven pins are required for the interface. These are shown in Table 4-1.

4.2 Serial Program Mode Entry

To place the device into the serial programming test mode, two pins will need to be placed at VIHH. These are the TEST pin and the MCLR/VPP pins. Also, the following sequence of events must occur:

1. The TEST pin is placed at VIHH.
2. The MCLR/VPP pin is placed at VIHH.

There is a setup time between step 1 and step 2 that must be meet (See “Electrical Specifications for Serial Programming Mode” on page 93.)

After this sequence the Program Counter is pointing to Program Memory Address 0xFF60. This location is in the Boot ROM. The code initializes the USART/SCI so that it can receive commands. For this the device must be clocked. The device clock source in this mode is the RA1/T0CKI pin. Once the USART/SCI has been initialized, commands may be received. The flow is show in these 3 steps:

1. The device clock source starts.
2. Wait 80 device clocks for Boot ROM code to configure the USART/SCI.
3. Commands may be sent now.

TABLE 4-1: ISP Interface Pins

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA4/RX/DT</td>
<td>DT</td>
<td>I/O</td>
<td>Serial Data</td>
</tr>
<tr>
<td>RA5/TX/CK</td>
<td>CK</td>
<td>I</td>
<td>Serial Clock</td>
</tr>
<tr>
<td>RA1/T0CKI</td>
<td>OSCI</td>
<td>I</td>
<td>Device Clock Source</td>
</tr>
<tr>
<td>TEST</td>
<td>TEST</td>
<td>I</td>
<td>Test mode selection control input. Force to VIHH,</td>
</tr>
<tr>
<td>MCLR/VPP</td>
<td>MCLR/VPP</td>
<td>P</td>
<td>Programming Power</td>
</tr>
<tr>
<td>VDD</td>
<td>VDD</td>
<td>P</td>
<td>Power Supply</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>P</td>
<td>Ground</td>
</tr>
</tbody>
</table>
4.3 Software Commands

This feature is similar to that of the PIC16CXXX mid-range family, but the programming commands have been implemented in the device Boot ROM. The Boot ROM is located in the program memory from 0xFF60 to 0xFFFF. The ISP mode is entered when the TEST pin has a VIHH voltage applied. Once in ISP mode, the USART/SCI module is configured as a synchronous slave receiver, and the device waits for a command to be received. The ISP firmware recognizes eight commands. These are shown in Table 4-2.

### TABLE 4-2: ISP COMMANDS

<table>
<thead>
<tr>
<th>Command</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET PROGRAM MEMORY POINTER</td>
<td>0000 0000</td>
</tr>
<tr>
<td>LOAD DATA</td>
<td>0000 0010</td>
</tr>
<tr>
<td>READ DATA</td>
<td>0000 0100</td>
</tr>
<tr>
<td>INCREMENT ADDRESS</td>
<td>0000 0110</td>
</tr>
<tr>
<td>BEGIN PROGRAMMING</td>
<td>0000 1000</td>
</tr>
<tr>
<td>LOAD ADDRESS</td>
<td>0000 1010</td>
</tr>
<tr>
<td>READ ADDRESS</td>
<td>0000 1100</td>
</tr>
<tr>
<td>END PROGRAMMING</td>
<td>0000 1110</td>
</tr>
</tbody>
</table>

4.3.1 RESET PROGRAM MEMORY POINTER

This is used to clear the address pointer to the Program Memory. This ensures that the pointer is at a known state as well as pointing to the first location in program memory.

4.3.2 INCREMENT ADDRESS

This is used to increment the address pointer to the Program Memory. This is used after the current location has been programmed (or read).
4.3.3  LOAD ADDRESS

This is used to load the address pointer to the Program Memory with a specific 16-bit value. This is useful when a specific range of locations are to be accessed.

4.3.4  READ ADDRESS

This is used so that the current address in the Program Memory pointer can be determined. This can be used to increase the robustness of the ISP programming (ensure that the Program Memory pointers are still in sync).

FIGURE 4-3: LOAD ADDRESS COMMAND

FIGURE 4-4: READ ADDRESS COMMAND
4.3.5 LOAD DATA

This is used to load the 16-bit data that is to be programmed into the Program Memory location. The Program Memory address may be modified after the data is loaded. This data will not be programmed until a BEGIN PROGRAMMING command is executed.

4.3.6 READ DATA

This is used to read the data in Program Memory that is pointed to by the current address pointer. This is useful for doing a verify of the programming cycle and can be used to determine the number for programming cycles that are required for the 3X overprogramming.
4.3.7 BEGIN PROGRAMMING

This is used to program the current 16-bit data (last data sent with LOAD DATA Command) into the Program Memory at the address specified by the current address pointer. The programming cycle time is specified by specification P10. After this time has elapsed, any command must be sent, which wakes the processor from the Long Write cycle. This command will be the next executed command.

4.3.8 3X OVERPROGRAMMING

Once a location has been both programmed and verified over a range of voltages, 3X overprogramming should be applied. In other words, apply three times the number of programming pulses that were required to program a location in memory, to ensure a solid programming margin.

This means that every location will be programmed a minimum of 4 times (1 + 3X overprogramming).

FIGURE 4-7: BEGIN PROGRAMMING COMMAND (PROGRAM)
Figure 4-8: Recommended Programming Flowchart

START
TEST = MCLR = RA4 = RA5 = Vss
4.75V < VDD < 5.25V

TEST = Vihh
MCLR = Vihh
Start Device Clock (on RA0),
Wait 80 Device Clocks

ISP Command
RESET ADDRESS

N = 1

ISP Command
LOAD DATA

ISP Command
BEGIN PROGRAMMING
Wait approx 100 ms

ISP Command
READ DATA
Data Correct?
N = N + 1
N > 25?
Yes
Report Programming Failure

N = 3N

ISP Command
BEGIN PROGRAMMING
Wait approx 100 ms
N = N - 1
N = 0?
Yes
Programmed all
required locations?

Yes

Verify all Locations
@ Vddmin
Data Correct?
No
Yes
Report Verify Error
@ Vddmax

No

Verify all Locations
@ Vddmin
Data Correct?
Yes

Verify all Locations
@ Vddmax
Data Correct?
No
Yes
No

Report Verify Error
@ Vddmax

DONE
5.0 CONFIGURATION WORD

Configuration bits are mapped into program memory. Each bit is assigned one memory location. In erased condition, a bit will read as ‘1’. To program a bit, the user needs to write to the memory address. The data is immaterial; the very act of writing will program the bit. The configuration word locations are shown in Table 5-3. The programmer should not program the reserved locations to avoid unpredictable results and to be compatible with future variations of the PIC17C7XX. It is also mandatory that configuration locations are programmed in the strict order starting from the first location (0xFE00) and ending with the last (0xFE0F). Unpredictable results may occur if the sequence is violated.

5.1 Reading Configuration Word

The PIC17C7XX has seven configuration locations (Table 5-1). These locations can be programmed (read as ‘0’) or left unprogrammed (read as ‘1’) to select various device configurations. Any write to a configuration location, regardless of the data, will program that configuration bit. Reading any configuration location between 0xFE00 and 0xFE07 will place the low byte of the configuration word (Table 5-2) into DAD<7:0> (PORTC). DAD<15:8> (PORTD) will be set to 0xFF. Reading a configuration location between 0xFE08 and 0xFE0F will place the high byte of the configuration word into DAD<7:0> (PORTC). DAD<15:8> (PORTD) will be set to 0xFF.

TABLE 5-1: CONFIGURATION BIT PROGRAMMING LOCATIONS

<table>
<thead>
<tr>
<th>Bit</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOSC0</td>
<td>0xFE00</td>
</tr>
<tr>
<td>FOSC1</td>
<td>0xFE01</td>
</tr>
<tr>
<td>WDTPS0</td>
<td>0xFE02</td>
</tr>
<tr>
<td>WDTPS1</td>
<td>0xFE03</td>
</tr>
<tr>
<td>PM0</td>
<td>0xFE04</td>
</tr>
<tr>
<td>PM1</td>
<td>0xFE06</td>
</tr>
<tr>
<td>BODEN</td>
<td>0xFE0E</td>
</tr>
<tr>
<td>PM2</td>
<td>0xFE0F</td>
</tr>
</tbody>
</table>

— = Unused

PM<2:0>, Processor Mode Select bits
111 = Microprocessor mode
110 = Microcontroller mode
101 = Extended Microcontroller mode
000 = Code protected microcontroller mode

BODEN, Brown-out Detect Enable
1 = Brown-out Detect Circuitry enabled
0 = Brown-out Detect Circuitry disabled

WDTPS1:WDTPS0, WDT Prescaler Select bits.
11 = WDT enabled, postscaler = 1
10 = WDT enabled, postscaler = 256
01 = WDT enabled, postscaler = 64
00 = WDT disabled, 16-bit overflow timer

FOSC1:FOSC0, Oscillator Select bits
11 = EC oscillator
10 = XT oscillator
01 = RC oscillator
00 = LF oscillator

TABLE 5-2: READ MAPPING OF CONFIGURATION BITS

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>PM1</td>
<td>—</td>
<td>PM0</td>
<td>WDTPS1</td>
<td>WDTPS0</td>
<td>FOSC1</td>
<td>FOSC0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>PM2</td>
<td>BODEN</td>
<td>PM2</td>
<td>PM2</td>
<td>PM2</td>
<td>PM2</td>
<td>PM2</td>
<td>PM2</td>
<td>PM2</td>
</tr>
</tbody>
</table>

— = Unused

PM<2:0>, Processor Mode Select bits
111 = Microprocessor mode
110 = Microcontroller mode
101 = Extended Microcontroller mode
000 = Code protected microcontroller mode

BODEN, Brown-out Detect Enable
1 = Brown-out Detect Circuitry enabled
0 = Brown-out Detect Circuitry disabled

WDTPS1:WDTPS0, WDT Prescaler Select bits.
11 = WDT enabled, postscaler = 1
10 = WDT enabled, postscaler = 256
01 = WDT enabled, postscaler = 64
00 = WDT disabled, 16-bit overflow timer

FOSC1:FOSC0, Oscillator Select bits
11 = EC oscillator
10 = XT oscillator
01 = RC oscillator
00 = LF oscillator
## 5.2 Embedding Configuration Word Information in the Hex File

To allow portability of code, a PIC17C7XX programmer is required to read the configuration word locations from the hex file when loading the hex file. If the configuration word information was not present in the hex file, then a simple warning message may be issued. Similarly, while saving a hex file, all configuration word information must be included. An option to not include the configuration word information may be provided. When embedding configuration word information in the hex file, it should be to address FE00h.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

## 5.3 Reading From and Writing To a Code Protected Device

When a device is code-protected, writing to program memory is disabled. If program memory is read, the value returned is the XNOR8 result of the actual program memory word. The XNOR8 result is the upper eight bits of the program memory word XNOR’d with the lower eight bits of the same word. This 8-bit result is then duplicated into both the upper and lower 8-bits of the read value. The configuration word can always be read and written.
5.4 CHECKSUM COMPUTATION

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

Table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently, depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

**Note:** Some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

### TABLE 5-3: CHECKSUM COMPUTATION

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0xC0DE at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC17C752</td>
<td>MP mode</td>
<td>SUM[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0xA05F</td>
<td>0x221D</td>
</tr>
<tr>
<td></td>
<td>MC mode</td>
<td>SUM[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0xA04F</td>
<td>0x220D</td>
</tr>
<tr>
<td></td>
<td>EMC mode</td>
<td>SUM[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0xA01F</td>
<td>0x211D</td>
</tr>
<tr>
<td></td>
<td>PMC mode</td>
<td>SUM_XNOR8[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x200F</td>
<td>0xE3D3</td>
</tr>
<tr>
<td>PIC17C756</td>
<td>MP mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x805F</td>
<td>0x221D</td>
</tr>
<tr>
<td></td>
<td>MC mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x804F</td>
<td>0x220D</td>
</tr>
<tr>
<td></td>
<td>EMC mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x801F</td>
<td>0x211D</td>
</tr>
<tr>
<td></td>
<td>PMC mode</td>
<td>SUM_XNOR8[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x000F</td>
<td>0xC3D3</td>
</tr>
<tr>
<td>PIC17C756A</td>
<td>MP mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x805F</td>
<td>0x221D</td>
</tr>
<tr>
<td></td>
<td>MC mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x804F</td>
<td>0x220D</td>
</tr>
<tr>
<td></td>
<td>EMC mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x801F</td>
<td>0x211D</td>
</tr>
<tr>
<td></td>
<td>PMC mode</td>
<td>SUM_XNOR8[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x000F</td>
<td>0xC3D3</td>
</tr>
<tr>
<td>PIC17C762</td>
<td>MP mode</td>
<td>SUM[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0xA05F</td>
<td>0x221D</td>
</tr>
<tr>
<td></td>
<td>MC mode</td>
<td>SUM[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0xA04F</td>
<td>0x220D</td>
</tr>
<tr>
<td></td>
<td>EMC mode</td>
<td>SUM[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0xA01F</td>
<td>0x211D</td>
</tr>
<tr>
<td></td>
<td>PMC mode</td>
<td>SUM_XNOR8[0x0000:0x1FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x200F</td>
<td>0xE3D3</td>
</tr>
<tr>
<td>PIC17C766</td>
<td>MP mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x805F</td>
<td>0x221D</td>
</tr>
<tr>
<td></td>
<td>MC mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x804F</td>
<td>0x220D</td>
</tr>
<tr>
<td></td>
<td>EMC mode</td>
<td>SUM[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x801F</td>
<td>0x211D</td>
</tr>
<tr>
<td></td>
<td>PMC mode</td>
<td>SUM_XNOR8[0x0000:0x3FFF] + (CONFIG &amp; 0xC05F)</td>
<td>0x000F</td>
<td>0xC3D3</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word
SUM[a:b] = [Sum of locations a to b inclusive]
SUM_XNOR8(a:b) = [Sum of 8-bit wide XOR copied into upper and lower byte, of locations a to b inclusive]

*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
+ = Addition
& = Bitwise AND
5.5 Device ID Register

Program memory location FDFFh is preprogrammed during the fabrication process with information on the device and revision information. These bits are accessed by a TABLR0 instruction, and are access when the TEST pin is high. As a result, the device ID bits can be read when the part is code protected.

TABLE 5-4: DEVICE ID REGISTER DECODE

<table>
<thead>
<tr>
<th>Device</th>
<th>Device ID Value</th>
<th>Resultant Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC17C766</td>
<td>0000 0001 001</td>
<td>X XXXX</td>
</tr>
<tr>
<td>PIC17C762</td>
<td>0000 0001 101</td>
<td>X XXXX</td>
</tr>
<tr>
<td>PIC17C756</td>
<td>0000 0000 001</td>
<td>X XXXX</td>
</tr>
<tr>
<td>PIC17C756A</td>
<td>0000 0010 001</td>
<td>X XXXX</td>
</tr>
<tr>
<td>PIC17C752</td>
<td>0000 0010 101</td>
<td>X XXXX</td>
</tr>
</tbody>
</table>
### 6.0 PARALLEL MODE AC/DC CHARACTERISTICS AND TIMING

#### REQUIREMENTS FOR PROGRAM/VERIFY TEST MODE

**Standard Operating Conditions**

- **Operating Temperature:** $+10^\circ C \leq T_A \leq +70^\circ C$, unless otherwise stated, (25$^\circ C$ is recommended)
- **Operating Voltage:** $4.5V \leq V_{DD} \leq 5.25V$, unless otherwise stated.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PD1</td>
<td>$V_{DDP}$</td>
<td>Supply voltage during programming</td>
<td>4.75</td>
<td>5.0</td>
<td>5.25</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PD2</td>
<td>$I_{DDP}$</td>
<td>Supply current during programming</td>
<td>—</td>
<td>—</td>
<td>50</td>
<td>mA</td>
<td>Freq = 10MHz, $V_{DD} = 5.5V$</td>
</tr>
<tr>
<td>PD3</td>
<td>$V_{DDV}$</td>
<td>Supply voltage during verify</td>
<td>$V_{DD}$ min.</td>
<td>—</td>
<td>$V_{DD}$ max.</td>
<td>V</td>
<td>Note 2</td>
</tr>
<tr>
<td>PD4</td>
<td>$V_{PP}$</td>
<td>Voltage on $V_{PP}$/MCLR pin during programming</td>
<td>12.75</td>
<td>—</td>
<td>13.25</td>
<td>V</td>
<td>Note 1</td>
</tr>
<tr>
<td>PD6</td>
<td>$I_{PP}$</td>
<td>Programming current on $V_{PP}$/MCLR pin</td>
<td>—</td>
<td>25</td>
<td>50</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>$F_{OSCP}$</td>
<td>Osc/clockin frequency during programming</td>
<td>4</td>
<td>—</td>
<td>10</td>
<td>MHz</td>
<td>$T_{CY} = 4/F_{OSCP}$</td>
</tr>
<tr>
<td>P2</td>
<td>$T_{CY}$</td>
<td>Instruction cycle</td>
<td>1</td>
<td>—</td>
<td>0.4</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>$T_{IRV2TSH}$</td>
<td>RA0, RA1, RA2, RA3, RA4 setup before TEST↑</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>$T_{TSH2MCH}$</td>
<td>TEST↑ to MCLR↑</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>$T_{BCV2TIRH}$</td>
<td>RC7:RC0, RB7:RB0 valid to RA1 or RA0↑:Address/Data input setup time; $RC7:RC0$ invalid</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>$T_{IRH2BCL}$</td>
<td>RA1 or RA0↑ to RB7:RB0; $RC7:RC0$ invalid; Address data hold time;</td>
<td>10 $T_{CY}$</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>$T_{0CKI2RBCZ}$</td>
<td>$RT\downarrow$ to RB7:RB0, $RC7:RC0$ hi-impedance</td>
<td>—</td>
<td>—</td>
<td>8$T_{CY}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>$T_{0CKI2BCV}$</td>
<td>RA1↑ to data out valid</td>
<td>—</td>
<td>—</td>
<td>10 $T_{CY}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>$T_{PROG}$</td>
<td>Programming pulse width</td>
<td>100</td>
<td>1000</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P10</td>
<td>$T_{IRH2IRL}$</td>
<td>RA0, RA1 high pulse width</td>
<td>10 $T_{CY}$</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P11</td>
<td>$T_{IRL2IRH}$</td>
<td>RA0, RA1 low pulse width</td>
<td>10 $T_{CY}$</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P12</td>
<td>$T_{0CKI2INL}$</td>
<td>RA1↑ before INT↓ (to go from prog cycle to verify w/o increment)</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P13</td>
<td>$T_{INL2RTL}$</td>
<td>RA1 valid after RA0 (to select increment or no increment going from program to verify cycle)</td>
<td>10 $T_{CY}$</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P14</td>
<td>$T_{VPSS}$</td>
<td>$V_{PP}$ setup time before RA0↑</td>
<td>100</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td>Note 1</td>
</tr>
<tr>
<td>P15</td>
<td>$T_{VPDH}$</td>
<td>$V_{PP}$ hold time after INT↓</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td>Note 1</td>
</tr>
<tr>
<td>P16</td>
<td>$T_{VDV2TSH}$</td>
<td>$V_{DD}$ stable to TEST↑</td>
<td>10</td>
<td>—</td>
<td>—</td>
<td>ms</td>
<td></td>
</tr>
<tr>
<td>P17</td>
<td>$T_{RBV2MCH}$</td>
<td>RB input (E1h) valid to $V_{PP}$/MCLR↑</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P18</td>
<td>$T_{MCH2RBIL}$</td>
<td>RB input (E1h) hold after $V_{PP}$/MCLR↑</td>
<td>10$T_{CY}$</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P19</td>
<td>$T_{VPPL2VLDL}$</td>
<td>$V_{DD}$ power down after $V_{PP}$ power down</td>
<td>10</td>
<td>—</td>
<td>—</td>
<td>ms</td>
<td></td>
</tr>
</tbody>
</table>

**Note 1:** $V_{PP}$/MCLR pin must only be equal to or greater than $V_{DD}$ at times other than programming.

**Note 2:** Program must be verified at the minimum and maximum $V_{DD}$ limits for the part.
FIGURE 6-1: PARALLEL MODE PROGRAMMING AND VERIFY TIMINGS I

<table>
<thead>
<tr>
<th>Test</th>
<th>INCER</th>
<th>RA1</th>
<th>RA0</th>
<th>RB&lt;7:0&gt;</th>
<th>RC&lt;7:0&gt;</th>
<th>P4</th>
<th>P5</th>
<th>P10</th>
<th>P11</th>
<th>P14</th>
<th>P15</th>
</tr>
</thead>
<tbody>
<tr>
<td>13V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:
- INC = ADDR
- ADDR_HI DATA_HI OUT DATA_HI OUT DDA TA_HI OUT
- ADDR_LO DATA_LO OUT DATA_LO OUT DATA_LO OUT DATA_LO OUT
- RA2 = 0
- RA3 = 0
- RA4 = 1

Verify location X + 1
Program location X - 1
Increment Address to X + 1 by pulsing RA1
Load Address X
Programming Mode Entry
Increment Address to X + 1 by raising RA0 before RA1

Programming Mode Entry Load Address X
FIGURE 6-2: PARALLEL MODE PROGRAMMING AND VERIFY TIMINGS II

Test  

Vpp/MCLR  

13V  

RA1  

RA0  

RB-7:0  

RC-7:0  

E1H  

ADDR_HI  

DATA_HI  

OUT  

DATA_HI_IN  

DATA_HI_IN  

DATA_HI_IN  

DATA_HI  

ADDR_LO  

DATA_LO  

OUT  

DATA_LO_IN  

DATA_LO_IN  

DATA_LO_IN  

P15  

P9  

P9  

P9  

P14  

Jump Address  

Programming mode entry  

Load address X  

Verify location X  

Programming mode entry  

Verify location X  

Verify location X  

Note:  

RA2 = 0  

RA3 = 0  

RA4 = 1
FIGURE 6-3: PARALLEL MODE PROGRAMMING AND VERIFY TIMINGS III

- **RA1**
- **RA0**
- **RB<7:0>**
- **RC<7:0>**

**Note:**
- Device in PGM mode
- Test = +6
- VPP/MCLR = VPP
- RA2 = 0
- RA3 = 0
- RA4 = 1
FIGURE 6-4: POWER-UP/DOWN SEQUENCE FOR PROGRAMMING
# 7.0 ELECTRICAL SPECIFICATIONS FOR SERIAL PROGRAMMING MODE

All parameters apply across the specified operating ranges unless otherwise noted.

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym</th>
<th>Characteristic</th>
<th>Min</th>
<th>Typ†</th>
<th>Max</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIHH</td>
<td></td>
<td>Programming Voltage on VPP/ MCLR pin and TEST pin.</td>
<td>12.75</td>
<td>—</td>
<td>13.75</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>IPP</td>
<td></td>
<td>Programming current on MCLR pin</td>
<td>—</td>
<td>25</td>
<td>50</td>
<td>mA</td>
<td></td>
</tr>
<tr>
<td>FOSC</td>
<td></td>
<td>Input OSC frequency on RA1</td>
<td>—</td>
<td>—</td>
<td>8</td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td>TCY</td>
<td></td>
<td>Instruction Cycle Time</td>
<td>—</td>
<td>4/FOSC</td>
<td>—</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS1</td>
<td>TVH2VH</td>
<td>Setup time between TEST = VIHH and MCLR = VIHH</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>PS2</td>
<td>TSER</td>
<td>Serial setup time</td>
<td>20</td>
<td>—</td>
<td>—</td>
<td>TCY</td>
<td></td>
</tr>
<tr>
<td>PS3</td>
<td>TSCLK</td>
<td>Serial Clock period</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>TCY</td>
<td></td>
</tr>
<tr>
<td>PS4</td>
<td>TSET1</td>
<td>Input Data Setup Time to serial clock ↓</td>
<td>15</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>PS5</td>
<td>THLD1</td>
<td>Input Data Hold Time from serial clock ↑</td>
<td>15</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>PS6</td>
<td>TDLY1</td>
<td>Delay between last clock ↓ to first clock ↑ of next command</td>
<td>20</td>
<td>—</td>
<td>—</td>
<td>TCY</td>
<td></td>
</tr>
<tr>
<td>PS7</td>
<td>TDLY2</td>
<td>Delay between last clock ↓ of command byte to first clock ↑ of read of data word</td>
<td>20</td>
<td>—</td>
<td>—</td>
<td>TCY</td>
<td></td>
</tr>
<tr>
<td>PS8</td>
<td>TDLY3</td>
<td>Delay between last clock ↓ of command byte to first clock ↑ of write of data word</td>
<td>30</td>
<td>—</td>
<td>—</td>
<td>TCY</td>
<td></td>
</tr>
<tr>
<td>PS9</td>
<td>TDLY4</td>
<td>Data input not driven to next clock input</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>TCY</td>
<td></td>
</tr>
<tr>
<td>PS10</td>
<td>TDLY5</td>
<td>Delay between last begin programming clock ↓ to last clock ↓ of next command (minimum programming time)</td>
<td>100</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
</tbody>
</table>

* These parameters are characterized but not tested.
† Data in “Typ” column is at 5V, 25°C unless otherwise stated. These parameters are for design guidance only and are not tested.
FIGURE 7-3: LOAD ADDRESS COMMAND

- **RA1/T0CKI**: Test
- **MCLR/VPP**: VIHH
- **RA5 (Clock)**: 1 2 3 4 5 6 7 8 1 2 3 15 16 1
- **RA4 (Data)**: 0 1 0 1 0 0 0 0
- **PS2**: RA4 = Input
- **PS3**: Program/Verify Test Mode

FIGURE 7-4: READ ADDRESS COMMAND

- **RA1/T0CKI**: Test
- **MCLR/VPP**: VIHH
- **RA5 (Clock)**: 1 2 3 4 5 6 7 8 1 2 3 15 16 1
- **RA4 (Data)**: 0 0 1 1 0 0 0 0
- **PS2**: RA4 = Input
- **PS3**: RA4 = Output
- **PS3**: Program/Verify Test Mode
FIGURE 7-5: LOAD DATA COMMAND

FIGURE 7-6: READ DATA COMMAND

FIGURE 7-7: BEGIN PROGRAMMING COMMAND (PROGRAM)
This document includes the programming specifications for the following devices:

- PIC18C452
- PIC18C242
- PIC18C252
- PIC18C442

1.0 PROGRAMMING THE PIC18CXXX

The PIC18CXXX can be programmed using a serial method while in the user's system. This allows for increased design flexibility. This programming specification applies to PIC18CXXX devices in all package types.

1.1 Hardware Requirements

The PIC18CXXX requires two programmable power supplies, one for VDD (2.0V to 5.5V recommended) and one for VPP (12V to 14V). Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC18CXXX allows programming of user program memory, special locations used for ID, and the configuration word for the PIC18CXXX.

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Pin Type</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLR/VPP</td>
<td>VPP</td>
<td>Programming Power</td>
</tr>
<tr>
<td>VDD</td>
<td>VDD</td>
<td>Power Supply</td>
</tr>
<tr>
<td>Vss</td>
<td>Vss</td>
<td>Ground</td>
</tr>
<tr>
<td>RB6</td>
<td>RB6</td>
<td>Serial Clock</td>
</tr>
<tr>
<td>RB7</td>
<td>RB7</td>
<td>Serial Data</td>
</tr>
</tbody>
</table>

Legend: I = Input, O = Output, P = Power
2.0 IN-CIRCUIT SERIAL PROGRAMMING MODE (ICSP)

2.1 Introduction

Serial programming mode is entered by asserting MCLR/VPP = VIHH and RB6, RB7 = 0.

Instructions are fed into the CPU serially on RB7, and are shifted in on the rising edge of the serial clock presented on RB6. Programming and verification are performed by executing TBLRD and TBLWT instructions. The address pointer to the program memory is simply the table pointer. The address pointer can be incremented and decremented by executing table reads and writes with auto-decrement and auto-increment.

2.2 ICSP OPERATION

In ICSP mode, instruction execution takes place through a serial interface using RB6 and RB7. RB7 is used to shift in instructions and shift out data from the TABLAT register. RB6 is used as the serial shift clock and the CPU execution clock. Instructions and data are shifted in LSB first.

In this mode all instructions are shifted serially, then loaded into the instruction register, and executed. No program fetching occurs from internal or external program memory. 8-bit data bytes are read from the TABLAT register via the same serial interface.

2.2.1 4-BIT SERIAL INSTRUCTIONS

A set of 4-bit instructions are provided for ICSP mode, so that the most common instructions used for ICSP can be fetched quickly, and thus reduce the amount of time required to program a device. The 4-bit opcode is shifted in while the previous instruction fetched executes. The 4-bit instruction contains the lower 4-bits of an instruction opcode. The upper 12-bits default as all 0’s. Instructions with all 0’s in the upper byte of the instruction word, are by default considered special instructions. The serial instructions are decoded as shown in Table 2-1:

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Description</th>
<th>Cycles</th>
<th>4-Bit Opcode</th>
<th>Status Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>No Operation (Shift in 16-bit instruction)</td>
<td>1</td>
<td>0000</td>
<td>None</td>
</tr>
<tr>
<td>TBLRD *</td>
<td>Table Read (no change to TBLPTR)</td>
<td>2</td>
<td>1000</td>
<td>None</td>
</tr>
<tr>
<td>TBLRD *+</td>
<td>Table Read (post-increment TBLPTR)</td>
<td>2</td>
<td>1001</td>
<td>None</td>
</tr>
<tr>
<td>TBLRD *-</td>
<td>Table Read (post-decrement TBLPTR)</td>
<td>2</td>
<td>1010</td>
<td>None</td>
</tr>
<tr>
<td>TBLRD +*</td>
<td>Table Read (pre-increment TBLPTR)</td>
<td>2</td>
<td>1011</td>
<td>None</td>
</tr>
<tr>
<td>TBLWT *</td>
<td>Table Write (no change to TBLPTR)</td>
<td>2</td>
<td>1100</td>
<td>None</td>
</tr>
<tr>
<td>TBLWT *+</td>
<td>Table Write (post-increment TBLPTR)</td>
<td>2</td>
<td>1101</td>
<td>None</td>
</tr>
<tr>
<td>TBLWT *-</td>
<td>Table Write (post-decrement TBLPTR)</td>
<td>2</td>
<td>1110</td>
<td>None</td>
</tr>
<tr>
<td>TBLWT +*</td>
<td>Table Write (pre-increment TBLPTR)</td>
<td>2</td>
<td>1111</td>
<td>None</td>
</tr>
</tbody>
</table>

Legend: Refer to the PIC18CXXX Data Sheet (DS39026) for opcode field descriptions.

Note: All special instructions not included in this table are decoded as NOP's.

In-Circuit Serial Programming™ (ICSP) is a trademark of Microchip Technology Inc.
2.2.2 INITIAL SERIAL INSTRUCTION OPERATION

Upon ICSP mode entry, the CPU is idle. The execution of the CPU is governed by a state machine. The CPU clock source comes from RB6 which also acts as the serial shift clock. The first clock transition on RB6 is absorbed after RESET. While the first instruction is being clocked in, a forced NOP is executed.

Following the FNOP instruction execution and the next shifting in of the next instruction, the serial state machine will do one of three things depending upon the 4-bit instruction that was fetched:

1. If the instruction fetched was a NOP, the state machine will suspend the CPU awaiting a 16-bit wide instruction to be shifted in.
2. If the instruction is a TBLWT, the state machine suspends the CPU from execution while sixteen bits of data are shifted in as data for the TBLWT instruction.
3. If the instruction is a TBLRD, then execution of the TBLRD instruction begins immediately for eight clock cycles, followed by eight clock cycles where the contents of the TABLAT register is shifted out onto RB7.

Once sixteen clock cycles have elapsed, the next 4-bit instruction is fetched while the current instruction is executed. Each instruction type is described in later sections.

FIGURE 2-1: SERIAL INSTRUCTION TIMING AFTER RESET
2.2.3 NOP SERIAL INSTRUCTION EXECUTION

The NOP serial instruction is used to allow execution of all other instructions not included in Table 2-1. When the NOP instruction is fetched, the serial execution state machine suspends the CPU for 16 clock cycles. During these 16 clock cycles, all 16-bits of an instruction are fed into the CPU and the NOP instruction is discarded. Once all 16 bits have been shifted in the state machine will allow the instruction to be executed for the next 4 clock cycles.

**Note:** 16-bit TBLWT and TBLRD instructions are not permitted. They will cause timing problems with the serial state machine. If the user wishes to perform a TBLWT or TBLRD instruction, it must be performed as a 4-bit instruction.

2.2.4 ONE CYCLE 16-BIT INSTRUCTIONS

If the instruction fetched is a one cycle instruction, then the instruction operation will be completed in the 4 clock cycles following the instruction fetched. During instruction execution, the next 4-bit serial instruction is fetched (See Figure 2-2).

![FIGURE 2-2: SERIAL INSTRUCTION TIMING FOR 1 CYCLE 16-BIT INSTRUCTIONS](image-url)
FIGURE 2-3: 16-BIT 1 CYCLE SERIAL INSTRUCTION FLOW AFTER RESET

- **Start**
  - MCLR = Vss, RB6, RB7 = 0
  - MCLR = Vih
  - Shift in 1st 4-bit instruction, Num_Clk = 1
  - Clock Transition RB6?
    - No
    - Yes
      - Shift(R) RB7, Num_Clk = Num_Clk + 1
      - Clock Transition RB6?
        - No
        - Yes
          - 4-bit instruction = NOP, Shift in 16-bit instruction, Num_Clk = 1
          - Clock Transition RB6?
            - No
            - Yes
              - Shift(R) RB7, Num_Clk = Num_Clk + 1
              - Num_Clk = 16?
                - No
                - Yes

- Execute 16-bit Instruction, and shift in next 4-bit instruction, Num_Clk = 1.
FIGURE 2-4: 16-BIT 1 CYCLE SERIAL INSTRUCTION FLOW

Start

execute (PC - 2), and shift in next 4-bit instruction, Num_Clk = 1,

Clock Transition RB6?

Yes

Shift(R) RB7
Num_Clk = Num_Clk + 1

4-bit instruction = NOP, Shift in 16-bit instruction, Num_Clk = 1

Clock Transition RB6?

Yes

Shift(R) RB7 into ROMLAT<15>, Num_Clk = Num_Clk + 1

No

Num_Clk = 16?

Yes

execute 16-bit Instruction, and shift in next 4-bit instruction, Num_Clk = 1,

Clock Transition RB6?

Yes

Shift(R) RB7
Num_Clk = Num_Clk + 1

No

End
2.3 **Serial Instruction Execution For Two Cycle, One Word Instructions**

When a `NOP` instruction is fetched, the serial execution state machine suspends the CPU for 16 clock cycles. During these 16 clock cycles, all 16-bits of an instruction are fed in and the `NOP` instruction is discarded.

If the instruction fetched is a two cycle, one word instruction, then the instruction operation will require a second "dummy fetch" to be performed before the instruction execution can be completed. The first cycle of the instruction will be executed in the 4 clock cycles following the instruction fetched. During the first cycle of instruction execution, the next 4-bit serial instruction is fetched. In order to perform the second half of the two cycle instruction, this 4-bit instruction loaded in must be a `NOP`, so that state machine will remain idle for the second half of the instruction. Following the fetch of the second `NOP`, the state machine will shift 16-bits of data that will be discarded. After the 16-bits of data is shifted in, the state machine will release the CPU, and allow it to perform the second half of the two cycle instruction. During the second half of the two cycle instruction execution, the next 4-bit instruction is loaded (See Figure 2-5).

### FIGURE 2-5: 2 CYCLE 1 WORD 16-BIT INSTRUCTION SEQUENCE

<table>
<thead>
<tr>
<th>Q Cycles</th>
<th>Q1 Q2 Q3 Q4</th>
<th>Q1 Q2 Q3 Q4</th>
<th>Q1 Q2 Q3 Q4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>MCLR/VPP</td>
<td>P2</td>
<td>P2</td>
<td>P2</td>
</tr>
<tr>
<td>RB6 (Clock)</td>
<td>PS PS PS PS</td>
<td>PS PS PS PS</td>
<td>PS PS PS PS</td>
</tr>
<tr>
<td>RB7 (Data)</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>1 1 0 1</td>
</tr>
</tbody>
</table>

- **Execute PC-2**
- **Fetch 4-bit NOP**
- **Fetch 16-bit Instruction**
- **Fetch 4-bit NOP, Execute 1st Cycle of 16-bit Instruction**
- **Fetch 2nd 16-bit Operand Word (discarded)**
- **Execute 2nd Cycle, Fetch Next 4-bit Instruction**

ICSP Mode

`RB7 = Input`
2.4 Serial Instruction Execution For Two Word, Two Cycle Instructions

After a NOP instruction is fetched, the serial execution state machine suspends the CPU in the Q4 state for 16 clock cycles. During these 16 clock cycles, all 16-bits of an instruction are fed in and the NOP instruction is discarded.

If the 16-bit instruction fetched is a two cycle, two word instruction, then the instruction operation will require a second operand fetch to be performed before the instruction execution can be completed. The first cycle of the instruction will be executed in the 4 clock cycles following the 16-bit instruction fetch. During the first cycle of instruction execution, the next 4-bit serial instruction is fetched. In order to perform the second half of the two cycle instruction, this 4-bit instruction loaded in must also be a NOP, so that the state machine will remain idle for the second half of the instruction. Following the fetch of the second NOP, the state machine will shift 16-bits of data that will be used as an operand for the two cycle instruction. After the 16-bits of data are shifted in, the state machine will release the CPU, and allow it to execute the second half of the two cycle instruction. During the second half of the two cycle instruction execution, the next 4-bit instruction is loaded (see Figure 2-6).

FIGURE 2-6: 16-BIT 2 CYCLE 2 WORD INSTRUCTION SEQUENCE

<table>
<thead>
<tr>
<th>Q Cycles</th>
<th>‘Q1’</th>
<th>‘Q2’</th>
<th>‘Q3’</th>
<th>‘Q4’</th>
<th>‘Q1’</th>
<th>‘Q2’</th>
<th>‘Q3’</th>
<th>‘Q4’</th>
<th>‘Q1’</th>
<th>‘Q2’</th>
<th>‘Q3’</th>
<th>‘Q4’</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLR/VPP = Vihh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB6 (Clock)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB7 (Data)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RB7 = Input

ICSP Mode
FIGURE 2-7: 16-BIT 2 CYCLE 2 WORD SERIAL INSTRUCTION FLOW AFTER RESET

Start

MCLR = VPP, RB6, RB7 = 0

MCLR = VIH

execute FNOP and shift in 1st 4-bit instruction, Num_Clk = 1,

Clock Transition RB6?
Yes

Shift(R) RB7, Num_Clk = Num_Clk + 1

No

4-bit instruction = NOP, Shift in 16-bit instruction, Num_Clk = 1

Clock Transition RB6?
Yes

Shift(R) RB7, Num_Clk = Num_Clk + 1

No

Clock Transition RB6?

Enable CPU, execute 1st cycle of 16-bit instruction, and shift in next 4-bit instruction, Num_Clk = 1,

Clock Transition RB6?
Yes

Shift(R) RB7, Num_Clk = Num_Clk + 1

No

Shift(R) RB7, Num_Clk = Num_Clk + 1

Num_Clk = 16?
Yes

execute 2nd cycle of 16-bit instruction, and shift in next 4-bit instruction Num_Clk = 1

No

Clock Transition RB6?

Shift(R) RB7, Num_Clk = Num_Clk + 1

No

Shift(R) RB7, Num_Clk = Num_Clk + 1

End
FIGURE 2-8: 16-BIT 2 CYCLE 2 WORD SERIAL INSTRUCTION FLOW

- Start
- Execute (PC-2) and shift in 4-bit instruction, Num_Clk = 1,
- Clock Transition RB6?
  - Yes, shift (R) RB7, Num_Clk = Num_Clk + 1
  - No, 4-bit instruction = NOP, shift in 16-bit instruction, Num_Clk = 1
- Clock Transition RB6?
  - Yes, shift (R) RB7, Num_Clk = Num_Clk + 1
  - No, Num_Clk = 16?
    - No, execute 2nd cycle of 16-bit instruction, and shift in next 4-bit instruction, Num_Clk = 1
    - Yes, execute 2nd cycle of 16-bit instruction, Num_Clk = Num_Clk + 1
- Clock Transition RB6?
  - Yes, shift (R) RB7, Num_Clk = Num_Clk + 1
  - No, Num_Clk = 16?
    - No, execute 2nd cycle of 16-bit instruction, and shift in next 4-bit instruction, Num_Clk = 1
    - Yes, execute 2nd cycle of 16-bit instruction, Num_Clk = Num_Clk + 1
- End
2.5 TBLWT Instruction

The TBLWT instruction is a unique two cycle instruction.

All forms of TBLWT instructions (post/pre-increment, post decrement, etc.) are encoded as 4-bit special instructions. This is useful as TBLWT instructions are used repeatedly in ICSP mode. A 4-bit instruction will minimize the total number of clock cycles required to perform programming algorithms.

The TBLWT instruction sequence operates as follows:

1. The 4-bit TBLWT instruction is read in by the state machine on RB7 during the 4 clock cycle execution of the instruction fetched previous to the TBLWT (which is an FNOP if the TBLWT is executed following a reset).

2. Once the state machine recognizes that the instruction fetched is a TBLWT, the state machine proceeds to fetch in the 16-bits of data that will be written into the program memory location pointed to by the TBLPTR.

3. The serial state machine releases the CPU to execute the first cycle of the TBLWT instruction while the first 4 bits of the 16-bit data word are shifted in. After the first cycle of TBLWT instruction has completed the state machine shifts in the remaining 12 of the sixteen bits of data. The data word will not be used until the second cycle of the instruction.

4. After all 16-bits of data are shifted in and the first cycle of the TBLWT is performed, the CPU is allowed to execute the second cycle of the TBLWT operation, programming the current memory location with the 16-bit value. The next instruction following the TBLWT instruction is shifted in during the execution of the second cycle (See Figure 2-9).

The TBLWT instruction is used in ICSP mode to program the EPROM array. When writing a 16-bit value to the EPROM, ID locations, or configuration locations, the device, RB6, must be held high for the appropriate programming time during the TBLWT instruction as specified by parameter P9.

When RB6 is asserted low the device will cease programming the specified location.

After RB6 is asserted low, RB6 is held low for the time specified by parameter P10, to allow high voltage discharge of the program memory array.

**FIGURE 2-9: TBLWT INSTRUCTION SEQUENCE**

```
<table>
<thead>
<tr>
<th>MCLR/VPP = VIHH</th>
<th>Q Cycles</th>
<th>'Q1', 'Q2', 'Q3', 'Q4</th>
<th>'Q1', 'Q2', 'Q3', 'Q4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB6 (Clock)</td>
<td>1</td>
<td>2 3 4 5</td>
<td>12 13 14 15 16</td>
</tr>
<tr>
<td>RB7 (Data)</td>
<td>1</td>
<td>0 0 0 0</td>
<td>1 0 1 1</td>
</tr>
</tbody>
</table>
```

- Execute PC-2
- Fetch TBLWT
- Execute 1st Cycle TBLWT
- Load TBLWT Data
- Execute 2nd Cycle TBLWT
- and fetch next 4-bit instruction

RB7 = Input

ICSP Mode
FIGURE 2-10: TBLWT SERIAL INSTRUCTION FLOW AFTER RESET

Start

MCLR = VSS, RB6, RB7 = 0

MCLR = VIHH

Execute FNOP, and shift in 4-bit TBLWT instruction, Num_Clk = 1,

Clock Transition RB6? No

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Clock Transition RB6? No

Yes

4-bit instruction = TBLWT, Execute 1st cycle of TBLWT, Begin Shifting in TBLWT data, Num_Clk = 1

Clock Transition RB6? No

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Num_Clk = 4? No

Yes

shift in last 12 bits of TBLWT data, Num_Clk = 1,

Execute 2nd cycle of TBLWT instruction and shift in next 4-bit instruction, Num_Clk = 1,

Clock Transition RB6? No

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

End
FIGURE 2-11: TBLWT SERIAL INSTRUCTION FLOW

Start

Execute (PC-2), and shift in 4-bit TBLWT instruction, Num_Clk = 1.

Clock Transition RB6?

No

4-bit instruction = TBLWT, Execute 1st cycle of TBLWT, Begin Shifting in TBLWT data, Num_Clk = 1

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Clock Transition RB6?

No

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Num_Clk = 4?

No

Yes

Shift in last 12 bits of TBLWT data, Num_Clk = 1,

Execute 2nd cycle of TBLWT instruction and shift in next 4-bit instruction, Num_Clk = 1.

Clock Transition RB6?

No

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

End

Num_Clk = 12?

Yes

No
2.6 TBLRD Instruction

The TBLRD instruction is another unique two cycle instruction.

All forms of TBLRD instructions (post/pre-increment, post decrement, etc.) are encoded as 4-bit special instructions. This is useful as TBLRD instructions are used repeatedly in ICSP mode. A 4-bit instruction will minimize the total number of clock cycles required to perform programming algorithms.

The TBLRD instruction sequence operates as follows:

1. The 4-bit TBLRD instruction is read in by the state machine on RB7 during the 4 clock cycle execution of the instruction fetched previous to the TBLRD (which is an FNOP if the TBLRD is executed following a reset).

2. Once the state machine recognizes that the instruction fetched is a TBLRD, the state machine releases the CPU and allows execution of the first and second cycles of the TBLRD instruction for eight clock cycles. When the TBLRD is performed, the contents of the program memory byte pointed to by the TBLPTR is loaded into the TABLAT register.

3. After eight clock cycles have transitioned on RB6, and the TBLRD instruction has completed, the state machine will suspend the CPU for eight clock cycles. During these eight clock cycles, the state machine configures RB7 as an output, and will shift out the contents of the TABLAT register onto RB7 LSb first.

4. When the state machine has shifted out all eight bits of data, the state machine suspends the CPU to allow an instruction pre-fetch. Four (4) clock cycles are required on RB6 to shift in the next 4-bit instruction.

<table>
<thead>
<tr>
<th>FIGURE 2-12: TBLRD INSTRUCTION SEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q Cycles</td>
</tr>
<tr>
<td>MCLR/VPP = VIHH</td>
</tr>
<tr>
<td>RB6 (Clock)</td>
</tr>
<tr>
<td>RB7 (Data)</td>
</tr>
<tr>
<td>Execute PC-2</td>
</tr>
<tr>
<td>Fetch TBLRD</td>
</tr>
<tr>
<td>RB7 = Input</td>
</tr>
<tr>
<td>ICSP Mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
FIGURE 2-13: TBLRD SERIAL INSTRUCTION FLOW AFTER RESET

Start

MCLR = Vss, RB6, RB7 = 0

MCLR = VihH

Execute FNOP, and shift in 4-bit TBLRD instruction, Num_Clk = 1,

Clock Transition RB6?

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Enable CPU, execute 1st and 2nd cycle TBLRD instruction

Clock Transition RB6?

Yes

TBLRD instruction execution takes place here Num_Clk = Num_Clk + 1

Num_Clk = 8?

No

Shift out 8-bits of data to RB7

Yes

End

No

Clock Transition RB6?

Yes

Shift(R) TABLAT<0> out onto RB7 Num_Clk = Num_Clk + 1

Num_Clk = 8?

No

Shift in next 4-bit instruction

Yes

Clock Transition RB6?

Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Num_Clk = 4?

No

Yes
FIGURE 2-14: TBLRD SERIAL INSTRUCTION FLOW

Start

Execute (PC-2), and shift in 4-bit TBLRD instruction, Num_Clk = 1,

Clock Transition RB6? Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Execute 1st and 2nd cycle TBLRD instruction

Clock Transition RB6? Yes

TBLRD instruction execution takes place here Num_Clk = Num_Clk + 1

Num_Clk = 8?

Yes

Shift out 8-bits of data to RB7

No

Clock Transition RB6? Yes

Shift(R) TABLAT<0> out onto RB7 Num_Clk = Num_Clk + 1

Num_Clk = 8?

Yes

Shift in next 4-bit instruction

No

Clock Transition RB6? Yes

Shift(R) RB7 Num_Clk = Num_Clk + 1

Num_Clk = 4?

Yes

End

No
2.6.1 SOFTWARE COMMANDS

ICSP commands of the PICmicro® MCU are supported in the PIC18CXXX family by simply combining CPU instructions. Once in In-Circuit Serial Programming (ICSP) mode, the instructions are loaded into a shift register, and the device waits for a command to be received. The ICSP commands for the PIC16CXXX family are now “pseudo-commands” and are shown in Table 2-2. The following sections are a description of how the pseudo-commands can be implemented using CPU instructions.

TABLE 2-2: ICSP PSEUDO COMMAND MAPPING

<table>
<thead>
<tr>
<th>ICSP Command</th>
<th>Golden Gate Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>MOVLW #Address1 MOVWF TBLPTRL MOVLW #Address2 MOVWF TBLPTRH MOVLW #Address3 MOVWF TBLPTRU</td>
</tr>
<tr>
<td>Load Data</td>
<td>Not needed. Data encoded in 4-bit TBLWT instruction sequence.</td>
</tr>
<tr>
<td>Read Data</td>
<td>TBLRD instruction</td>
</tr>
<tr>
<td>Increment Address</td>
<td>Not needed. Use TBLWT with increment/decrement (TBLWT <em>+/</em>-).</td>
</tr>
<tr>
<td>Load Address</td>
<td>MOVLW #Addr_low MOVWF TBLPTRL MOVLW #Addr_high MOVWF TBLPTRH MOVLW #Addr_upper MOVWF TBLPTRU</td>
</tr>
<tr>
<td>Reset Address</td>
<td>MOVLW #Data MOVWF TBLPTRL MOVWF TBLPTRH MOVWF TBLPTRU</td>
</tr>
<tr>
<td>Begin programming</td>
<td>TBLWT</td>
</tr>
<tr>
<td>End Programming</td>
<td>Not needed. Programming will cease at the end of TBLWT execution.</td>
</tr>
</tbody>
</table>
2.6.2 RESET ADDRESS

A reset of the program memory pointer is a write to the upper, high, and low bytes of the TBLPTR. To reset the program memory pointer, the following instruction sequence is used.

```assembly
NOP ;(4-BIT INSTRUCTION)
MOVLW 00h
NOP ;(4-BIT INSTRUCTION)
MOVWF TBLPTRU, 0
NOP ;(4-BIT INSTRUCTION)
MOVWF TBLPTRH, 0
NOP ;(4-BIT INSTRUCTION)
MOVWF TBLPTRL, 0
```
FIGURE 2-15: RESET ADDRESS SERIAL INSTRUCTION SEQUENCE

Start

1. execute (PC - 2), shift in next 4-bit instruction, Num_Clk = 1,

2. On rising edge RB6 Shift(R) RB7 into Shift Reg<3>, Num_Clk = Num_Clk + 1

   a. Num_Clk = 4?
      - No: 4-bit instruction = NOP, Shift in 16-bit MOVWF instruction, Num_Clk = 1
      - Yes: 4-bit instruction = NOP, Shift in 16-bit MOVLF instruction, Num_Clk = 1

   b. On rising edge RB6 Shift(R) RB7 into Shift Reg<15>, Num_Clk = Num_Clk + 1

   c. MOVWF 00h
      - No: Num_Clk = 16?
        - Yes: Execute MOVWF instruction, shift in 4-bit NOP instruction, Num_Clk = 1
        - No: Num_Clk = 4?
          - Yes: Execute MOVWF instruction, shift in 4-bit NOP instruction, Num_Clk = 1
          - No: Num_Clk = 1

   d. On rising edge RB6 Shift(R) RB7 into Shift Reg<3>, Num_Clk = Num_Clk + 1

   e. MOVWF TBLPTRM,0
      - Yes: Execute MOVWF instruction, shift in next 4-bit instruction, Num_Clk = 1

End

4-bit instruction = NOP, Shift in 16-bit MOVWF instruction, Num_Clk = 1

On rising edge RB6 Shift(R) RB7 into Shift Reg<15>, Num_Clk = Num_Clk + 1

- Num_Clk = 16?
  - Yes: execute MOVWF instruction, shift in 4-bit NOP instruction, Num_Clk = 1
  - No: Num_Clk = 4?
    - Yes: execute MOVWF instruction, shift in 4-bit NOP instruction, Num_Clk = 1
    - No: Num_Clk = 1

On rising edge RB6 Shift(R) RB7 into Shift Reg<3>, Num_Clk = Num_Clk + 1

- Num_Clk = 16?
  - Yes: execute MOVWF instruction, shift in next 4-bit instruction, Num_Clk = 1
  - No: Num_Clk = 4?
    - Yes: execute MOVWF instruction, shift in next 4-bit instruction, Num_Clk = 1
    - No: Num_Clk = 1

On rising edge RB6 Shift(R) RB7 into Shift Reg<15>, Num_Clk = Num_Clk + 1

- MOVWF TBLPTRM,0
  - Yes: Execute MOVWF instruction, shift in next 4-bit instruction, Num_Clk = 1

End
2.6.3 LOAD ADDRESS

This is used to load the address pointer to the Program Memory with a specific 22-bit value. This is useful when a specific range of locations are to be accessed. To load the address into the table pointer, the following commands must be used:

```
NOP ; 4-bit instruction
MOVLW Low_Address
NOP ; 4-bit instruction
MOVWF TBLPTRL, 0
NOP ; 4-bit instruction
MOVLW High_Address
NOP ; 4-bit instruction
MOVWF TBLPTRH, 0
NOP ; 4-bit instruction
MOVLW Upper_Address
NOP ; 4-bit instruction
MOVWF TBLPTRU, 0
```
FIGURE 2-16: LOAD ADDRESS SERIAL INSTRUCTION SEQUENCE

Start

1. execute (PC - 2), shift in next 4-bit instruction, Num_Clk = 1,
   - On rising edge RB6 Shift(R) RB7 into Shift Reg<3>, Num_Clk = Num_Clk + 1
     - Num_Clk = 4?
       - No
       - Num_Clk = 16?
         - Yes
         - (NOP)
         - MOVLW LOW_Address
       - Yes
       - execute MOVLW Instruction, shift in next 4-bit instruction, Num_Clk = 1,
     - Num_Clk = 16?
       - No
       - (NOP)
     - Num_Clk = 4?
       - No
       - Num_Clk = 16?
         - Yes
         - (NOP)
         - MOVLW HIGH_Address
       - Yes
       - execute MOVLW Instruction, shift in next 4-bit instruction, Num_Clk = 1,

2. 4-bit instruction = NOP, shift in 16-bit MOVWF instruction, Num_Clk = 1
   - On rising edge RB6 Shift(R) RB7 into Shift Reg<15>, Num_Clk = Num_Clk + 1
     - Num_Clk = 4?
       - No
       - Num_Clk = 16?
         - Yes
         - (NOP)
         - MOVWF TBLPTR,0
       - Yes
       - execute MOVWF Instruction, shift in 4-bit NOP instruction, Num_Clk = 1,
     - Num_Clk = 16?
       - No
       - (NOP)
     - Num_Clk = 4?
       - No
       - Num_Clk = 16?
         - Yes
         - (NOP)
         - MOVWF HIGH_Address
       - Yes
       - execute MOVWF Instruction, shift in next 4-bit instruction, Num_Clk = 1,

End
2.6.4 ICSP BEGIN PROGRAMMING

Programming is performed by executing a TBLWT instruction. In ICSP mode the TBLWT instruction sequence will include 16-bits of data that are shifted into a data buffer, and then written to the word location that is addressed by the TBLPTR. Although the TBLPTR addresses the program memory on a byte wide boundary, all 16-bits of data that are shifted in during the TBLWT sequence are written at once. The 16-bits are shifted into the TABLAT and buffer registers. The TBLPTR points to the word that will be programmed; it can point to either the high or the low byte. (See Figure 2-17).

The sequence for programming a location could occur as follows:

1. Setup the TLBPTR with the first ok address to be programmed (even or odd byte).
2. Shift in a 4 bit TBLWT instruction.
3. 16-bits of data are then shifted in for programming both high and low byte of the first programmed location.
4. Execute TBLWT instruction to program location.
5. Verify high byte (odd address) by executing TBLRD *- (post-decrement). (If TBLPTR pointing at odd address.)
6. Verify low byte (even address) by executing TBLRD *+ (post-increment). TBLPTR is pointing to odd address again.
7. If location doesn't verify, go back to step 4.
8. If location does verify, begin 3x overprogramming.

The TBLWT instruction offers flexibility with multiple addressing modes: pre-increment, post-increment, post decrement, and no change of the TBLPTR. These modes eliminate the need for the increment address command sequence.

FIGURE 2-17: DATA BUFFERING SCHEME FOR ICSP
2.6.5 PROGRAMMING INSTRUCTION SEQUENCE

The series of instructions needed to execute a programming sequence is as follows. Many of the instruction sequences used in the following example are also shown in previous sections.

NOP ; 4-bit instruction
; Set up low byte of program address
MOVLW Low_Bit_Address ; = 00
NOP ; 4-bit instruction
MOVWF TBLPTRL, 0
NOP ; 4-bit instruction
; Set up high byte of program address
MOVLW High_Bit_Address ; = 00
NOP ; 4-bit instruction
MOVWF TBLPTRH, 0
NOP ; 4-bit instruction
; Set up upper byte of program address
MOVLW Upper_Bit_Address; = 00
NOP ; 4-bit instruction
MOVWF TBLPTRU, 0 ; Program data byte included in TBLWT instruction sequence
TBLWT+* ; TBLPTR = 000000h

A write of a program memory location with an odd or an even address causes a long write cycle in ICSP mode. The 16-bit data is encoded in the TBLWT sequence and is loaded into the temporary buffer register for word wide writes.

The user must wait 100 µs for the long write to complete before the next instruction is executed.

2.6.6 VERIFY SEQUENCE

The table pointer = 000001h in the last example. A TBLRD will then read the odd address byte of the current program word address location first. The verify sequence will be as follows:

; Read/verify high byte first
TBLRD*-
; TBLPTR = 0000 post-dec
; Read/verify low byte
TBLRD*

The first TBLRD decrements the table pointer to point to the even address byte of the current program word. After the first and second cycle of the TBLRD are performed, all 8-bits of data are shifted out on RB7. The fetch of the second TBLRD occurs on the next 4 clock cycles. The second TBLRD does not modify the table pointer address. This allows another programming cycle (TBLWT+) to take place if the verify doesn’t match the program data without having to update the table pointer.

If the contents of the verify do not match the intended program data word, then the TBLWT instruction must be repeated with the correct contents of the current program word. Therefore, only one instruction needs to be performed to repeat the programming cycle:

TBLWT+*

2.6.7 3X OVER PROGRAMMING

Once a location has been both programmed and verified over a range of voltages, 3x over programming should be applied. In other words, apply three times the number of programming pulses that were required to program a location in memory, to ensure a solid programming margin.

This means that every location will be programmed a minimum of 4 times (1 + 3x over programming).
FIGURE 2-18: DETAILED PROGRAMMING FLOW CHART – PROGRAM MEMORY

Start

MCLR = VPP, RB6, RB7 = 0

N = 0

Execute FNOP for four clock cycles
shift in 4-bit NOP

Start

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVLW Low_Addr
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRL
instruction for 16 clock cycles

Execute MOVWF for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF Upper_Addr
instruction for 16 clock cycles

Execute MOVWF for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRU
instruction for 16 clock cycles

 Execute current instruction
for 4 clock cycles, and
shift in 4-bit TBLWT+*

Execute 1st cycle
TBLWT +*, and shift in
first 4-bits of data
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

Execute 2nd cycle
TBLWT +* for 4 clock cycles
Shift in TBLRD *-
for 4 clock cycles

Hold RB6
Clock high

Wait 100 µsec to
ensure programming

Clock Low
for Discharge

Hold RB6
Clock high (P10)

Execute 1st and 2nd cycle
TBLRD *- for 8 clock cycles

Shift Data Out
for 8 clock cycles

Hold CPU,
Shift in TBLRD *-
for 4 clock cycles

Execute 1st and 2nd cycle
TBLRD *- for 8 clock cycles

Shift Data Out
for 8 clock cycles

Verify?

Yes

N = N + 1

Yes

Report Programming Failure

No

N > 25?

Yes
FIGURE 2-19: DETAILED PROGRAMMING FLOW CHART – PROGRAM MEMORY (CONTINUED)

A

N = 3 * N

Execute current instruction, 
Shift in TBLWT *+ 
for 4 clock cycles

Execute 1st cycle 
TBLWT *+, or *, and shift in 
first 4-bits of data 
for 4 clock cycles

Shift in last 12-bits of data 
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle 
TBLWT * for 4 clock cycles 
Shift in TBLWT *+ 
for 4 clock cycles

Hold RB6 high

Wait 100 µS

Clock Low

for Discharge

N = N - 1

No

Execute 2nd cycle 
TBLWT * for 4 clock cycles 
Shift in TBLWT *+ 
for 4 clock cycles

Shift in last 12-bits of data 
for 12 clock cycles

Execute current instruction 
for 4 clock cycles, and 
shift in 4-bit TBLRD+*

Hold RB6 high

Wait 100 µsec to 
ensure programming

All locations 
programmed?

Yes

To B

No

Verify all Locations 
@ VDDMIN

Data Correct?

Yes

Verify all Locations 
@ VDDMAX

Data Correct?

Yes

Report Verify 
Error @ VDDMAX

No

Report Verify 
Error @ VDDMIN

End
2.6.8 LOAD CONFIGURATION

The Configuration registers are located in C0 memory, and are only addressable when the high address bit of the TBLPTR (bit 21) is set. Test program memory contains test memory, configuration registers, calibration registers, and ID locations. The desired address must be loaded into all three bytes of the table pointer to program specific ID locations or the configuration bits. To program the configuration registers, the following sequence must be followed:

- NOP ; 4-bit instruction
- MOVFLW 03h
- NOP ; 4-bit instruction
- MOVFLW Low_Config_Address
- NOP ; 4-bit instruction
- MOVFLW High_Config_Address
- NOP ; 4-bit instruction
- MOVFLW TBLPTR, 0
- MOVFLW TBLPTRH, 0
- TBLWT *+

2.6.9 END PROGRAMMING

When programming occurs, 16 bits of data are programmed into memory. The 16-bits of data are shifted in during the TBLWT sequence. After the programming command (TBLWT) has been executed, the user must wait for 100 μs until programming is complete, before another command can be executed by the CPU. There is no command to end programming.

RB6 must remain high for as long as programming is desired. When RB6 is lowered programming will cease.

After the falling edge occurs on RB6, RB6 must be held low for a period of time so that a high voltage discharge can be performed to ensure that the program array isn’t stressed at high voltage during execution of the next instruction. The high voltage discharge will occur while RB6 is low following the programming time.
FIGURE 2-20: SYMBOLIC PROGRAMMING FLOW CHART – CONFIG WORD / ID LOCATION

START

MCLR = Vss
4.75V < VDD < 5.25V

MCLR = Vpp

ICSP Command
LOAD CONFIGURATION
Address = 300000h

N = 0

No

Program ID Loc?

Yes

ICSP Command
LOAD DATA

N = 100

ICSP Command
BEGIN PROGRAMMING

Wait approx 100 µs

N = N - 1

ICSP Command
LOAD ADDRESS
Address = 300000h

No

N = 0?

Yes

ICSP Command
READ DATA

Data Correct?

Yes

Report Programming Failure

No

ICSP Command
LOAD DATA

N > 25?

No

Data Correct?

Yes

N = 3N

Yes

ICSP Command
BEGIN PROGRAMMING

Wait approx 100 µs

N = 0

Address = 300000h?

No

N = 0

Yes

ICSP Command
INCREMENT ADDRESS

Data Correct?

Yes

Report Programming Failure

No

Verify all Locations @ VDDMIN

Data Correct?

Yes

Verify all Locations @ VDDMAX

Data Correct?

Yes

DONE

Report Verify Error @ VDDMIN

Report Verify Error @ VDDMAX
FIGURE 2-21: DETAILED PROGRAMMING FLOW CHART – CONFIG WORD

START

MCLR = Vss
4.75V < Vdd < 5.25V

MCLR = Vhhh

Execute FNOP for four clock cycles
shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF 30
instruction for 16 clock cycles

Execute MOVFLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRU
instruction for 16 clock cycles

Execute MOVWF for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRH
instruction for 16 clock cycles

Execute MOVWF for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF 00
instruction for 16 clock cycles

Execute MOVFLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF 00
instruction for 16 clock cycles

Execute MOVFLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBPLTRL
instruction for 16 clock cycles

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBPLTRH
instruction for 16 clock cycles

Execute MOVFLW for 4 clock cycles
and shift in 4-bit NOP

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+

Execute 1st cycle
TBLWT, and shift in
first 4-bits of config. reg.,
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

N = 1?

Yes

Execute 2nd cycle
TBLWT for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

RB6 High

Wait 100 μsec to
ensure programming

Clock Low
for Discharge

N = N - 1

No

Execute 2nd cycle
TBLWT* for 4 clock cycles
Shift in TBLWT *-
for 4 clock cycles

Wait 100 μsec to
ensure programming

N = 99

Execute last fetched inst.
for 4 clock cycles
and shift in 4-bit TBLWT*+
FIGURE 2-22: DETAILED PROGRAMMING FLOW CHART – CONFIG WORD

A

Execute 1st cycle TBLWT*- for 4 clock cycles, and shift in first 4-bits of config. reg. for 4 clock cycles

Shift in last 12-bits of data for 12 clock cycles

Execute 2nd cycle TBLWT*- for 4 clock cycles, shift in TBLRD*+ for 4 clock cycles

Wait 100 µsec to ensure programming

Execute 1st and 2nd cycle TBLRD*+ for 8 clock cycles

Shift Data Out for 8 clock cycles

Shift in TBLRD*+ for 4 clock cycles

Execute 1st and 2nd cycle TBLRD*+ for 8 clock cycles

Shift Data Out for 8 clock cycles

B

Verify?

No

Report Verify Error

Yes

All locations programmed?

No

Report Verify Error @ VDDMIN

Yes

Verify all ID_Locations @ VDDMIN

Data Correct?

No

Verify all Locations @ VDDMAX

Yes

Data Correct?

No

Report Verify Error @ VDDMIN

Yes

Report Verify Error @ VDDMAX

DONE
FIGURE 2-23: DETAILED PROGRAMMING FLOW CHART – ID LOCATION

Start

- MCLR = VPP, RB6, RB7 = 0
- N = 0

Execute FNOP for four clock cycles
shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVLW Low_Addr
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRL
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVLW High_Addr
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRH
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRL
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRU
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRL
instruction for 16 clock cycles

Execute MOVLW for 4 clock cycles
and shift in 4-bit NOP

4-bit instruction = NOP,
Shift in 16-bit MOVWF TBLPTRU
instruction for 16 clock cycles

Execute current instruction
for 4 clock cycles, and shift in 4-bit TBLWT+*

Execute 1st cycle
TBLWT +*, and shift in first 4-bits of data
for 4 clock cycles

Shift in last 12-bits of data
for 12 clock cycles

Execute 2nd cycle
TBLWT +* for 4 clock cycles
Shift in TBLRD *-
for 4 clock cycles

Wait 100 µsec to ensure programming

Execute 1st and 2nd cycle
TBLRD *- for 8 clock cycles

Shift Data Out
for 8 clock cycles

Execute 1st cycle
TBLWT +*, and shift in last 12-bits of data
for 12 clock cycles

Execute 2nd cycle
TBLWT +* for 4 clock cycles
Shift in TBLRD *-
for 4 clock cycles

Wait 100 µsec to ensure programming

Execute 1st and 2nd cycle
TBLRD *- for 8 clock cycles

Shift Data Out
for 8 clock cycles

Verify?

- No
  - N = N + 1

- Yes
  - N > 25?
    - No
      - Report Programming Failure
    - Yes
      - N = N + 1
FIGURE 2-24: DETAILED PROGRAMMING FLOW CHART – ID LOCATIONS (CONTINUED)

A

N = 3 * N

Execute current instruction, Shift in TBLWT *+ for 4 clock cycles

Execute 1st cycle TBLWT *+ or *, and shift in first 4-bits of data for 4 clock cycles

Shift in last 12-bits of data for 12 clock cycles

N = 1?

Yes

Wait 100 µsec to ensure programming

Execute 2nd cycle TBLWT * for 4 clock cycles Shift in TBLWT ** for 4 clock cycles

Execute 1st cycle TBLWT **, and shift in first 4-bits of data for 4 clock cycles

Shift in last 12-bits of data for 12 clock cycles

Execute 2nd cycle TBLWT **+ for 4 clock cycles, and shift in 4-bit TBLWT +*

Wait 100 µsec to ensure programming

Verify all Locations @ VDDMIN

Data Correct? Yes

End

All locations programmed? Yes

No

Verify all Locations @ VDDMAX

Data Correct? Yes

No

Report Verify Error @ VDDMAX

Report Verify Error @ VDDMIN

No

Yes

B

N = N - 1

No

Execute 2nd cycle TBLWT * for 4 clock cycles Shift in TBLWT * for 4 clock cycles

N = 3 * N

Yes

All locations programmed?
3.0 CONFIGURATION WORD

The configuration bits can be programmed (read as '0') or left unprogrammed (read as '1') to select various device configurations. These bits are mapped starting at program memory location 300000h.

The user will note that address 300000h is beyond the user program memory space. In fact, it belongs to the configuration memory space (300000h – 3FFFFFh).

TABLE 3-1: CONFIGURATION BITS AND DEVICE IDS

<table>
<thead>
<tr>
<th>Filename</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Default / unprogrammed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>300000h</td>
<td>CONFIG1L</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>1111 1111</td>
</tr>
<tr>
<td>300001h</td>
<td>CONFIG1H</td>
<td>RES1</td>
<td>RES1</td>
<td>OSCSEN</td>
<td>—</td>
<td>—</td>
<td>FOSC2</td>
<td>FOSC1</td>
<td>FOSC0 1111 -111</td>
</tr>
<tr>
<td>300002h</td>
<td>CONFIG2L</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>BORV1</td>
<td>BORV0</td>
<td>BODEN</td>
<td>PWRREN</td>
<td>---- 1111</td>
</tr>
<tr>
<td>300003h</td>
<td>CONFIG2H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>WDTPS2</td>
<td>WDTPS1</td>
<td>WDTPS0</td>
<td>WDTEN</td>
<td>---- 1111</td>
</tr>
<tr>
<td>300005h</td>
<td>CONFIG3H</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>CCP2MX ---- ----1</td>
</tr>
<tr>
<td>300006h</td>
<td>CONFIG4L</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>RES1 STVREN ---- ----1</td>
</tr>
<tr>
<td>3FFFFEh</td>
<td>DEVID1</td>
<td>DEV2</td>
<td>DEV1</td>
<td>DEV0</td>
<td>REV4</td>
<td>REV3</td>
<td>REV2</td>
<td>REV1</td>
<td>REV0 ---- ----</td>
</tr>
<tr>
<td>3FFFFFh</td>
<td>DEVID2</td>
<td>DEV10</td>
<td>DEV9</td>
<td>DEV8</td>
<td>DEV7</td>
<td>DEV6</td>
<td>DEV5</td>
<td>DEV4</td>
<td>DEV3 ---- ----</td>
</tr>
</tbody>
</table>

Legend:  
- x = unknown, u = unchanged, - = unimplemented, q = value depends on condition, grayed cells are unimplemented read as 0

Note 1:  
- Reserved – Read as 1.
Register 3-1: Configuration Register 1 High (CONFIG1H: Byte Address 300001h)

<table>
<thead>
<tr>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>U-0</th>
<th>U-0</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Reserved</td>
<td>OSCSEN</td>
<td>—</td>
<td>—</td>
<td>FOSC2</td>
<td>FOSC1</td>
<td>FOSC0</td>
</tr>
</tbody>
</table>

bit 7-6  **Reserved**: Read as ‘1’

bit 5  **OSCSEN**: Oscillator System Clock Switch Enable bit

1 = Oscillator system clock switch option is disabled (OSCA is source)
0 = Oscillator system clock switch option is enabled
(OSCA → OSCB, OSCB → OSCA switching is enabled)

bit 4-3  **Reserved**: Read as ‘0’

bit 2-0  **FOSC2:FOSC0**: Oscillator Selection bits

111 = RC oscillator w/ OSC2 configured as RA6
110 = HS oscillator with PLL enabled/Clock frequency = (4 x Fosc1)
101 = EC oscillator w/ OSC2 configured as RA6
100 = EC oscillator w/ OSC2 configured as divide by 4 clock output
011 = RC oscillator
010 = HS oscillator
001 = XT oscillator
000 = LP oscillator

Legend
R = Readable bit  P = Programmable bit  U = Unimplemented bit, read as ‘0’
- n = Value when device is unprogrammed  u = Unchanged from programmed state

Register 3-2: Configuration Register 1 Low (CONFIG1L: Byte Address 300000h)

<table>
<thead>
<tr>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
</tr>
</tbody>
</table>

bit 7-0  **CP**: Code Protection bits (apply when in Code Protected Microcontroller Mode)

1 = Program memory code protection off
0 = All of program memory code protected

Legend
R = Readable bit  P = Programmable bit  U = Unimplemented bit, read as ‘0’
- n = Value when device is unprogrammed  u = Unchanged from programmed state
Register 3-3: Configuration Register 2 High (CONFIG2H: Byte Address 300003h)

<table>
<thead>
<tr>
<th>bit 7-4</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved: Read as ‘0’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bit 3-1  WDTPS2:WDTPS0: Watchdog Timer Postscale Select bits
111 = 1:128  
110 = 1:64  
101 = 1:32  
100 = 1:16  
011 = 1:8  
010 = 1:4  
001 = 1:2  
000 = 1:1

bit 0  WDTEN: Watchdog Timer Enable bit
1 = WDT enabled  
0 = WDT disabled (control is placed on the SWDTE bit)

Legend
R = Readable bit  P = Programmable bit  U = Unimplemented bit, read as ‘0’
- n = Value when device is unprogrammed  u = Unchanged from programmed state

Register 3-4: Configuration Register 2 Low (CONFIG2L: Byte Address 300002h)

<table>
<thead>
<tr>
<th>bit 7-4</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
<th>R/P-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved: Read as ‘0’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bit 3-2  BORV1:BORV0: Brown-out Reset Voltage bits
11 = VBOR set to 2.5V  
10 = VBOR set to 2.7V  
01 = VBOR set to 4.2V  
00 = VBOR set to 4.5V

bit 1  BOREN: Brown-out Reset Enable bit (1)
1 = Brown-out Reset enabled  
0 = Brown-out Reset disabled
Enabling Brown-out Reset automatically enables the Power-up Timer (PWRT) regardless of the value of bit PWRTEN. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

bit 0  PWRTEN: Power-up Timer Enable bit (1)
1 = PWRT disabled  
0 = PWRT enabled
Enabling Brown-out Reset automatically enables the Power-up Timer (PWRT) regardless of the value of bit PWRTEN. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

Legend
R = Readable bit  P = Programmable bit  U = Unimplemented bit, read as ‘0’
- n = Value when device is unprogrammed  u = Unchanged from programmed state
3.1 ID Locations

A user may store identification information (ID) in 8 ID locations. The ID locations are mapped in [0x200000:0x200007]. It is recommended that the user use only the 4 least significant bits of each ID location. The ID locations do not read out in a scrambled fashion after code protection is enabled. For all devices it is recommended that all ID locations are written as ‘1111 bbbb’ where bbbb is the ID information. When the upper four bits of an ID location is written as ‘1111’, the resulting opcode when executed is read as a NOP. This allows Reset testing of test program memory after ID locations have been programmed.
3.2  Embedding Configuration Word Information in the Hex File

To allow portability of code, a PIC18C4X programmer is required to read the configuration word locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, all configuration word information must be included. An option to not include the configuration word information may be provided. When embedding configuration word information in the hex file, it should be to address FE00h.

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

3.3  CHECKSUM COMPUTATION

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0xAA at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC18C452</td>
<td>Disable</td>
<td>SUM[0C000:0x7FFF] + CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0x8148</td>
<td>0x809E</td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
<td>CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0x005E</td>
<td>0x0068</td>
</tr>
<tr>
<td>PIC18C442</td>
<td>Disable</td>
<td>SUM[0x000:0x3FFF] + CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0xC148</td>
<td>0xC09E</td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
<td>CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0x0062</td>
<td>0x006C</td>
</tr>
<tr>
<td>PIC18C252</td>
<td>Disable</td>
<td>SUM[0x000:0x7FFF] + CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0x8148</td>
<td>0x809E</td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
<td>CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0x005E</td>
<td>0x0068</td>
</tr>
<tr>
<td>PIC18C242</td>
<td>Disable</td>
<td>SUM[0x000:0x3FFF] + CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0xC148</td>
<td>0xC09E</td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
<td>CFGW0 &amp; 0xFF + CFGW1 &amp; 0x27 + CFGW2 &amp; 0xF + CFGW3 &amp; 0x0F + CFGW4 &amp; 0x00 + CFGW5 &amp; 0x01 + CFGW6 &amp; 0x03 + CFGW7 &amp; 0x00</td>
<td>0x0062</td>
<td>0x006C</td>
</tr>
</tbody>
</table>

Legend:  
CFGW = Configuration Word  
SUM[a:b] = [Sum of locations a to b inclusive]  
SUM_ID = Byte-wise sum of lower four bits of all ID locations  
+ = Addition  
& = Bitwise AND
### 4.0 AC/DC CHARACTERISTICS

#### TIMING REQUIREMENTS FOR PROGRAM/VERIFY TEST MODE

**Standard Operating Conditions**

Operating Temperature: \(+10^\circ\text{C} \leq T_A \leq +70^\circ\text{C}\), unless otherwise stated, \((25^\circ\text{C} \text{ is recommended})\)

Operating Voltage: \(4.5V \leq V_{DD} \leq 5.25V\), unless otherwise stated.

<table>
<thead>
<tr>
<th>Parameter No.</th>
<th>Sym</th>
<th>Characteristic</th>
<th>Min</th>
<th>Typ†</th>
<th>Max</th>
<th>Units</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIHH</td>
<td>Programming Voltage on VPP/ MCLR pin and TEST pin.</td>
<td>VDD + 4.0</td>
<td>—</td>
<td>13.25</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPP</td>
<td>Programming current on MCLR pin</td>
<td>25</td>
<td>50</td>
<td>mA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>TSER</td>
<td>Serial setup time</td>
<td>20</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>TSClk</td>
<td>Serial Clock period</td>
<td>100</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>TSET1</td>
<td>Input Data Setup Time to serial clock ↓</td>
<td>15</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>THLD1</td>
<td>Input Data Hold Time from serial clock ↓</td>
<td>15</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>TDLY1</td>
<td>Delay between last clock ↓ to first clock ↑ of next command</td>
<td>20</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>TDLY2</td>
<td>Delay between last clock ↓ of command byte to first clock ↑ of read of data word</td>
<td>20</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>TDLY4</td>
<td>Data input not driven to next clock input</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>TDLY5</td>
<td>RB6 high time (minimum programming time)</td>
<td>100</td>
<td>—</td>
<td>—</td>
<td>µs</td>
<td></td>
</tr>
<tr>
<td>P10</td>
<td>TDLY6</td>
<td>RB6 low time after programming (high voltage discharge time)</td>
<td>100</td>
<td>—</td>
<td>—</td>
<td>ns</td>
<td></td>
</tr>
</tbody>
</table>

* These parameters are characterized but not tested.
† Data in “Typ” column is at 5V, 25°C unless otherwise stated. These parameters are for design guidance only and are not tested.
This document includes the programming specifications for the following devices:

- PIC16F627
- PIC16F628
- PIC16LF627
- PIC16LF628

1.0 PROGRAMMING THE PIC16F62X

The PIC16F62X is programmed using a serial method. The serial mode will allow the PIC16F62X to be programmed while in the user's system. This allows for increased design flexibility. This programming specification applies to PIC16F62X devices in all packages.

PIC16F62X devices may be programmed using a single +5 volt supply (low voltage programming mode).

1.1 Hardware Requirements

The PIC16F62X requires one programmable power supply for Vdd (4.5V to 5.5V) and a VPP of 12V to 14V or VPP of (4.5V to 5.5V) when using low voltage. Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC16F62X allows programming of user program memory, data memory, special locations used for ID, and the configuration word.

PIN DESCRIPTIONS (DURING PROGRAMMING): PIC16F62X

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Function</th>
<th>Pin Type</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB4</td>
<td>PGM</td>
<td>I</td>
<td>Low voltage programming input if configuration bit equals 1</td>
</tr>
<tr>
<td>RB6</td>
<td>CLOCK</td>
<td>I</td>
<td>Clock input</td>
</tr>
<tr>
<td>RB7</td>
<td>DATA</td>
<td>I/O</td>
<td>Data input/output</td>
</tr>
<tr>
<td>MCLR</td>
<td>VTEST MODE</td>
<td>I/O</td>
<td>Program Mode Select</td>
</tr>
<tr>
<td>VDD</td>
<td>VDD</td>
<td>P</td>
<td>Power Supply</td>
</tr>
<tr>
<td>Vss</td>
<td>VSS</td>
<td>P</td>
<td>Ground</td>
</tr>
</tbody>
</table>

Legend: I = Input, O = Output, P = Power

*In the PIC16F62X, the programming high voltage is internally generated. To activate the programming mode, high voltage needs to be applied to MCLR input. Since the MCLR is used for a level source, this means that MCLR does not draw any significant current.*
2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The user memory space extends from 0x0000 to 0x7FFF. In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000-0x7FFF) being user program memory and the second half (0x2000-0x3FFF) being configuration memory. The PC will increment from 0x0000 to 0x7FFF and wrap to 0x000, 0x2000 to 0x3FFF and wrap around to 0x2000 (not to 0x0000). Once in configuration memory, the highest bit of the PC stays a ‘1’, thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode as described in section 2.3.

In the configuration memory space, 0x2000-0x200F are physically implemented. However, only locations 0x2000 through 0x2007 are available. Other locations are reserved. Locations beyond 0x200F will physically access user memory. (See Figure 2-1).

2.2 ID Locations

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000 : 0x2003]. It is recommended that the user use only the four least significant bits of each ID location. In some devices, the ID locations read-out in an unscrambled fashion after code protection is enabled. For these devices, it is recommended that ID location is written as "11 1111 1000 bbbb" where ‘bbbb’ is ID information.

In other devices, the ID locations read out normally, even after code protection. To understand how the devices behave, refer to Table 4-1.

To understand the scrambling mechanism after code protection, refer to Section 3-1.

FIGURE 2-1: PROGRAM MEMORY MAPPING
2.3 Program/Verify Mode

The program/verify mode is entered by holding pins RB6 and RB7 low while raising MCLR pin from VIL to VHH (high voltage) or by applying VDD to MCLR and raising RB3 from VIL to VDD. Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program memory. RB6 and RB7 are Schmitt Trigger Inputs in this mode.

**Note:** The OSC must not have 72 osc clocks while the device MCLR is between VIL and VHH.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at VIL). This means that all I/O are in the reset state (High impedance inputs).

The normal sequence for programming is to use the load data command to set a value to be written at the selected address. Issue the begin programming command followed by read data command to verify, and then increment the address.

A device reset will clear the PC and set the address to 0. The “increment address” command will increment the PC. The “load configuration” command will set the PC to 0x2000. The available commands are shown in Table 2-1.

2.3.1 LOW-VOLTAGE PROGRAMMING MODE

When LVP bit is set to ‘1’, the low-voltage programming entry is enabled. Since the LVP configuration bit allows low voltage programming entry in its erased state, an erased device will have the LVP bit enabled at the factory. While LVP is ‘1’, RB4 is dedicated to low voltage programming. Bring MCLR to VDD and then RB4 to VDD to enter programming mode. All other specifications for high-voltage ICSP™ apply.

To disable low voltage mode, the LVP bit must be programmed to ‘0’. This must be done while entered with high voltage entry mode (LVP bit= 1). RB4 is now a general purpose I/O pin.

2.3.2 SERIAL PROGRAM/VERIFY OPERATION

The RB6 pin is used as a clock input pin, and the RB7 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (RB6) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin RB7 is required to have a minimum setup and hold time (see AC/DC specifications) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1 µs between the command and the data. After this delay, the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first.

Therefore, during a read operation the LSB will be transmitted onto pin RB7 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1µs delay is also specified between consecutive commands.

All commands are transmitted LSB first. Data words are also transmitted LSB first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1 µs is required between a command and a data word (or another command).

The commands that are available are:

2.3.2.1 LOAD CONFIGURATION

After receiving this command, the program counter (PC) will be set to 0x2000. By then applying 16 cycles to the clock pin, the chip will load 14-bits in a “data word,” as described above, to be programmed into the configuration memory. A description of the memory mapping schemes of the program memory for normal operation and configuration mode operation is shown in Figure 2-1. After the configuration memory is entered, the only way to get back to the user program memory is to exit the program/verify test mode by taking MCLR low (VIL).
2.3.2.2 LOAD DATA FOR PROGRAM MEMORY

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 5-1.

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB ... LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>X X 0 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Load Data for Program Memory</td>
<td>X X 0 0 1 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Program Memory</td>
<td>X X 0 1 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>X X 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>Begin Erase Programming Cycle</td>
<td>0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Begin Programming Only Cycle</td>
<td>0 1 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Load Data for Data Memory</td>
<td>X X 0 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Data Memory</td>
<td>X X 0 1 0 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Bulk Erase Program Memory</td>
<td>X X 1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>Bulk Erase Data Memory</td>
<td>X X 1 0 1 1</td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 2-2: PROGRAM FLOW CHART - PIC16F62X PROGRAM MEMORY

Start

Set VDD = VDDP

Program Cycle

Read Data Command

Data Correct?

Report Programming Failure

No

All Locations Done?

Increment Address Command

No

Verify all Locations @ VDDMIN

Report Verify Error @ VDDMIN

No

Data Correct?

Verify all Locations @ VDDMAX

Report Verify Error @ VDDMAX

No

Data Correct?

Done

PROGRAM CYCLE

Load Data Command

Begin Programming Command

Wait 2 ms
FIGURE 2-3: PROGRAM FLOW CHART - PIC16F62X CONFIGURATION MEMORY

Start

Load Configuration Data

Program ID Location?

Yes

Program Cycle

Read Data Command

No

Report Programming Failure

Data Correct?

Yes

No

Data Correct?

Yes

Increment Address Command

Address = 0x2004?

Yes

Increment Address Command

No

Increment Address Command

Increment Address Command

Increment Address Command

Report Program Configuration Word Error

Data Correct?

Yes

No

Set VDD = VDDMAX

Read Data Command

Set VDD = VDDMAX

Read Data Command

Done Yes

Data Correct?
2.3.2.3 LOAD DATA FOR DATA MEMORY

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied. However, the data memory is only 8-bits wide, and thus only the first 8-bits of data after the start bit will be programmed into the data memory. It is still necessary to cycle the clock the full 16 cycles in order to allow the internal circuitry to reset properly. The data memory contains 64 words. Only the lower 8-bits of the PC are decoded by the data memory, and therefore if the PC is greater than 0x3F, it will wrap around and address a location within the physically implemented memory. If the device is code protected, the data is read as all zeros.

2.3.2.4 READ DATA FROM PROGRAM MEMORY

After receiving this command, the chip will transmit data bits out of the program memory (user or configuration) currently accessed starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 5-2.

2.3.2.5 READ DATA FROM DATA MEMORY

After receiving this command, the chip will transmit data bits out of the data memory starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. As previously stated, the data memory is 8-bits wide, and therefore, only the first 8-bits that are output are actual data.

2.3.2.6 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.3.2.7 BEGIN ERASE/PROGRAM CYCLE

A load command must be given before every begin programming command. Programming of the appropriate memory (test program memory, user program memory or data memory) will begin after this command is received and decoded. An internal timing mechanism executes an erase before write. The user must allow for both erase and programming cycle times for programming to complete. No “end programming” command is required.

2.3.2.8 BEGIN PROGRAMMING

A load command must be given before every begin programming command. Programming of the appropriate memory (test program memory, user program memory or data memory) will begin after this command is received and decoded. An internal timing mechanism executes a write. The user must allow for program cycle time for programming to complete. No “end programming” command is required.

This command is similar to the ERASE/PROGRAM CYCLE command, except that a word erase is not done. It is recommended that a bulk erase be performed before starting a series of programming only cycles.

2.3.2.9 BULK ERASE PROGRAM MEMORY

After this command is performed, the next program command will erase the entire program memory.

To perform a bulk erase of the program memory, the following sequence must be performed.

1. Do a “Load Data All 1’s” command.
2. Do a “Bulk Erase User Memory” command.
3. Do a “Begin Programming” command.
4. Wait 10 ms to complete bulk erase.

If the address is pointing to the test program memory (0x2000 - 0x200F), then both the user memory and the test memory will be erased. The configuration word will not be erased, even if the address is pointing to location 0x2007.

Note: If the device is code-protected, the BULK ERASE command will not work.

2.3.2.10 BULK ERASE DATA MEMORY

To perform a bulk erase of the data memory, the following sequence must be performed.

1. Do a “Load Data All 1’s” command.
2. Do a “Bulk Erase Data Memory” command.
3. Do a “Begin Programming” command.
4. Wait 10 ms to complete bulk erase.

Note: All BULK ERASE operations must take place at 4.5 to 5.5 VDD range.
2.4 **Programming Algorithm Requires Variable VDD**

The PIC16F62X uses an intelligent algorithm. The algorithm calls for program verification at VDDmin. as well as VDDmax. Verification at VDDmin. guarantees good “erase margin”. Verification at VDDmax guarantees good “program margin”.

The actual programming must be done with VDD in the VDDP range (See Table 5-1).

VDDP = VCC range required during programming.
VDDmin. = minimum operating VDD spec for the part.
VDDmax.= maximum operating VDD spec for the part.

Programmers must verify the PIC16F62X at its specified VDD max. and VDDmin levels. Since Microchip may introduce future versions of the PIC16F62X with a broader VDD range, it is best that these levels are user selectable (defaults are ok).

**Note:** Any programmer not meeting these requirements may only be classified as “prototype” or “development” programmer but not a “production” quality programmer.
3.0 CONFIGURATION WORD

The PIC16F62X has several configuration bits. These bits can be set (reads '0') or left unchanged (reads '1') to select various device configurations.

3.1 Device ID Word

The device ID word for the PIC16F62X is located at 2006h.

FIGURE 3-1: CONFIGURATION WORD FOR PIC16F877/876/873

<table>
<thead>
<tr>
<th>Device ID Value</th>
<th>Device ID Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F627</td>
<td>00 0111 111</td>
</tr>
<tr>
<td>PIC16F628</td>
<td>00 0111 001</td>
</tr>
</tbody>
</table>

TABLE 3-1:

<table>
<thead>
<tr>
<th>Device ID Value</th>
<th>Device ID Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F627</td>
<td>00 0111 111</td>
</tr>
<tr>
<td>PIC16F628</td>
<td>00 0111 001</td>
</tr>
</tbody>
</table>

bit 13-10: CP1:CP0: Code Protection bits (2)

- Code protection for 2K program memory
  - 11 = Program memory code protection off
  - 10 = 0400h-07FFh code protected
  - 01 = 0200h-07FFh code protected
  - 00 = 0000h-07FFh code protected

- Code protection for 1K program memory
  - 11 = Program memory code protection off
  - 10 = Program memory code protection off
  - 01 = 0200h-03FFh code protected
  - 00 = 0000h-03FFh code protected

bit 8: CPD: Data Code Protection bit (3)

- 1 = Data memory code protection off
- 0 = Data memory code protected

bit 7: LVP: Low Voltage Programming Enable

- 1 = RB4/PGM pin has PGM function, low voltage programming enabled
- 0 = RB4/PGM is digital I/O, HV on MCLR must be used for programming

bit 6: BODEN: Brown-out Detect Reset Enable bit (1)

- 1 = BOD reset enabled
- 0 = BOD reset disabled

bit 5: MCLRE: RA5/MCLR pin function select

- 1 = RA5/MCLR pin function is MCLR
- 0 = RA5/MCLR pin function is digital I/O, MCLR internally tied to Vdd

bit 3: PWRTE: Power-up Timer Enable bit (1)

- 1 = PWRT disabled
- 0 = PWRT enabled

bit 2: WDTE: Watchdog Timer Enable bit

- 1 = WDT enabled
- 0 = WDT disabled

bit 4,1-0: FOSC2:FOSC0: Oscillator Selection bits (4)

- 111 = ER oscillator: CLKOUT function on RA6/OSC2/CLKOUT pin, Resistor on RA7/OSC1/CLKIN
- 110 = ER oscillator: I/O function on RA6/OSC2/CLKOUT pin, Resistor on RA7/OSC1/CLKIN
- 101 = INTRC oscillator: CLKOUT function on RA6/OSC2/CLKOUT pin, I/O function on RA7/OSC1/CLKIN
- 100 = INTRC oscillator: I/O function on RA6/OSC2/CLKOUT pin, I/O function on RA7/OSC1/CLKIN
- 011 = EXTCLK: I/O function on RA6/OSC2/CLKOUT pin, CLKIN on RA7/OSC1/CLKIN
- 010 = HS oscillator: High speed crystal/resonator on RA6/OSC2/CLKOUT and RA7/OSC1/CLKIN
- 001 = XT oscillator: Crystal/resonator on RA6/OSC2/CLKOUT and RA7/OSC1/CLKIN
- 000 = LP oscillator: Low power crystal on RA6/OSC2/CLKOUT and RA7/OSC1/CLKIN

Note 1: Enabling Brown-out Reset automatically enables Power-up Timer (PWRT) regardless of the value of bit PWRTE. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

2: All of the CP1:CP0 pairs have to be given the same value to enable the code protection scheme listed. The entire program EEPROM will be erased if the code protection is reduced.

3: The entire data EEPROM will be erased when the code protection is turned off. The calibration space in the test memory is not erased.

4: When MCLRE is asserted in INTRC or ER mode, the internal clock oscillator is disabled.
4.0 CODE PROTECTION

For PIC16F62X devices, once code protection is enabled, all program memory locations read all 0’s. The ID locations and the configuration word read out in an unscrambled fashion. Further programming is disabled for the entire program memory as well as data memory. It is possible to program the ID locations and the configuration word.

4.1 Disabling Code-Protection

It is recommended that the following procedure be performed before any other programming is attempted. It is also possible to turn code protection off (code protect bit = 1) using this procedure; however, all data within the program memory and the data memory will be erased when this procedure is executed, and thus, the security of the data or code is not compromised.

Procedure to disable code protect:

a) Execute load configuration (with a ‘1’ in bit 4, code protect).
b) Increment to configuration word location (0x2007)
c) Execute command (000001)
d) Execute command (000111)
e) Execute ‘Begin Programming’ (001000)
f) Wait 10 ms
g) Execute command (000001)
h) Execute command (000111)

4.2 Embedding Configuration Word and ID Information in the Hex File

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Specifically for the PIC16F62X, the EEPROM data memory should also be embedded in the hex file (see Section 5.1).

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.
4.3 CHECKSUM COMPUTATION

4.3.1 CHECKSUM

Checksum is calculated by reading the contents of the PIC16F62X memory locations and adding up the opcodes up to the maximum user addressable location, e.g., 0x1FF for the PIC16F62X. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC16F62X devices is shown in Table 4-1.

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F627</td>
<td>OFF</td>
<td>SUM[0x0000:0x3FFF] + CFGW &amp; 0x3DFF</td>
<td>0x39FF</td>
<td>0x05CD</td>
</tr>
<tr>
<td></td>
<td>0x200 : 0x3FF</td>
<td>SUM[0x0000:0x01FF] + CFGW &amp; 0x3DFF + SUM_ID</td>
<td>0x3DFE</td>
<td>0xFFB3</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td></td>
<td>0x3BFE</td>
<td>0x07CC</td>
</tr>
<tr>
<td>PIC16F628</td>
<td>OFF</td>
<td>SUM[0x0000:0x07FF] + CFGW &amp; 0x3DFF</td>
<td>0x35FF</td>
<td>0x01CD</td>
</tr>
<tr>
<td></td>
<td>0x400 : 0xFFF</td>
<td>SUM[0x0000:0x03FF] + CFGW &amp; 0x3DFF + SUM_ID</td>
<td>0x5BFE</td>
<td>0x0DB3</td>
</tr>
<tr>
<td></td>
<td>0x200 : 0x7FF</td>
<td>SUM[0x0000:0x01FF] + CFGW &amp; 0x3DFF + SUM_ID</td>
<td>0x49FE</td>
<td>0xFBB3</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3DFF + SUM_ID</td>
<td>0x37FE</td>
<td>0x03CC</td>
</tr>
</tbody>
</table>

Legend: 
- CFGW = Configuration Word
- SUM[a:b] = [Sum of locations a to b inclusive]
- SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble. For example, ID0 = 0x1, ID1 = 0x2, ID3 = 0x3, ID4 = 0x4, then SUM_ID = 0x1234
- *Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]
- + = Addition
- & = Bitwise AND
5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

5.1 Embedding Data EEPROM Contents in Hex File

The programmer should be able to read data EEPROM information from a hex file and conversely (as an option) write data EEPROM contents to a hex file along with program memory information and fuse information.

The 64 data memory locations are logically mapped starting at address 0x2100. The format for data memory storage is one data byte per address location, LSB aligned.

<table>
<thead>
<tr>
<th>TABLE 5-1: AC/DC CHARACTERISTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TIMING REQUIREMENTS FOR PROGRAM/VERIFY MODE</strong></td>
</tr>
</tbody>
</table>

**Standard Operating Conditions (unless otherwise stated)**

<table>
<thead>
<tr>
<th>Operating Temperature:</th>
<th>0°C ≤ TA ≤ +70°C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage:</td>
<td>4.5V ≤ VDD ≤ 5.5V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Sym</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
<th>Conditions/Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDD level for word operations, program memory</td>
<td>VDD</td>
<td>2.0</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDD level for word operations, data memory</td>
<td>VDD</td>
<td>2.0</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDD level for bulk erase/write operations, program and data memory</td>
<td>VDD</td>
<td>4.5</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High voltage on MCLR and RA4/T0CKI for test-mode entry</td>
<td>VIH</td>
<td>VDD + 3.5</td>
<td>13.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCLR rise time (VSS to VHH) for test mode entry</td>
<td>tVHHR</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(RB6, RB7) input high level</td>
<td>Vih1</td>
<td>0.8VDD</td>
<td>V</td>
<td></td>
<td></td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>(RB6, RB7) input low level</td>
<td>VIL1</td>
<td>0.2VDD</td>
<td>V</td>
<td></td>
<td></td>
<td>Schmitt Trigger input</td>
</tr>
<tr>
<td>RB&lt;7:4&gt; setup time before MCLR↑ (test mode selection pattern setup time)</td>
<td>tset0</td>
<td>100</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB&lt;7:4&gt; hold time after MCLR↑ (test mode selection pattern setup time)</td>
<td>thld0</td>
<td>5</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serial Program/Verify</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in setup time before clock↓</td>
<td>tset1</td>
<td>100</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in hold time after clock↓</td>
<td>thld1</td>
<td>100</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data input not driven to next clock input (delay required between command/data or command/command)</td>
<td>tdly1</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delay between clock↓ to clock↑ of next command or data</td>
<td>tdly2</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock↑ to data out valid (during read data)</td>
<td>tdly3</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel Program/Verify</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in setup time before clock↓</td>
<td>tset0</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in hold time after clock↓</td>
<td>thld0</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB6 and RB7 setup time before clock↓</td>
<td>tset1</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB6 and RB7 hold time after clock↓</td>
<td>thld1</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA4/T0CKI (clock)↓ to (clock)↑</td>
<td>tdly4</td>
<td>2.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB7 (data/command select input) setup before RA4/T0CKI (clock)↑</td>
<td>tset2</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB7 (data/command select input) hold time after RA4/T0CKI (clock)↓</td>
<td>thld2</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA4/T0CKI (clock)↑ to data out valid</td>
<td>tdly5</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA6 (hi/lo select) valid to data out valid</td>
<td>tdly6</td>
<td>1.0</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erase cycle time</td>
<td>tera</td>
<td>2</td>
<td>5</td>
<td>ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming cycle time</td>
<td>tprog</td>
<td>2</td>
<td>5</td>
<td>ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time delay from program to compare (HV discharge time)</td>
<td>tdis</td>
<td>0.5</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 5-1: LOAD DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-2: READ DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-3: INCREMENT ADDRESS COMMAND (PROGRAM/VERIFY)
This document includes the programming specifications for the following devices:

- PIC16F83
- PIC16CR83
- PIC16F84
- PIC16CR84
- PIC16F84A
- PIC16F877

1.0 PROGRAMMING THE PIC16F8X

The PIC16F8X is programmed using a serial method. The serial mode will allow the PIC16F8X to be programmed while in the users system. This allows for increased design flexibility. This programming specification applies to PIC16F8X devices in all packages.

1.1 Hardware Requirements

The PIC16F8X requires one programmable power supply for Vdd (4.5V to 5.5V) and a Vpp of 12V to 14V. Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC16F8X allows programming of user program memory, data memory, special locations used for ID, and the configuration word.

PIN DESCRIPTIONS (DURING PROGRAMMING): PIC16F8X

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Function</th>
<th>Pin Type</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB6</td>
<td>CLOCK</td>
<td>I</td>
<td>Clock input</td>
</tr>
<tr>
<td>RB7</td>
<td>DATA</td>
<td>I/O</td>
<td>Data input/output</td>
</tr>
<tr>
<td>MCLR</td>
<td>VTEST MODE</td>
<td>P*</td>
<td>Program Mode Select</td>
</tr>
<tr>
<td>VDD</td>
<td>VDD</td>
<td>P</td>
<td>Power Supply</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>P</td>
<td>Ground</td>
</tr>
</tbody>
</table>

Legend: I = Input, O = Output, P = Power

*In the PIC16F8X, the programming high voltage is internally generated. To activate the programming mode, high voltage needs to be applied to MCLR input. Since the MCLR is used for a level source, this means that MCLR does not draw any significant current.
2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The user memory space extends from 0x0000 to 0x1FFF (8K), of which 1K (0x0000 - 0x03FF) is physically implemented. In actual implementation the on-chip user program memory is accessed by the lower 10-bits of the PC, with the upper 3-bits of the PC ignored. Therefore if the PC is greater than 0x3FF, it will wrap around and address a location within the physically implemented memory. (See Figure 2-1).

In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000-0x1FFF) being user program memory and the second half (0x2000-0x3FFF) being configuration memory. The PC will increment from 0x0000 to 0x1FFF and wrap to 0x000 or 0x2000 to 0x3FFF and wrap around to 0x2000 (not to 0x0000). Once in configuration memory, the highest bit of the PC stays a '1', thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode as described in Section 2.3.

In the configuration memory space, 0x2000-0x200F are physically implemented. However, only locations 0x2000 through 0x2007 are available. Other locations are reserved. Locations beyond 0x200F will physically access user memory. (See Figure 2-1).

2.2 ID Locations

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000 : 0x2003]. It is recommended that the user use only the four least significant bits of each ID location. In some devices, the ID locations read-out in an unscrambled fashion after code protection is enabled. For these devices, it is recommended that ID location is written as “11 1111 1000 bbbb” where ‘bbbb’ is ID information.

In other devices, the ID locations read out normally, even after code protection. To understand how the devices behave, refer to Table 4-2.

To understand the scrambling mechanism after code protection, refer to Section 4.0.
FIGURE 2-1: PROGRAM MEMORY MAPPING

\begin{tabular}{|c|c|c|}
\hline
0 & 1FF & 3FF \_ 400 \\
\hline
1FFF & Implemented & 1 KK & Implemented \\
2000 & Implemented & Implemented & Implemented \\
2008 & Not Implemented & Not Implemented & Not Implemented \\
3FFF & Not Implemented & Not Implemented & Not Implemented \\
\hline
\end{tabular}
2.3 **Program/Verify Mode**

The program/verify mode is entered by holding pins RB6 and RB7 low while raising MCLR pin from VIL to V\text{IH} (high voltage). Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program memory. RB6 and RB7 are Schmitt Trigger Inputs in this mode.

**Note:** The OSC must not have 72 osc clocks while the device MCLR is between V\text{IL} and V\text{IH}.

The sequence that enters the device into the program/verify mode places all other logic into the reset state (the MCLR pin was initially at V\text{IL}). This means that all I/O are in the reset state (High impedance inputs).

The normal sequence for programming is to use the load data command to set a value to be written at the selected address. Issue the begin programming command followed by read data command to verify, and then increment the address.

### 2.3.1 SERIAL PROGRAM/VERIFY OPERATION

The RB6 pin is used as a clock input pin, and the RB7 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (RB6) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin RB7 is required to have a minimum setup and hold time (see AC/DC specifications) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1 \(\mu\text{s}\) between the command and the data. After this delay, the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first.

Therefore, during a read operation the LSB will be transmitted onto pin RB7 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1\(\mu\text{s}\) delay is also specified between consecutive commands.

All commands are transmitted LSB first. Data words are also transmitted LSB first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1 \(\mu\text{s}\) is required between a command and a data word (or another command).

The commands that are available are:

### 2.3.1.1 LOAD CONFIGURATION

After receiving this command, the program counter (PC) will be set to 0x2000. By then applying 16 cycles to the clock pin, the chip will load 14-bits in a “data word,” as described above, to be programmed into the configuration memory. A description of the memory mapping schemes of the program memory for normal operation and configuration mode operation is shown in Figure 2-1. After the configuration memory is entered, the only way to get back to the user program memory is to exit the program/verify test mode by taking MCLR low (V\text{IL}).

Note: The OSC must not have 72 osc clocks while the device MCLR is between V\text{IL} and V\text{IH}.
2.3.1.2 LOAD DATA FOR PROGRAM MEMORY

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 5-1.

**TABLE 2-1: COMMAND MAPPING FOR PIC16F83/CR83/F84/CR84**

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB … LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Load Data for Program Memory</td>
<td>0 0 0 0 1 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Program Memory</td>
<td>0 0 0 1 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>0 0 0 1 1 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Begin Programming</td>
<td>0 0 1 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Load Data for Data Memory</td>
<td>0 0 0 0 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Data Memory</td>
<td>0 0 0 1 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Bulk Erase Program Memory</td>
<td>0 0 1 0 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Bulk Erase Data Memory</td>
<td>0 0 1 0 1 1 1</td>
<td>0, data (14), 0</td>
</tr>
</tbody>
</table>

**TABLE 2-2: COMMAND MAPPING FOR PIC16F84A/PIC16F877**

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB … LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>X X 0 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Load Data for Program Memory</td>
<td>X X 0 0 1 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Program Memory</td>
<td>X X 0 1 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>X X 0 1 1 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Begin Erase Programming Cycle</td>
<td>0 0 1 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Begin Programming Only Cycle</td>
<td>0 1 1 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Load Data for Data Memory</td>
<td>X X 0 1 1 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Data Memory</td>
<td>X X 1 0 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Bulk Erase Program Memory</td>
<td>X X 1 0 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Bulk Erase Data Memory</td>
<td>X X 1 0 1 1 1</td>
<td>0, data (14), 0</td>
</tr>
</tbody>
</table>
FIGURE 2-2: PROGRAM FLOW CHART - PIC16F8X PROGRAM MEMORY

Start

Set VDD = VDDP

Program Cycle

Read Data Command

Data Correct? Yes

Load Data Command

Begin Programming Command

Wait 10 ms

No

Report Programming Failure

Increment Address Command

All Locations Done?

No

Report Verify Error @ VDDMIN

Verify all Locations @ VDDMIN

Data Correct?

Verify all Locations @ VDDMAX

Report Verify Error @ VDDMAX

Data Correct?

Done
FIGURE 2-3: PROGRAM FLOW CHART - PIC16F8X CONFIGURATION MEMORY

Start

Load Configuration Data

Program ID Location?

Yes

Program Cycle

Read Data Command

No

Program ID Location?

Yes

Read Data Command

Report Programming Failure

Data Correct?

No

Data Correct?

Yes

Yes

Yes

Program Cycle (Config. Word)

Set VDD = VDDMAX

Report Program Configuration Word Error

Data Correct?

No

Set VDD = VDDMAX

Read Data Command

Yes

Yes

Data Correct?

Read Data Command

No

Yes

Done

Increment Address Command

Increment Address Command

Increment Address Command

Increment Address Command

Address = 0x2004?

No

Yes

Increment Address Command

Increment Address Command
2.3.1.3 LOAD DATA FOR DATA MEMORY

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied. However, the data memory is only 8-bits wide, and thus only the first 8-bits of data after the start bit will be programmed into the data memory. It is still necessary to cycle the clock the full 16 cycles in order to allow the internal circuitry to reset properly. The data memory contains 64 words. Only the lower 8-bits of the PC are decoded by the data memory, and therefore if the PC is greater than 0x3F, it will wrap around and address a location within the physically implemented memory.

2.3.1.4 READ DATA FROM PROGRAM MEMORY

After receiving this command, the chip will transmit data bits out of the program memory (user or configuration) currently accessed starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 5-2.

2.3.1.5 READ DATA FROM DATA MEMORY

After receiving this command, the chip will transmit data bits out of the data memory starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. As previously stated, the data memory is 8-bits wide, and therefore, only the first 8-bits that are output are actual data.

2.3.1.6 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.3.1.7 BEGIN ERASE/PROGRAM CYCLE

A load command must be given before every begin programming command. Programming of the appropriate memory (test program memory, user program memory or data memory) will begin after this command is received and decoded. An internal timing mechanism executes an erase before write. The user must allow for both erase and programming cycle times for programming to complete. No “end programming” command is required.

2.3.1.8 BEGIN PROGRAMMING

A load command must be given before every begin programming command. Programming of the appropriate memory (test program memory, user program memory or data memory) will begin after this command is received and decoded. An internal timing mechanism executes a write. The user must allow for program cycle time for programming to complete. No “end programming” command is required.

This command is similar to the ERASE/PROGRAM CYCLE command, except that a word erase is not done. It is recommended that a bulk erase be performed before starting a series of programming only cycles.

2.3.1.9 BULK ERASE PROGRAM MEMORY

After this command is performed, the next program command will erase the entire program memory.

To perform a bulk erase of the program memory, the following sequence must be performed.
1. Do a “Load Data All 1’s” command.
2. Do a “Bulk Erase User Memory” command.
3. Do a “Begin Programming” command.
4. Wait 10 ms to complete bulk erase.

If the address is pointing to the test program memory (0x2000 - 0x200F), then both the user memory and the test memory will be erased. The configuration word will not be erased, even if the address is pointing to location 0x2007.

For PIC16F84 perform the following commands:
1. Issue Command 2 (write program memory).
2. Send out 3FFFH data.
3. Issue Command 1 (toggle select even rows).
4. Issue Command 7 (toggle select even rows).
5. Issue Command 8 (begin programming)
6. Delay 10 ms
7. Issue Command 1 (toggle select even rows).
8. Issue Command 7 (toggle select even rows).

Note: If the device is code-protected (PIC16F84A), the BULK ERASE command will not work.
2.3.1.10 BULK ERASE DATA MEMORY

To perform a bulk erase of the data memory, the following sequence must be performed.

1. Do a "Load Data All 1's" command.
2. Do a "Bulk Erase Data Memory" command.
3. Do a "Begin Programming" command.
4. Wait 10 ms to complete bulk erase.

For PIC16F84 perform the data memory.

5. Send out 3FFFH data.
6. Issue Command 1 (toggle select even rows).
7. Issue Command 7 (toggle select even rows).
8. Issue Command 8 (begin data)
9. Delay 10 ms
10. Issue Command 1 (toggle select even rows).
    Issue Command 7 (toggle select even rows).

**Note:** All BULK ERASE operations must take place at 4.5 to 5.5 VDD range.

2.4 Programming Algorithm Requires Variable VDD

The PIC16F8X uses an intelligent algorithm. The algorithm calls for program verification at VDDmin. as well as VDDmax. Verification at VDDmin. guarantees good "erase margin". Verification at VDDmax guarantees good "program margin".

The actual programming must be done with VDD in the VDDP range (See Table 5-1).

VDDP = VCC range required during programming.
VDDmin. = minimum operating VDD spec for the part.
VDDmax. = maximum operating VDD spec for the part.

Programmers must verify the PIC16F8X at its specified VDD max. and VDDmin levels. Since Microchip may introduce future versions of the PIC16F8X with a broader VDD range, it is best that these levels are user selectable (defaults are ok).

**Note:** Any programmer not meeting these requirements may only be classified as "prototype" or "development" programmer but not a "production" quality programmer.
3.0 CONFIGURATION WORD

The PIC16F8X has five configuration bits. These bits can be set (reads ‘0’) or left unchanged (reads ‘1’) to select various device configurations.

3.1 Device ID Word

The device ID word for the PIC16F8XX is located at 2006h.

<table>
<thead>
<tr>
<th>Device</th>
<th>Device ID Value</th>
<th>Dev</th>
<th>Rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F84A</td>
<td>00 0101 010</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>PIC16F877</td>
<td>00 1001 101</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 3-1: CONFIGURATION WORD BIT MAP FOR PIC16F83/CR83/F84/CR84/F84A

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F83/F84/F84A</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>PWRTE</td>
<td>WDTE</td>
<td>FOSC1</td>
<td>FOSC0</td>
<td></td>
</tr>
<tr>
<td>PIC16CR83/CR84</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>DP</td>
<td>CP</td>
<td>CP</td>
<td>CP</td>
<td>PWRTE</td>
<td>WDTE</td>
<td>FOSC1</td>
<td>FOSC0</td>
<td></td>
</tr>
</tbody>
</table>

bit 4-13: **CP**, Code Protection Configuration Bits
- 1 = code protection off
- 0 = code protection on

bit 7: **PIC16CR83/CR84 only**
- **DP**, Data Memory Code Protection Bit
  - 1 = code protection off
  - 0 = data memory is code protected

bit 3: **PWRTE**, Power Up Timer Enable Configuration Bit
- 1 = Power up timer disabled
- 0 = Power up timer enabled

bit 2: **WDTE**, WDT Enable Configuration Bits
- 1 = WDT enabled
- 0 = WDT disabled

bit 1-0 **FOSC<1:0>**, Oscillator Selection Configuration Bits
- 11: RC oscillator
- 10: HS oscillator
- 01: XT oscillator
- 00: LP oscillator
FIGURE 3-2: CONFIGURATION WORD FOR PIC16F877

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>BKBUG</td>
<td>Background Debugger Mode (This bit documented as reserved in data sheet)</td>
</tr>
<tr>
<td>11</td>
<td>CP1:CP0</td>
<td>Flash Program Memory Code Protection bits</td>
</tr>
<tr>
<td>10</td>
<td>LVP</td>
<td>Low voltage programming Enable bit</td>
</tr>
<tr>
<td>9</td>
<td>BODEN</td>
<td>Brown-out Reset Enable bit</td>
</tr>
<tr>
<td>8</td>
<td>PWRT</td>
<td>Power-up Timer Enable bit</td>
</tr>
<tr>
<td>7</td>
<td>WDTE</td>
<td>Watchdog Timer Enable bit</td>
</tr>
<tr>
<td>6</td>
<td>FOSC1:FOSC0</td>
<td>Oscillator Selection bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Register:</strong> CONFIG</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Address:</strong> 2007h</td>
</tr>
</tbody>
</table>

- **BKBUG**: Background Debugger Mode (This bit documented as reserved in data sheet)
  - 1 = Background debugger functions not enabled
  - 0 = Background debugger functional.

- **CP1:CP0**: Flash Program Memory Code Protection bits
  - 11 = Code protection off
  - 10 = 1F00h to 1FFFh code protected
  - 01 = 1000h to 1FFFh code protected
  - 00 = 0000h to 1FFFh code protected

- **LVP**: Low voltage programming Enable bit
  - 1 = RB3/PGM pin has PGM function, low voltage programming enabled
  - 0 = RB3 is digital I/O, HV on MCLR must be used for programming

- **BODEN**: Brown-out Reset Enable bit
  - 1 = BOR enabled
  - 0 = BOR disabled

- **PWRT**: Power-up Timer Enable bit
  - 1 = PWRT enabled
  - 0 = PWRT disabled

- **WDTE**: Watchdog Timer Enable bit
  - 1 = WDT enabled
  - 0 = WDT disabled

- **FOSC1:FOSC0**: Oscillator Selection bits
  - 11 = RC oscillator
  - 10 = HS oscillator
  - 01 = XT oscillator
  - 00 = LP oscillator

**Note**: Enabling Brown-out Reset automatically enables Power-up Timer (PWRT) regardless of the value of bit PWRT.

Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

2: All of the CP1:CP0 pairs have to be given the same value to enable the code protection scheme listed.
4.0 CODE PROTECTION
For PIC16F8X devices, once code protection is enabled, all program memory locations read all 0’s. The ID locations and the configuration word read out in an unscrambled fashion. Further programming is disabled for the entire program memory as well as data memory. It is possible to program the ID locations and the configuration word.

4.1 Disabling Code-Protection
It is recommended that the following procedure be performed before any other programming is attempted. It is also possible to turn code protection off (code protect bit = 1) using this procedure; however, all data within the program memory and the data memory will be erased when this procedure is executed, and thus, the security of the data or code is not compromised.

Procedure to disable code protect:
a) Execute load configuration (with a ‘1’ in bit 4, code protect).
b) Increment to configuration word location (0x2007)
c) Execute command (000001)
d) Execute command (000111)
e) Execute ‘Begin Programming’ (001000)
f) Wait 10 ms
g) Execute command (000001)
h) Execute command (000111)

4.2 Embedding Configuration Word and ID Information in the Hex File
To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.
Specifically for the PIC16F8X, the EEPROM data memory should also be embedded in the hex file (see Section 5.1). Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.

| TABLE 4-1: CONFIGURATION WORD
PIC16F83
To code protect: 0000000000XXXX |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Memory Segment</td>
<td>R/W in Protected Mode</td>
<td>R/W in Unprotected Mode</td>
</tr>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>All memory</td>
<td>Read All 0’s, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations [0x2000 : 0x2003]</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

PIC16CR83
To code protect: 0000000000XXXX

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled</td>
<td>Read Unscrambled</td>
</tr>
<tr>
<td>All memory</td>
<td>Read All 0’s for Program Memory, Read All 1’s for Data Memory - Write Disabled</td>
<td>Read Unscrambled, Data Memory - Write Enabled</td>
</tr>
<tr>
<td>ID Locations [0x2000 : 0x2003]</td>
<td>Read Unscrambled</td>
<td>Read Unscrambled</td>
</tr>
</tbody>
</table>
### PIC16CR84

To code protect: 0000000000XXXX

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled</td>
<td>Read Unscrambled</td>
</tr>
<tr>
<td>All memory</td>
<td>Read All 0’s for Program Memory, Read All 1’s for Data Memory - Write Disabled</td>
<td>Read Unscrambled, Data Memory - Write Enabled</td>
</tr>
<tr>
<td>ID Locations [0x2000 : 0x2003]</td>
<td>Read Unscrambled</td>
<td>Read Unscrambled</td>
</tr>
</tbody>
</table>

### PIC16F84

To code protect: 0000000000XXXX

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>All memory</td>
<td>Read All 0’s, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations [0x2000 : 0x2003]</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

### PIC16F84A

To code protect: 0000000000XXXX

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>All memory</td>
<td>Read All 0’s, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations [0x2000 : 0x2003]</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

### PIC16F8XX

To code protect: 00X1XXXX00XXXX

<table>
<thead>
<tr>
<th>Program Memory Segment</th>
<th>R/W in Protected Mode</th>
<th>R/W in Unprotected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Word (0x2007)</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>All memory</td>
<td>Read All 0’s, Write Disabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
<tr>
<td>ID Locations [0x2000 : 0x2003]</td>
<td>Read Unscrambled, Write Enabled</td>
<td>Read Unscrambled, Write Enabled</td>
</tr>
</tbody>
</table>

Legend: X = Don’t care
4.3 CHECKSUM COMPUTATION

4.3.1 CHECKSUM

Checksum is calculated by reading the contents of the PIC16F8X memory locations and adding up the opcodes up to the maximum user addressable location, e.g., 0x1FF for the PIC16F8X. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC16F8X devices is shown in Table 4-2.

The checksum is calculated by summing the following:
- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.

### TABLE 4-2: CHECKSUM COMPUTATION

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank Value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F83</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3FFF</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>0x3E0E</td>
<td>0x09DC</td>
</tr>
<tr>
<td>PIC16CR83</td>
<td>OFF</td>
<td>SUM[0x000:0x1FF] + CFGW &amp; 0x3FFF</td>
<td>0x3DFF</td>
<td>0x09CD</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>0x3E0E</td>
<td>0x09DC</td>
</tr>
<tr>
<td>PIC16F84</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3FFF</td>
<td>0x3BFF</td>
<td>0x07CD</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>0x3C0E</td>
<td>0x07DC</td>
</tr>
<tr>
<td>PIC16CR84</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3FFF</td>
<td>0x3BFF</td>
<td>0x07CD</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>0x3C0E</td>
<td>0x07DC</td>
</tr>
<tr>
<td>PIC16F84A</td>
<td>OFF</td>
<td>SUM[0x000:0x3FF] + CFGW &amp; 0x3FFF</td>
<td>0x3BFF</td>
<td>0x07CD</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>CFGW &amp; 0x3FFF + SUM_ID</td>
<td>0x3C0E</td>
<td>0x07DC</td>
</tr>
<tr>
<td>PIC16F877</td>
<td>OFF</td>
<td>SUM[0x0000:0x1FFF]</td>
<td>0x1BFF</td>
<td>0xE7CD</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x0000:0x1EFF] + CFGW &amp; 0x3BFF &amp; SUM_ID</td>
<td>0x28EE</td>
<td>0xDA0A</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x0000:0x1FF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x27DE</td>
<td>0xD993</td>
</tr>
<tr>
<td></td>
<td>ON</td>
<td>SUM[0x0000:0x0FFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x27CE</td>
<td>0xF39C</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word

SUM[a:b] = [Sum of locations a to b inclusive]

SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble.

For example, ID0 = 0x1, ID1 = 0x2, ID3 = 0x3, ID4 = 0x4, then SUM_ID = 0x1234

*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]

+ = Addition
& = Bitwise AND
5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

5.1 Embedding Data EEPROM Contents in Hex File

The programmer should be able to read data EEPROM information from a hex file and conversely (as an option) write data EEPROM contents to a hex file along with program memory information and fuse information.

The 64 data memory locations are logically mapped starting at address 0x2100. The format for data memory storage is one data byte per address location, LSB aligned.

TABLE 5-1: AC/DC CHARACTERISTICS
TIMING REQUIREMENTS FOR PROGRAM/VERIFY TEST MODE

<table>
<thead>
<tr>
<th>Standard Operating Conditions</th>
<th>Operating Temperature: +10°C ≤ TA ≤ +40°C, unless otherwise stated, (25°C is recommended)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage:</td>
<td>4.5V ≤ VDD ≤ 5.5V, unless otherwise stated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VDDP</td>
<td>Supply voltage during programming</td>
<td>4.5</td>
<td>5.0</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDDV</td>
<td>Supply voltage during verify</td>
<td>VDDmin</td>
<td>VDDmax</td>
<td></td>
<td>V</td>
<td>Note 1</td>
<td></td>
</tr>
<tr>
<td>VIHH</td>
<td>High voltage on MCLR for test mode entry</td>
<td>12</td>
<td>14.0</td>
<td></td>
<td>V</td>
<td>Note 2</td>
<td></td>
</tr>
<tr>
<td>IDDp</td>
<td>Supply current (from Vdd) during program/verify</td>
<td></td>
<td></td>
<td>50</td>
<td>mA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHH</td>
<td>Supply current from VIHH (on MCLR)</td>
<td></td>
<td></td>
<td>200</td>
<td>μA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIH1</td>
<td>(RB6, RB7) input high level</td>
<td>0.8</td>
<td>VDD</td>
<td></td>
<td>V</td>
<td>Schmitt Trigger input</td>
<td></td>
</tr>
<tr>
<td>VIH1</td>
<td>(RB6, RB7) input low level MCLR</td>
<td>0.2</td>
<td>VDD</td>
<td></td>
<td>V</td>
<td>Schmitt Trigger input</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>TVHHR</td>
<td>MCLR rise time (VSS to VHH) for test mode entry</td>
<td>8.0</td>
<td></td>
<td></td>
<td>μs</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>Tset0</td>
<td>RB6, RB7 setup time (before pattern setup time)</td>
<td>100</td>
<td></td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>Tset1</td>
<td>Data in setup time before clock ↓</td>
<td>100</td>
<td></td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>Thld1</td>
<td>Data in hold time after clock ↓</td>
<td>100</td>
<td></td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>Tdly1</td>
<td>Data input not driven to next clock input (delay required between command/data or command/command)</td>
<td>1.0</td>
<td></td>
<td></td>
<td>μs</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>Tdly2</td>
<td>Delay between clock ↓ to clock ↑ of next command or data</td>
<td>1.0</td>
<td></td>
<td></td>
<td>μs</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>Tdly3</td>
<td>Clock to data out valid (during read data)</td>
<td>80</td>
<td></td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>Thld0</td>
<td>RB &lt;7:6&gt; hold time after MCLR ↑</td>
<td>100</td>
<td></td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>ms</td>
<td></td>
</tr>
</tbody>
</table>

Note 1: Program must be verified at the minimum and maximum VDD limits for the part.
Note 2: VIHH must be greater than VDD + 4.5V to stay in programming/verify mode.
FIGURE 5-1: LOAD DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-2: READ DATA COMMAND (PROGRAM/VERIFY)

FIGURE 5-3: INCREMENT ADDRESS COMMAND (PROGRAM/VERIFY)
This document includes the programming specifications for the following devices:

- PIC16F870
- PIC16F871
- PIC16F872
- PIC16F873
- PIC16F874
- PIC16F876
- PIC16F877
- PIC16F878

1.0 PROGRAMMING THE PIC16F8XX

The PIC16F8XX is programmed using a serial method. The serial mode will allow the PIC16F8XX to be programmed while in the users system. This allows for increased design flexibility. This programming specification applies to PIC16F8XX devices in all packages.

PIC16F8XX devices may be programmed using a single +5 volt supply (low voltage programming mode).

1.1 Hardware Requirements

The PIC16F8XX requires one programmable power supply for VDD (4.5V to 5.5V) and a VPP of 12V to 14V or VPP of (4.5V to 5.5V) when using low voltage In-Circuit Serial Programming™ (ICSP™). Both supplies should have a minimum resolution of 0.25V.

1.2 Programming Mode

The programming mode for the PIC16F8XX allows programming of user program memory, data memory, special locations used for ID, and the configuration word.

**PIN DESCRIPTIONS (DURING PROGRAMMING): PIC16F8XX**

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Function</th>
<th>Pin Type</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB3</td>
<td>PGM</td>
<td>I</td>
<td>Low voltage ICSP programming input if configuration bit equals 1</td>
</tr>
<tr>
<td>RB6</td>
<td>CLOCK</td>
<td>I</td>
<td>Clock input</td>
</tr>
<tr>
<td>RB7</td>
<td>DATA</td>
<td>I/O</td>
<td>Data input/output</td>
</tr>
<tr>
<td>MCLR</td>
<td>VTEST MODE</td>
<td>I/O</td>
<td>Program Mode Select</td>
</tr>
<tr>
<td>VDD</td>
<td>VDD</td>
<td>P*</td>
<td>Power Supply</td>
</tr>
<tr>
<td>VSS</td>
<td>VSS</td>
<td>P</td>
<td>Ground</td>
</tr>
</tbody>
</table>

*In the PIC16F8XX, the programming high voltage is internally generated. To activate the programming mode, high voltage needs to be applied to MCLR input. Since the MCLR is used for a level source, this means that MCLR does not draw any significant current.*
2.0 PROGRAM MODE ENTRY

2.1 User Program Memory Map

The user memory space extends from 0x0000 to 0x1FFF (8K). In programming mode the program memory space extends from 0x0000 to 0x3FFF, with the first half (0x0000-0x1FFF) being user program memory and the second half (0x2000-0x3FFF) being configuration memory. The PC will increment from 0x0000 to 0x1FFF and wrap to 0x000, 0x2000 to 0x3FFF and wrap around to 0x2000 (not to 0x0000). Once in configuration memory, the highest bit of the PC stays a ‘1’, thus always pointing to the configuration memory. The only way to point to user program memory is to reset the part and reenter program/verify mode as described in Section 2.3.

In the configuration memory space, 0x2000-0x200F are physically implemented. However, only locations 0x2000 through 0x2007 are available. Other locations are reserved. Locations beyond 0x200F will physically access user memory. (See Figure 2-1).

2.2 ID Locations

A user may store identification information (ID) in four ID locations. The ID locations are mapped in [0x2000 : 0x2003]. It is recommended that the user use only the four least significant bits of each ID location. In some devices, the ID locations read-out in an unscrambled fashion after code protection is enabled. For these devices, it is recommended that ID location is written as “11 1111 1000 bbbb” where ‘bbbb’ is ID information.

In other devices, the ID locations read out normally, even after code protection. To understand how the devices behave, refer to Table 4-1.

To understand the scrambling mechanism after code protection, refer to Section 4.0.
FIGURE 2-1: PROGRAM MEMORY MAPPING

<table>
<thead>
<tr>
<th>Address</th>
<th>ID Location</th>
<th>2K words</th>
<th>4K words</th>
<th>8K words</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Reserved</td>
</tr>
<tr>
<td>2001h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Reserved</td>
</tr>
<tr>
<td>2002h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Reserved</td>
</tr>
<tr>
<td>2003h</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Implemented</td>
<td>Reserved</td>
</tr>
<tr>
<td>2004h</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2005h</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2006h</td>
<td>Device ID</td>
<td>Reserved</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2007h</td>
<td>Configuration Word</td>
<td>Reserved</td>
<td>Implemented</td>
<td>Implemented</td>
</tr>
<tr>
<td>2008h</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>2100h</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>3FFFh</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
2.3 Program/Verify Mode

The program/verify mode is entered by holding pins RB6 and RB7 low while raising MCLR pin from VIL to VIHH (high voltage). In this mode, the state of the RB3 pin does not affect programming. Low-voltage ICSP programming mode is entered by applying VDD to MCLR and raising RB3 from VIL to VDD. Once in this mode the user program memory and the configuration memory can be accessed and programmed in serial fashion. The mode of operation is serial, and the memory that is accessed is the user program memory. RB6 and RB7 are Schmitt Trigger Inputs in this mode.

Note: The OSC must not have 72 osc clocks while the device MCLR is between VIL and VIHH.

The sequence that enters the device into the programming/verify mode places all other logic into the reset state (the MCLR pin was initially at VIL). This means that all I/O are in the reset state (High impedance inputs).

The normal sequence for programming is to use the load data command to set a value to be written at the selected address. Issue the begin programming command followed by read data command to verify, and then increment the address.

A device reset will clear the PC and set the address to 0. The “increment address” command will increment the PC. The “load configuration” command will set the PC to 0x2000. The available commands are shown in Table 2-1.

2.3.1 LOW-VOLTAGE ICSP PROGRAMMING MODE

When LVP bit is set to ‘1’, the low-voltage ICSP programming entry is enabled. Since the LVP configuration bit allows low voltage ICSP programming entry in its erased state, an erased device will have the LVP bit enabled at the factory. While LVP is ‘1’, RB3 is dedicated to low voltage ICSP programming. Bring MCLR to VDD and then RB3 to VDD to enter programming mode. All other specifications for high-voltage ICSP™ apply.

To disable low voltage ICSP mode, the LVP bit must be programmed to ‘0’. This must be done while entered with high voltage entry mode (LVP bit = 1). RB3 is now a general purpose I/O pin.

2.3.2 SERIAL PROGRAM/VERIFY OPERATION

The RB6 pin is used as a clock input pin, and the RB7 pin is used for entering command bits and data input/output during serial operation. To input a command, the clock pin (RB6) is cycled six times. Each command bit is latched on the falling edge of the clock with the least significant bit (LSB) of the command being input first. The data on pin RB7 is required to have a minimum setup and hold time (see AC/DC specifications) with respect to the falling edge of the clock. Commands that have data associated with them (read and load) are specified to have a minimum delay of 1 µs between the command and the data. After this delay, the clock pin is cycled 16 times with the first cycle being a start bit and the last cycle being a stop bit. Data is also input and output LSB first.

Therefore, during a read operation the LSB will be transmitted onto pin RB7 on the rising edge of the second cycle, and during a load operation the LSB will be latched on the falling edge of the second cycle. A minimum 1µs delay is also specified between consecutive commands.

All commands are transmitted LSB first. Data words are also transmitted LSB first. The data is transmitted on the rising edge and latched on the falling edge of the clock. To allow for decoding of commands and reversal of data pin configuration, a time separation of at least 1 µs is required between a command and a data word (or another command).

The commands that are available are:

2.3.2.1 LOAD CONFIGURATION

After receiving this command, the program counter (PC) will be set to 0x2000. By then applying 16 cycles to the clock pin, the chip will load 14-bits in a “data word,” as described above, to be programmed into the configuration memory. A description of the memory mapping schemes of the program memory for normal operation and configuration mode operation is shown in Figure 2-1. After the configuration memory is entered, the only way to get back to the user program memory is to exit the program/verify test mode by taking MCLR low (VIL).
2.3.2.2 LOAD DATA FOR PROGRAM MEMORY

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied, as described previously. A timing diagram for the load data command is shown in Figure 5-1.

<table>
<thead>
<tr>
<th>Command</th>
<th>Mapping (MSB … LSB)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Configuration</td>
<td>X X 0 0 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Load Data for Program Memory</td>
<td>X X 0 0 1 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Program Memory</td>
<td>X X 0 1 0 0</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Increment Address</td>
<td>X X 0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>Begin Erase Programming Cycle</td>
<td>0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Begin Programming Only Cycle</td>
<td>0 1 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Load Data for Data Memory</td>
<td>X X 0 0 1 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Read Data from Data Memory</td>
<td>X X 0 1 0 1</td>
<td>0, data (14), 0</td>
</tr>
<tr>
<td>Bulk Erase Program Memory</td>
<td>X X 1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>Bulk Erase Data Memory</td>
<td>X X 1 0 1 1</td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 2-2: PROGRAM FLOW CHART - PIC16F8XX PROGRAM MEMORY

Start
- Set VDD = VDDP

Program Cycle
- Read Data Command
  - Data Correct?
    - No → Report Programming Failure
    - Yes → Increment Address Command

All Locations Done?
- No → Verify all Locations @ VDDMIN
  - Data Correct?
    - No → Report Verify Error @ VDDMIN
    - Yes → Verify all Locations @ VDDMAX
  - Data Correct?
    - No → Report Verify Error @ VDDMAX
    - Yes → Done
- Yes → Wait tprog
FIGURE 2-3: PROGRAM FLOW CHART - PIC16F8XX CONFIGURATION MEMORY

Start

Load Configuration Data

Program ID Location?

Yes

Program Cycle

Read Data Command

No

Data Correct?

Yes

Report Programming Failure

No

Data Correct?

Set VDD = VDDMAX

Read Data Command

Increment Address Command

Address = 0x2004? Yes

Increment Address Command

Increment Address Command

Increment Address Command

Program Cycle (Config. Word)

Report Program Configuration Word Error

Data Correct?

No

Set VDD = VDDMAX

Yes

Read Data Command

Done

Data Correct?

No

Yes

Read Data Command
2.3.2.3 LOAD DATA FOR DATA MEMORY

After receiving this command, the chip will load in a 14-bit “data word” when 16 cycles are applied. However, the data memory is only 8-bits wide, and thus only the first 8-bits of data after the start bit will be programmed into the data memory. It is still necessary to cycle the clock the full 16 cycles in order to allow the internal circuitry to reset properly. The data memory contains 64 words. Only the lower 8-bits of the PC are decoded by the data memory, and therefore if the PC is greater than 0x3F, it will wrap around and address a location within the physically implemented memory. If the device is code protected, the data is read as all zeros.

2.3.2.4 READ DATA FROM PROGRAM MEMORY

After receiving this command, the chip will transmit data bits out of the program memory (user or configuration) currently accessed starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. A timing diagram of this command is shown in Figure 5-2.

2.3.2.5 READ DATA FROM DATA MEMORY

After receiving this command, the chip will transmit data bits out of the data memory starting with the second rising edge of the clock input. The RB7 pin will go into output mode on the second rising clock edge, and it will revert back to input mode (hi-impedance) after the 16th rising edge. As previously stated, the data memory is 8-bits wide, and therefore, only the first 8-bits that are output are actual data.

2.3.2.6 INCREMENT ADDRESS

The PC is incremented when this command is received. A timing diagram of this command is shown in Figure 5-3.

2.3.2.7 BEGIN ERASE/PROGRAM CYCLE

A load command must be given before every begin programming command. Programming of the appropriate memory (test program memory, user program memory or data memory) will begin after this command is received and decoded. An internal timing mechanism executes a write. The user must allow for program cycle time for programming to complete. No “end programming” command is required.

2.3.2.8 BEGIN PROGRAMMING

A load command must be given before every begin programming command. Programming of the appropriate memory (test program memory, user program memory or data memory) will begin after this command is received and decoded. An internal timing mechanism executes a write. The user must allow for program cycle time for programming to complete. No "end programming" command is required.

This command is similar to the ERASE/PROGRAM CYCLE command, except that a word erase is not done. It is recommended that a bulk erase be performed before starting a series of programming only cycles.

2.3.2.9 BULK ERASE PROGRAM MEMORY

After this command is performed, the next program command will erase the entire program memory.

To perform a bulk erase of the program memory, the following sequence must be performed.
1. Do a “Load Data All 1’s” command.
2. Do a “Bulk Erase Program Memory” command.
3. Do a “Begin Programming” command.
4. Wait 10 ms to complete bulk erase.

If the address is pointing to the test program memory (0x2000 - 0x200F), then both the user memory and the test memory will be erased. The configuration word will not be erased, even if the address is pointing to location 0x0207.

Note: If the device is code-protected, the BULK ERASE command will not work.

2.3.2.10 BULK ERASE DATA MEMORY

To perform a bulk erase of the data memory, the following sequence must be performed.
1. Do a “Load Data All 1’s” command.
2. Do a “Bulk Erase Data Memory” command.
3. Do a “Begin Programming” command.
4. Wait 10 ms to complete bulk erase.

Note: All BULK ERASE operations must take place at 4.5 to 5.5 V_{DD} range.
2.4 Programming Algorithm Requires Variable VDD

The PIC16F8XX uses an intelligent algorithm. The algorithm calls for program verification at VDD_min. as well as VDD_max. Verification at VDD_min. guarantees good “erase margin”. Verification at VDD_max guarantees good “program margin”.

The actual programming must be done with VDD in the VDD_P range (See Table 5-1).

VDD_P = VCC range required during programming.
VDD_min. = minimum operating VDD spec for the part.
VDD_max. = maximum operating VDD spec for the part.

Programmers must verify the PIC16F8XX at its specified VDD max. and VDD_min levels. Since Microchip may introduce future versions of the PIC16F8XX with a broader VDD range, it is best that these levels are user selectable (defaults are ok).

Note: Any programmer not meeting these requirements may only be classified as “prototype” or “development” programmer but not a “production” quality programmer.
3.0 CONFIGURATION WORD

The PIC16F8XX has several configuration bits. These bits can be set (reads '0') or left unchanged (reads '1') to select various device configurations.

3.1 Device ID Word

The device ID word for the PIC16F8XX is located at 2006h.

<table>
<thead>
<tr>
<th>Device ID Value</th>
<th>Device ID Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev</td>
<td>Rev</td>
</tr>
<tr>
<td>PIC16F870</td>
<td>00 1101 000</td>
</tr>
<tr>
<td>PIC16F871</td>
<td>00 1101 001</td>
</tr>
<tr>
<td>PIC16F872</td>
<td>00 1000 111</td>
</tr>
<tr>
<td>PIC16F873</td>
<td>00 1001 011</td>
</tr>
<tr>
<td>PIC16F874</td>
<td>00 1001 001</td>
</tr>
<tr>
<td>PIC16F876</td>
<td>00 1001 111</td>
</tr>
<tr>
<td>PIC16F877</td>
<td>00 1001 101</td>
</tr>
</tbody>
</table>

FIGURE 3-1: CONFIGURATION WORD FOR PIC16F873/874/876/877

<table>
<thead>
<tr>
<th>CP1</th>
<th>CP0</th>
<th>RESV</th>
<th>-</th>
<th>WRT</th>
<th>CPD</th>
<th>LVP</th>
<th>BODEN</th>
<th>CP1</th>
<th>CP0</th>
<th>PWRT</th>
<th>WDTE</th>
<th>F0SC1</th>
<th>F0SC0</th>
<th>Register</th>
<th>CONFIG Address</th>
<th>2007h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bit 13-12:

bit 11: Reserved: Set to '1' for normal operation

bit 5-4: CP1:CP0: Flash Program Memory Code Protection bits (2)

4K Devices:

11 = Code protection off
10 = not supported
01 = not supported
00 = 0000h to 0FFFh code protected

8K Devices:

11 = Code protection off
10 = 1F00h to 1FFFh code protected
01 = 1000h to 1FFFh code protected
00 = 0000h to 1FFFh code protected

bit 11: Reserved: Set to '1' for normal operation

bit 10: Unimplemented: Read as '1'

bit 9: WRT: Flash Program Memory Write Enable

1 = Unprotected program memory may be written to by EECON control
0 = Unprotected program memory may not be written to by EECON control

bit 8: CPD: Data EE Memory Code Protection

1 = Code protection off
0 = Data EE memory code protected

bit 7: LVP: Low voltage programming Enable bit

1 = RB3/PGM pin has PGM function, low voltage programming enabled
0 = RB3 is digital I/O, HV on MCLR must be used for programming

bit 6: BODEN: Brown-out Reset Enable bit (1)

1 = BOR enabled
0 = BOR disabled

bit 3: PWRT: Power-up Timer Enable bit (1)

1 = PWRT disabled
0 = PWRT enabled

bit 2: WDTE: Watchdog Timer Enable bit

1 = WDT enabled
0 = WDT disabled

bit 1-0: F0SC1:F0SC0: Oscillator Selection bits

11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

Note 1: Enabling Brown-out Reset automatically enables Power-up Timer (PWRT) regardless of the value of bit PWRT. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

2: All of the CP1:CP0 pairs have to be given the same value to enable the code protection scheme listed.
FIGURE 3-2: CONFIGURATION WORD FOR PIC16F870/871/872

<table>
<thead>
<tr>
<th>CP1</th>
<th>CP0</th>
<th>RESV</th>
<th>WRT</th>
<th>CPD</th>
<th>LVP</th>
<th>BODEN</th>
<th>CP1</th>
<th>CP0</th>
<th>PWRT</th>
<th>WDTE</th>
<th>FOSC1</th>
<th>FOSC0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit13</td>
<td>bit12</td>
<td>bit 5-4</td>
<td>bit 11</td>
<td>bit 10</td>
<td>bit 9</td>
<td>bit 8</td>
<td>bit 7</td>
<td>bit 6</td>
<td>bit 5</td>
<td>bit 4</td>
<td>bit 3</td>
<td>bit 2</td>
</tr>
</tbody>
</table>

- **bit 13**: Reserved
- **bit 12**: Read as ‘1’
- **bit 5-4**: **CP1:CP0**: Flash Program Memory Code Protection bits
  - 11 = Code protection off
  - 10 = not supported
  - 01 = not supported
  - 00 = 0000h to 07FFh code protected
- **bit 11**: Reserved: Set to ‘1’ for normal operation
- **bit 10**: Unimplemented: Read as ‘1’
- **bit 9**: **WRT**: Flash Program Memory Write Enable
  - 1 = Unprotected program memory may be written to by EECON control
  - 0 = Unprotected program memory may not be written to by EECON control
- **bit 8**: **CPD**: Data EE Memory Code Protection
  - 1 = Code protection off
  - 0 = Data EE memory code protected
- **bit 7**: **LVP**: Low voltage programming Enable bit
  - 1 = RB3/PGM pin has PGM function, low voltage programming enabled
  - 0 = RB3 is digital I/O, HV on MCLR must be used for programming
- **bit 6**: **BODEN**: Brown-out Reset Enable bit
  - 1 = BOR enabled
  - 0 = BOR disabled
- **bit 5**: **PWRT**: Power-up Timer Enable bit
  - 1 = PWRT disabled
  - 0 = PWRT enabled
- **bit 4**: **WDTE**: Watchdog Timer Enable bit
  - 1 = WDT enabled
  - 0 = WDT disabled
- **bit 3-0**: **FOSC1:FOSC0**: Oscillator Selection bits
  - 11 = RC oscillator
  - 10 = HS oscillator
  - 01 = XT oscillator
  - 00 = LP oscillator

**Note 1**: Enabling Brown-out Reset automatically enables Power-up Timer (PWRT) regardless of the value of bit **PWRT**. Ensure the Power-up Timer is enabled anytime Brown-out Reset is enabled.

**Note 2**: All of the CP1:CP0 pairs have to be given the same value to enable the code protection scheme listed.
4.0 CODE PROTECTION

For PIC16F8XX devices, once code protection is enabled, all program memory locations read all 0’s. The ID locations and the configuration word read out in an unscrambled fashion. Further programming is disabled for the entire program memory as well as data memory. It is possible to program the ID locations and the configuration word.

4.1 Disabling Code-Protection

It is recommended that the following procedure be performed before any other programming is attempted. It is also possible to turn code protection off (code protect bit = 1) using this procedure; however, all data within the program memory and the data memory will be erased when this procedure is executed, and thus, the security of the data or code is not compromised.

Procedure to disable code protect:

a) Execute load configuration (with a ’1’ in bit 13-4, code protect).

b) Increment to configuration word location (0x2007)

c) Execute command (000001)

d) Execute command (000111)

e) Execute ‘Begin Programming’ (001000)

f) Wait 12 ms

g) Execute command (000001)

h) Execute command (000111)

4.2 Embedding Configuration Word and ID Information in the Hex File

To allow portability of code, the programmer is required to read the configuration word and ID locations from the hex file when loading the hex file. If configuration word information was not present in the hex file then a simple warning message may be issued. Similarly, while saving a hex file, configuration word and ID information must be included. An option to not include this information may be provided.

Specifically for the PIC16F8XX, the EEPROM data memory should also be embedded in the hex file (see Section 5.1).

Microchip Technology Inc. feels strongly that this feature is important for the benefit of the end customer.
4.3 CHECKSUM COMPUTATION

4.3.1 CHECKSUM

Checksum is calculated by reading the contents of the PIC16F8XX memory locations and adding up the opcodes up to the maximum user addressable location, e.g., 0x1FF for the PIC16F8XX. Any carry bits exceeding 16-bits are neglected. Finally, the configuration word (appropriately masked) is added to the checksum. Checksum computation for each member of the PIC16F8XX devices is shown in Table 4-1.

The checksum is calculated by summing the following:

- The contents of all program memory locations
- The configuration word, appropriately masked
- Masked ID locations (when applicable)

The least significant 16 bits of this sum is the checksum.

The following table describes how to calculate the checksum for each device. Note that the checksum calculation differs depending on the code protect setting. Since the program memory locations read out differently depending on the code protect setting, the table describes how to manipulate the actual program memory values to simulate the values that would be read from a protected device. When calculating a checksum by reading a device, the entire program memory can simply be read and summed. The configuration word and ID locations can always be read.

Note that some older devices have an additional value added in the checksum. This is to maintain compatibility with older device programmer checksums.
## TABLE 4-1: CHECKSUM COMPUTATION

<table>
<thead>
<tr>
<th>Device</th>
<th>Code Protect</th>
<th>Checksum*</th>
<th>Blank&quot;V value</th>
<th>0x25E6 at 0 and max address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC16F870</td>
<td>OFF</td>
<td>SUM[0x0000:0x07FFF] + CFGW &amp; 0x3BFF</td>
<td>0x33FF</td>
<td>0xFFCD</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x3FCE</td>
<td>0x0B9C</td>
</tr>
<tr>
<td>PIC16F871</td>
<td>OFF</td>
<td>SUM[0x0000:0x07FFF] + CFGW &amp; 0x3BFF</td>
<td>0x33FF</td>
<td>0xFFCD</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x3FCE</td>
<td>0x0B9C</td>
</tr>
<tr>
<td>PIC16F872</td>
<td>OFF</td>
<td>SUM[0x0000:0x07FFF] + CFGW &amp; 0x3BFF</td>
<td>0x33FF</td>
<td>0xFFCD</td>
</tr>
<tr>
<td></td>
<td>ALL</td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x3FCE</td>
<td>0x0B9C</td>
</tr>
<tr>
<td>PIC16F873</td>
<td>OFF</td>
<td>SUM[0x0000:0x0FFF] + CFGW &amp; 0x3BFF</td>
<td>0x2BFF</td>
<td>0xF7CD</td>
</tr>
<tr>
<td>0x0F00 : 0xFFF</td>
<td></td>
<td>SUM[0x0000:0x0EFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x48EE</td>
<td>0xFAA3</td>
</tr>
<tr>
<td>0x0800 : 0xFFF</td>
<td></td>
<td>SUM[0x0000:0x07FFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x3FDE</td>
<td>0xF193</td>
</tr>
<tr>
<td>ALL</td>
<td></td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x37CE</td>
<td>0x039C</td>
</tr>
<tr>
<td>PIC16F874</td>
<td>OFF</td>
<td>SUM[0x0000:0x0FFF] + CFGW &amp; 0x3BFF</td>
<td>0x2BFF</td>
<td>0xF7CD</td>
</tr>
<tr>
<td>0x0F00 : 0xFFF</td>
<td></td>
<td>SUM[0x0000:0x0EFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x48EE</td>
<td>0xFAA3</td>
</tr>
<tr>
<td>0x0800 : 0xFFF</td>
<td></td>
<td>SUM[0x0000:0x07FFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x3FDE</td>
<td>0xF193</td>
</tr>
<tr>
<td>ALL</td>
<td></td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x37CE</td>
<td>0x039C</td>
</tr>
<tr>
<td>PIC16F876</td>
<td>OFF</td>
<td>SUM[0x0000:0x1FFF] + CFGW &amp; 0x3BFF</td>
<td>0x1BFF</td>
<td>0xE7CD</td>
</tr>
<tr>
<td>0x1F00 : 0x1FFF</td>
<td></td>
<td>SUM[0x0000:0x1EFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x28EE</td>
<td>0xDA93</td>
</tr>
<tr>
<td>0x1000 : 0x1FFF</td>
<td></td>
<td>SUM[0x0000:0x0FFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x27DE</td>
<td>0xD993</td>
</tr>
<tr>
<td>ALL</td>
<td></td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x27CE</td>
<td>0xF39C</td>
</tr>
<tr>
<td>PIC16F877</td>
<td>OFF</td>
<td>SUM[0x0000:0x1FFF] + CFGW &amp; 0x3BFF</td>
<td>0x1BFF</td>
<td>0xE7CD</td>
</tr>
<tr>
<td>0x1F00 : 0x1FFF</td>
<td></td>
<td>SUM[0x0000:0x1EFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x28EE</td>
<td>0xDA93</td>
</tr>
<tr>
<td>0x1000 : 0x1FFF</td>
<td></td>
<td>SUM[0x0000:0x0FFF] + CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x27DE</td>
<td>0xD993</td>
</tr>
<tr>
<td>ALL</td>
<td></td>
<td>CFGW &amp; 0x3BFF + SUM_ID</td>
<td>0x27CE</td>
<td>0xF39C</td>
</tr>
</tbody>
</table>

Legend: CFGW = Configuration Word  
SUM[a:b] = [Sum of locations a to b inclusive]  
SUM_ID = ID locations masked by 0xF then made into a 16-bit value with ID0 as the most significant nibble.  
For example, ID0 = 0x1, ID1 = 0x2, ID3 = 0x3, ID4 = 0x4, then SUM_ID = 0x1234  
*Checksum = [Sum of all the individual expressions] MODULO [0xFFFF]  
+ = Addition  
& = Bitwise AND
### 5.0 PROGRAM/VERIFY MODE ELECTRICAL CHARACTERISTICS

#### 5.1 Embedding Data EEPROM Contents in Hex File

The programmer should be able to read data EEPROM information from a hex file and conversely (as an option) write data EEPROM contents to a hex file along with program memory information and fuse information.

The 256 data memory locations are logically mapped starting at address 0x2100. The format for data memory storage is one data byte per address location, LSB aligned.

**TABLE 5-1: AC/DC CHARACTERISTICS**

**TIMING REQUIREMENTS FOR PROGRAM/VERIFY MODE**

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Sym</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
<th>Conditions/Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDD level for word operations, program memory</td>
<td>VDD</td>
<td>2.0</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDD level for word operations, data memory</td>
<td>VDD</td>
<td>2.0</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDD level for bulk erase/write operations, program and data memory</td>
<td>VDD</td>
<td>4.5</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High voltage on MCLR for high-voltage programming entry</td>
<td>VihH</td>
<td>VDD + 3.5</td>
<td>13.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voltage on MCLR for low-voltage programming entry</td>
<td>Vih</td>
<td>4.5</td>
<td>5.5</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCLR rise time (VSS to VHH) for test mode entry</td>
<td>tVHHR</td>
<td>1.0</td>
<td>1.0</td>
<td>µs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(RB6, RB7) input high level</td>
<td>ViH1</td>
<td>0.8VDD</td>
<td>V</td>
<td>Schmitt Trigger input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(RB6, RB7) input low level</td>
<td>ViL1</td>
<td>0.2VDD</td>
<td>V</td>
<td>Schmitt Trigger input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB&lt;7:4&gt; setup time before MCLR↑ (test mode selection pattern setup time)</td>
<td>tset0</td>
<td>100</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB&lt;7:4&gt; hold time after MCLR↑ (test mode selection pattern setup time)</td>
<td>thld0</td>
<td>5</td>
<td>µs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Serial Program/Verify</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in setup time before clock↓</td>
<td>tset1</td>
<td>100</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data in hold time after clock↓</td>
<td>thld1</td>
<td>100</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data input not driven to next clock input (delay required between command/data or command/command)</td>
<td>tdly1</td>
<td>1.0</td>
<td>µs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delay between clock↓ to clock↑ of next command or data</td>
<td>tdly2</td>
<td>1.0</td>
<td>µs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock↑ to data out valid (during read data)</td>
<td>tdly3</td>
<td>80</td>
<td>ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erase cycle time</td>
<td>tera</td>
<td>2</td>
<td>5</td>
<td>ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming cycle time</td>
<td>tprog</td>
<td>2</td>
<td>5</td>
<td>ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 5-1: LOAD DATA COMMAND HIGH-VOLTAGE MODE (PROGRAM/VERIFY)

FIGURE 5-2: READ DATA COMMAND HIGH-VOLTAGE MODE (PROGRAM/VERIFY)

FIGURE 5-3: INCREMENT ADDRESS COMMAND HIGH-VOLTAGE MODE (PROGRAM/VERIFY)
FIGURE 5-4: LOAD DATA COMMAND LOW-VOLTAGE MODE (PROGRAM/VERIFY)

FIGURE 5-5: READ DATA COMMAND LOW-VOLTAGE MODE (PROGRAM/VERIFY)

FIGURE 5-6: INCREMENT ADDRESS COMMAND LOW-VOLTAGE MODE (PROGRAM/VERIFY)
SECTION 4
APPLICATION NOTES

IN-CIRCUIT SERIAL PROGRAMMING™ (ICSP™) OF CALIBRATION PARAMETERS USING A PICmicro® MICROCONTROLLER ................................................................. 4-1
INTRODUCTION

Many embedded control applications, where sensor offsets, slopes and configuration information are measured and stored, require a calibration step. Traditionally, potentiometers or Serial EEPROM devices are used to set up and store this calibration information. This application note will show how to construct a programming jig that will receive calibration parameters from the application mid-range PICmicro® microcontrollers (MCU) and program this information into the application baseline PICmicro MCU using the In-Circuit Serial Programming (ICSP) protocol. This method uses the PIC16CXXX In-Circuit Serial Programming algorithm of the 14-bit core microcontrollers.

PROGRAMMING FIXTURE

A programming fixture is needed to assist with the self programming operation. This is typically a small reusable module that plugs into the application PCB being calibrated. Only five pin connections are needed and this programming fixture can draw its power from the application PCB to simplify the connections.
Electrical Interface
There are a total of five electrical connections needed between the application PIC16CXXX microcontroller and the programming jig:

- **MCLR/VPP** - High voltage pin used to place application PIC16CXXX into programming mode
- **VDD** - +5 volt power supply connection to the application PIC16CXXX
- **VSS** - Ground power supply connection to the application PIC16CXXX
- **RB6** - PORTB, bit6 connection to application PIC16CXXX used to clock programming data
- **RB7** - PORTB, bit7 connection to application PIC16CXXX used to send programming data

This programming jig is intended to grab power from the application power supply through the VDD connection. The programming jig will require 100 mA of peak current during programming. The application will need to set RB6 and RB7 as inputs, which means external devices cannot drive these lines. The calibration data will be sent to the programming jig by the application PIC16CXXX through RB6 and RB7. The programming jig will later use these lines to clock the calibration data into the application PIC16CXXX.

Programming Issues
The PIC16CXXX programming specification suggests verification of program memory at both Maximum and Minimum VDD for each device. This is done to ensure proper programming margins and to detect (and reject) any improperly programmed devices. All production quality programmers vary VDD from VDDmin to VDDmax after programming and verify the device under each of these conditions.

Since both the application voltage and its tolerances are known, it is not necessary to verify the PIC16CXXX calibration parameters at the device VDDmax and VDDmin. It is only necessary to verify at the application power supply Max and Min voltages. This application note shows the nominal (+5V) verification routine and hardware. If the power supply is a regulated +5V, this is adequate and no additional hardware or software is needed. If the application power supply is not regulated (such as a battery powered or poorly regulated system) it is important to complete a VDDmin and VDDmax verification cycle following the +5V verification cycle. See programming specifications for more details on VDD verification procedures.

- PIC16C5X Programming Specifications - DS30190
- PIC16C55X Programming Specifications - DS30261
- PIC16C6X/7X/9XX Programming Specifications - DS30228
- PIC16C84 Programming Specifications - DS30189

Note: The designer must consider environmental conditions, voltage ranges, and aging issues when determining VDD min/max verification levels. Please refer to the programming specification for the application device.

The calibration programming and initial verification MUST occur at +5V. If the application is intended to run at lower (or higher voltages), a second verification pass must be added where those voltages are applied to VDD and the device is verified.
Communication Format (Application Microcontroller to Programming Jig)

Unused program memory, in the application PIC16CXXX, is left unprogrammed as all 1s; therefore the unprogrammed program memory for the calibration look-up table would contain 3FFF (hex). This is interpreted as an "ADDLW FF". The application microcontroller simply needs one "RETLW FF" instruction at the end of the space allocated in program memory for the calibration parameter look-up table. When the application microcontroller is powered up, it will receive a "FFh" for each calibration parameter that is looked up; therefore, it can detect that it is uncalibrated and jump to the calibration code.

Once the calibration constants are calculated by the application PICmicro MCU, they need to be communicated to the (PIC16C58A based) programming jig. This is accomplished through the RB6 and RB7 lines. The format is a simple synchronous clock and data format as shown in Figure 2.

A pull-down on the clock line is used to hold it low. The application microcontroller needs to send the high and low bytes of the target start address of the calibration constants to the calibration jig. Next, the data bytes are sent followed by a checksum of the entire data transfer as shown in Figure 1.

Once the data transfer is complete, the checksum is verified by the programming jig and the data printed at 9600 baud, 8-bits, no parity, 1 stop bit through RB3. A connection to this pin is optional. Next the programming jig applies +13V, programs and verifies the application PIC16CXXX calibration parameters.

FIGURE 2:

<table>
<thead>
<tr>
<th>RB6</th>
<th>RB7</th>
<th>CALbit7</th>
<th>CALbit6</th>
<th>CALbit5</th>
<th>CALbit4</th>
<th>CALbit3</th>
<th>CALbit2</th>
<th>CALbit1</th>
<th>CALbit0</th>
</tr>
</thead>
</table>

FIGURE 1:

| AddrH | AddrL | Data 0 | Data 1 | ... | Data N | CKSUM |
LED Operation

When the programming jig is waiting for communication from the application PICmicro MCU, both LEDs are OFF. Once a valid data stream is received (with at least one calibration byte and a correct checksum) the WORK LED is lit while the calibration parameters are printed through the optional RB3 port. Next, the DONE LED is lit to indicate that these parameters are being programmed and verified by the programming jig. Once the programming is finished, the WORK LED is extinguished and the DONE LED remains lit. If any parameters fail programming, the DONE LED is extinguished; therefore both LEDs would remain off.

FIGURE 3: ISP CALIBRATION JIG PROGRAMMER SCHEMATIC
Code Protection

Selection of the code protection configuration bits on PIC16CXXX microcontrollers prevents further programming of the program memory array. This would prevent writing self calibration parameters if the device is code protected prior to calibration. There are two ways to address this issue:

1. Do not code protect the device when programming it with the programmer. Add additional code (See the PIC16C6X/7X programming Spec) to the ISPPRGM.ASM to program the code protection bit after complete verification of the calibration parameters.

2. Only code protect 1/2 or 3/4 of the program memory with the programmer. Place the calibration constants into the unprotected part of program memory.

Software Routines

There are two source code files needed for this application note:

1. ISPTEST.ASM (Appendix A) Contains the source code for the application PIC16CXXX, sets up the calibration look-up table and implements the communication protocol to the programming jig.

2. ISPPRGM.ASM (Appendix B) Source code for a PIC16C58A to implement the programming jig. This waits for and receives the calibration parameters from the application PIC16CXXX, places it into programming mode and programs/verifies each calibration word.

CONCLUSION

Typically, calibration information about a system is stored in EEPROM. For calibration data that does not change over time, the In-circuit Serial Programming capability of the PIC16CXXX devices provide a simple, cost effective solution to an external EEPROM. This method not only decreases the cost of a design, but also reduces the complexity and possible failure points of the application.

TABLE 1: PARTS LIST FOR PIC16CXXX ISP CALIBRATION JIG

<table>
<thead>
<tr>
<th>Bill of Material</th>
<th>Quantity</th>
<th>Reference</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>C1,C2</td>
<td>15 pF</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>C3</td>
<td>620 pF</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>C4</td>
<td>0.1 mF</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>C5,C6</td>
<td>220 mF</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>D1,D2</td>
<td>LED</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>E1</td>
<td>PIC16C58</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>E2</td>
<td>LM78S40</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>J1</td>
<td>CON5</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>L1</td>
<td>270 mH</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>Q1,Q2</td>
<td>2N2222</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>Q3,Q4</td>
<td>2N2907</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>R1,R2,R3,R4,R15</td>
<td>1k</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>R5,R6,R12,R14</td>
<td>10k</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>R7,R8</td>
<td>270</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>R9</td>
<td>180</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>R10</td>
<td>23.7k</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>R11</td>
<td>2.49k</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>R13</td>
<td>2.2k</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>Y1</td>
<td>4.0 MHz</td>
</tr>
</tbody>
</table>
APPENDIX A:
MPASM 01.40.01 Intermediate  ISPPRGM.ASM  3-31-1997  10:57:03  PAGE 1

LOC  OBJECT CODE   LINE SOURCE TEXT
VALUE

00001 ; Filename: ISPPRGM.ASM
00002 ; *******************************************************
00003 ; * Author:   John Day                               *
00004 ; *           Sr. Field Applications Engineer       *
00005 ; * Microchip Technology                           *
00006 ; * Revision: 1.0                                  *
00007 ; * Date      August 25, 1995                      *
00008 ; * Part:     PIC16C58                              *
00009 ; * Compiled using MPASM V1.40                     *
0010 ; *******************************************************
0011 ; * Include files:                                  *
0012 ; * P16CSX.ASM                                       *
0013 ; *******************************************************
0014 ; * Fuses:    OSC:  XT (4.0 Mhz xtal)                *
0015 ; *           WDT:  OFF                               *
0016 ; *            CP:  OFF                               *
0017 ;*******************************************************
0018 ; This program is intended to be used as a self programmer
0019 ; to store calibration constants into a lookup table
0020 ; within the main system processor.  A 4 Mhz crystal
0021 ; is needed and an optional 9600 baud serial port will
0022 ; display the parameters to be programmed.
0023 ;*******************************************************
0024 ; *******************************************************
0025 ; * Program Memory:                                   *
0026 ; * Words - communication with test jig              *
0027 ; * 17 Words - calibration look-up table (16 bytes of data)*
0028 ; * 13 Words - Test Code to generate Calibration Constants *
0029 ; * RAM memory:                                       *
0030 ; * 64 Bytes - Store up to 64 bytes of calibration constant *
0031 ; * 9 Bytes - Store 9 bytes of temp variables (reused) *
0032 ;*******************************************************
0033  list p=16C58A
0034  include <p16C5x.inc>
0001 LIST
0024 LIST
0OFF 0FF9
0035  __CONFIG _CP_OFF&_WDT_OFF&_XT_OSC
0036
0037 ;*******************************************************
0038 ; * Port A (RA0-RA4) bit definitions                  *
0039 ;*******************************************************
0040 ; No PORT A pins are used in this design              *
0041
0042 ;*******************************************************
0043 ; * Port B (RB0-RB7) bit definitions                  *
0044 ;*******************************************************
0045 ISPCLOCK       EQU 6   ; Clock line for ISP and parameter comm
0046 ISPDATA       EQU 7   ; Data line for ISP and parameter comm
0047 VPPFON       EQU 5   ; Apply +13V VPP voltage to MCLR (test mode)
0044 GNDON       EQU 4   ; Apply +0V (gnd) voltage to MCLR (reset)
0049 SEROUT       EQU 3   ; Optional RS-232 TX output (needs 12V driver)
0050 DONELED      EQU 2   ; Turns on LED when done successfullly program
0051 WORKLED      EQU 1   ; On during programming, off when done
0052 ; RB0 is not used in this design
00054 ; **************************************************
00055 ; * RAM register definition:                      *
00056 ; * 07h - 0Fh - used for internal counters, vars  *
00057 ; * 10h - 7Fh - 64 bytes for cal param storage   *
00058 ; **************************************************
00059 ; ***
00060 ; *** The following VARS are used during ISP programming:
00061 ; ***
00062     00000007 HIADDR     EQU 07h ; High address of CAL params to be stored
00063     00000008 LOADDR     EQU 08h ; Low address of CAL params to be stored
00064     00000007 HIDATA     EQU 07h ; High byte of data to be sent via ISP
00066     00000008 LODATA     EQU 08h ; Low byte of data to be sent via ISP
00067     00000009 HIBYTE     EQU 09h ; High byte of data received via ISP
00068     0000000A LOBYTE     EQU 0Ah ; Low byte of data received via ISP
00069     0000000B PULSECNT   EQU 0Bh ; Number of times PIC has been pulse programmed
00070     0000000C TEMPCODE    EQU 0Ch ; TEMP var used in counters
00071     0000000D TEMP       EQU 0Dh ; TEMP var used throughout program
00072 ; ***
00073 ; *** The following VARS are used to receive and store CAL params:
00074     00000007 COUNT      EQU 07h ; Counter var used to receive cal params
00075     00000008 TEMPI      EQU 08h ; TEMP var used for RS-232 comm
00076     00000007 DATAREG    EQU 09h ; Data register used for RS-232 comm
00077     0000000A CSUMTOTAL   EQU 0Ah ; Running total of checksum (addr + data)
00078     0000000B TIMEHIGH   EQU 0Bh ; Count how long CLOCK line is high
00079     0000000C TIMELOW    EQU 0Ch ; Count how long CLOCK line is low
00080     0000000E ADDRPTR    EQU 0Dh ; Pointer to next byte of CAL storage
00081     0000000F BYTECOUNT  EQU 0Eh ; Number of CAL bytes received
00082
00083 ; **************************************************
00084 ; * Various constants used in program *
00085 ; **************************************************
00086     00000001 DATISPOUT  EQU b'00000001' ; tris settings for ISP data out
00087     00000001 DATISPIN   EQU b'10000001' ; tris settings for ISP data in
00088     00000006 CMDISPNCNT  EQU 6 ; Number of bits for ISP command
00089     00000007 STARTCALBYTE EQU 10h ; Address in RAM where CAL byte data stored
00090     00000007 VFYYES     EQU PA2 ; Flag bit enables verification (STATUS)
00091     00000006 CMDISPINCADDR EQU b'00000010' ; ISP Pattern to increment address
00092     00000006 CMDISPSTART  EQU b'00000100' ; ISP Pattern to start programming
00093     00000006 CMDISPEND   EQU b'00001110' ; ISP Pattern to end programming
00094     00000006 CMDISPSLOAD EQU b'00000010' ; ISP Pattern to load data for program
00095     00000004 CMDISPSREAD EQU b'00000010' ; ISP Pattern to read data for verify
00096     00000034 UPPER6BITS  EQU 034h ; Upper 6 bits for retlw instruction
00097
00098 ; **************************************************
00099 ; * delaybit macro *
01000 ; * Delays for 104 uS (at 4 Mhz clock)*
01001 ; * for 9600 baud communications *
01002 ; * RAM used: COUNT *
01003 ; **************************************************
01004 delaybit macro
01005     local dlylabels
01006     9600 baud, 8 bit, no parity, 104 us per bit, 52 us per half bit
01007     (8) shift/usage + (2) setup + (1) nop + (3 * 31) literal = (104) 4Mhz
01008     movlw .31 ; place 31 decimal literal into count
01009     movwf COUNT ; Initialize COUNT with loop count
01010     nop ; Add one cycle delay
01011     dlylabels
01012     decfsz COUNT,F ; Decrement count until done
01013     goto dlylabels ; Not done delaying - go back!
01014     ENDM ; Done with Macro
01015 ; **************************************************
01016 ; **************************************************
01017 ; * addrtofsr macro *
01018 ; * Converts logical, continuous address 10h-4Fh *
01019 ; * to FSR address as follows for access to (4) *
00120 ; * banks of file registers in PIC16C58: *
00121 ; * Logical Address FSR Value *
00122 ; * 10h-1Fh 10h-1Fh *
00123 ; * 20h-2Fh 30h-3Fh *
00124 ; * 30h-3Fh 50h-5Fh *
00125 ; * 40h-4Fh 70h-7Fh *
00126 ; * Variable Passed: Logical Address *
00127 ; * RAM used: FSR *
00128 ; * W *
00129 ; ************************************************

00130 addrtofsr macro TESTADDR
00131     movlw   STARTCALBYTE        ; Place base address into W
00132     subwf   TESTADDR,w          ; Offset by STARTCALBYTE
00133     movwf   FSR                 ; Place into FSR
00134     btfsc   FSR,5               ; Shift bits 4,5 to 5,6
00135     bsf     FSR,6
00136     bcf     FSR,5
00137     btfsc   FSR,4
00138     bsf     FSR,5
00139     bsf     FSR,4
00140     endm
00141
00142
00143 ; ************************************************
00144 ; * The PC starts at the END of memory *
00145 ; ************************************************

07FF 00146     ORG 7FFh
Message[306]: Crossing page boundary -- ensure page bits are set.
07FF 0A00     goto    start

00147
00148
00149 ; ************************************************
00150 ; * Start of CAL param read routine *
00151 ; ************************************************

0000 00152     ORG 0h
0000 00153     start

0000 0C0A 00154     movlw b’00001010’ ; Serial OFF, LEDs OFF, VPP OFF
0001 0026 00155     movwf PORTB ; Place “0” into port b latch register
0002 0CC1 00156     movlw b’11000001’ ; RB7;:RB6, RB0 set to inputs
0003 0006 00157     tris PORTB ; Move to tris registers
0004 0040 00158     clrw ; Place 0 into W
0005 0065 00159     clrf PORTA ; Place all ZERO into latch
0006 0005 00160     tris PORTA ; Make all pins outputs to be safe..
0007 0586 00161     bsf PORTB,GNDON ; TEST ONLY-RESET PIC-NOT NEEDED IN REAL DESIGN!
0008 00162     clearram
0008 0C10 00163     movlw 010h ; Place start of buffer into W
0009 0027 00164     movwf COUNT ; Use count for RAM pointer
000A 00165     loopclrram
00166     addrtofsr COUNT ; Set up FSR
000A 0C10 00167     movlw STARTCALBYTE ; Place base address into W
000B 0087 00168     subwf COUNT,w ; Offset by STARTCALBYTE
000C 0024 00169     movwf FSR ; Place into FSR
000D 06A4 00170     bsf FSR,5 ; Shift bits 4,5 to 5,6
000E 05C4 00171     bsf FSR,6
000F 04A4 00172     bsf FSR,5
0010 0684 00173     bsf FSR,4
0011 05A4 00174     bsf FSR,5
0012 0584 00175     bsf FSR,4
0013 0060 00176     clrf INDF ; Clear buffer value
0014 02A7 00177     incf COUNT,F ; Move to next reg
0015 0C50 00178     movlw 050h ; Move end of buffer addr to W
0016 0087 00179     subwf COUNT,W ; Check if at last MEM
0017 0743 00180     btfsc STATUS,2 ; Skip when at end of counter
0018 0A0A 00181     goto loopclrram ; go back to next location
0019 0486 00182     bsf PORTB,GNDON ; TEST ONLY-LET IT GO-NOT NEEDED IN REAL DESIGN!
001A 00183     calget
001A 006A 00184     clrf CSUMTOTAL ; Clear checksum total byte
001B 0069    clrf    DATAREG             ; Clear out data receive register
001C 0077    movlw   STARTCALBYTE        ; Place RAM start address of first cal byte
001D 0078    movwf   ADDRPTR             ; Place this into ADDRPTR
001E 07C6    btfss   PORTB,ISPCLOCK      ; Wait for CLOCK high pulse - skip when high
001F 0A1E    goto    waitclockpulse    ; CLOCK is low - go back and wait!
0020
0020 001B    clrf    DATAREG             ; Clear out data receive register
0021 0C10    movlw   STARTCALBYTE        ; Place RAM start address of first cal byte
0022 0017    movwf   ADDRPTR             ; Place this into ADDRPTR
0023 0017    movwf   COUNT               ; set up counter register to count bits
0024 006B    clrf    TIMEHIGH            ; Clear timeout counter for high pulse
0025 006C    clrf    TIMELOW             ; Clear timeout counter for low pulse
0026 06C6    btfsc   PORTB,ISPCLOCK      ; Wait for CLOCK high - skip if it is low
0027 0A29    goto    waitclockpulse    ; CLOCK is low - go back and wait!
0028
0028 006B    clrf    TIMEHIGH            ; Clear timeout counter for high pulse
0029 006C    clrf    TIMELOW             ; Clear timeout counter for low pulse
002A 06C6    btfsc   PORTB,ISPCLOCK      ; Wait for CLOCK high - skip if it is low
002B 001B    clrf    DATAREG             ; Clear out data receive register
002C 006B    clrf    TIMEHIGH            ; Clear timeout counter for high pulse
002D 006C    clrf    TIMELOW             ; Clear timeout counter for low pulse
002E 06C6    btfsc   PORTB,ISPCLOCK      ; Wait for CLOCK high - skip if it is low
002F 001B    clrf    DATAREG             ; Clear out data receive register
0030 0743    btfss   PORTB,ISPDATA       ; Skip if the data bit was high
0031 0403    bcf     STATUS,C            ; Set data bit to low
0032 0369    rlf     DATAREG,F           ; Rotate next bit into DATAREG
0033 02E7    decfsz  COUNT,F             ; Skip after 8 bits
0034 0A22    goto    loopsendcal         ; Jump back and send next bit
0035 0209    movf    DATAREG,W           ; Place received byte into W
0036 0020    movwf   ADDRFTR,W           ; Move current address pointer to W
0037 06A4    movf    FSR,5               ; Shift bits 4,5 to 5,6
0038 04A4    movf    FSR,4               ; Shift bits 4,5 to 5,6
0039 05C4    movf    FSR,6               ; Shift bits 4,5 to 5,6
003A 0503    bsf     STATUS,C            ; Assume data bit is high
003B 07E6    btfss   STATUS,Z            ; Skip if we are at count 8 (first value)
003C 0403    bcf     STATUS,C            ; Set data bit to low
003D 0369    rlf     DATAREG,F           ; Rotate next bit into DATAREG
003E 02E7    decfsz  COUNT,F             ; Skip after 8 bits
003F 0A22    goto    loopsendcal         ; Jump back and send next bit
0040 0209    movf    DATAREG,W           ; Place received byte into W
0041 0020    movwf   ADDRFTR,W           ; Move current address pointer to W
0042 001B    clrf    DATAREG             ; Clear out data receive register
0043 0077    movlw   STARTCALBYTE        ; Place RAM start address of first cal byte
0044 0078    movwf   ADDRPTR             ; Place this into ADDRPTR
0045 018F    btfss   PORTB,ISPCLOCK      ; Wait for CLOCK high pulse - skip when high
0046 0A1E    goto    waitclockpulse    ; CLOCK is low - go back and wait!
0047
0047 0C14    movlw   STARTCALBYTE+4      ; check if we received (4) params
0048 008E    subwf   ADDRFTR,W           ; Move current address pointer to W
0049 0703    btfss   STATUS,C            ; Skip if we have at least (4)
004A 0A93    goto    sendnoise           ; not enough params - print and RESET!
004B 0200    movf    ADDRFTR,W           ; Move received checksum into W
004C 00AA    subwf   CSUMTOTAL,F         ; Subtract received Checksum from calc’d checksum
004D 0743    btfss   STATUS,Z            ; Skip if CSUM OK
004E 0A9F    goto    sendcsumbad         ; Checksum bad - print and RESET!
004F 0369    rlf     DATAREG,F           ; Rotate next bit into DATAREG
0050 0426    bcf     PORTB,WORKLED       ; Turn on WORK LED
0051 008E    subwf   ADDRFTR,W           ; Move current address pointer to W
0052 02F2    movwf   BYTECOUNT           ; Place into number of bytes into BYTECOUNT

© 2000 Microchip Technology Inc.
0053 002B 00233 movwf TIMEHIGH ; TEMP store into timehigh reg
0054 0C10 00234 movlw STARTCALBYTE ; Place start address into W
0055 002E 00235 movwf ADDRPR ; Set up address pointer
0056 0026 00236 loopprintnums ; Set up FSR
0057 0027 00237 addrsfor ADDRPR ; Set up FSR
0058 0C10 00238 movlw STARTCALBYTE ; Place start address into W
0059 002E 00239 movwf ADDRPTR ; Set up address pointer
005A 0024 00240 loopprintnums ; Set up FSR
005B 0C10 00241 movlw STARTCALBYTE ; Place base address into W
005C 008E 00242 subwf ADDRPTR,w ; Offset by STARTCALBYTE
005D 0024 00243 btsc STATUS,C ; Skip if TEMP is less than 9
005E 0380 00244 swapf INDF,W ; Place received char into W
005F 0E0F 00245 andlw 0Fh ; Strip off upper digits
0060 002D 00246 movwf TEMP ; Place into TEMP
0061 0020 00247 andlw 0Ph ; Strip off upper digits
0062 002D 00248 movwf TEMP ; Place into TEMP
0063 0C90 00249 movlw '0' ; Place ASCII '0' into W
0064 01CD 00250 addwf TEMP,w ; Add to TEMP, place into W
0065 09AE 00251 call putchar ; Send out char
0066 0A81 00252 goto printlo ; Jump to print next char
0067 0C57 00253 printloletter ; Greater than 9 - print letter instead
0068 0C57 00254 printlo ; Jump to print next char
0069 0C57 00255 printloletter ; Greater than 9 - print letter instead
006A 0E0F 00256 andlw 0Fh ; Strip off upper digits
006B 002D 00257 movwf TEMP ; Place into TEMP
006C 0C90 00258 movlw '0' ; Place ASCII '0' into W
006D 01CD 00259 addwf TEMP,w ; Add to TEMP, place into W
006E 09AE 00260 call putchar ; Send out char
006F 028E 00261 movf INDF,W ; Place received char into W
0070 0C7C 00262 andlw 0Ph ; Strip off upper digits
0071 01CD 00263 addwf TEMP,w ; Add to TEMP, place into W
0072 09AE 00264 movf INDF,W ; Place received char into W
0073 0E0F 00265 andlw 0Ph ; Strip off upper digits
0074 028E 00266 andlw 0Ph ; Strip off upper digits
0075 002D 00267 movf INDF,W ; Place received char into W
0076 0C90 00268 movlw '0' ; Place ASCII '0' into W
0077 01CD 00269 addwf TEMP,w ; Add to TEMP, place into W
0078 09AE 00270 call putchar ; Send out char
0079 028E 00271 movf INDF,W ; Place received char into W
007A 0C7C 00272 andlw 0Ph ; Strip off upper digits
007B 01CD 00273 addwf TEMP,w ; Add to TEMP, place into W
007C 09AE 00274 call putchar ; Send out char
007D 0C7C 00275 movf INDF,W ; Place received char into W
007E 0C7C 00276 andlw 0Ph ; Strip off upper digits
007F 01CD 00277 addwf TEMP,w ; Add to TEMP, place into W
0080 09AE 00278 call putchar ; Send out char
0081 028E 00279 movf INDF,W ; Place received char into W
0082 0C7C 00280 andlw 0Ph ; Strip off upper digits
0083 01CD 00281 addwf TEMP,w ; Add to TEMP, place into W
0084 09AE 00282 call putchar ; Send out char
0085 028E 00283 movf INDF,W ; Place received char into W
0086 0C7C 00284 andlw 0Ph ; Strip off upper digits
0087 01CD 00285 addwf TEMP,w ; Add to TEMP, place into W
0088 09AE 00286 call putchar ; Send out char
008C 09A9  00289  call  printcrlf  ; Print CR and LF every 16 chars
008D 02AE  00290  incf  ADDRPTR,F  ; go to next address
008E 02EF  00291  decfsz  BYTECOUNT,F  ; Skip after last byte
008F 0A56  00292  goto  loopprintnums  ; Go back and print next char
0090 09A9  00293  call  printcrlf  ; Print CR and LF
0091 05A3  00294  bsf  STATUS,PA0  ; Set page bit to page 1
0092 0A6B  00295  goto  programpartisp  ; Go to program part through ISP
0093 0C4E  00296  movlw  'N'  ; Place 'N' into W
0094 09AE  00297  call  putchar  ; Send char in W to terminal
0095 0C4F  00298  movlw  'O'  ; Place 'O' into W
0096 09AE  00299  call  putchar  ; Send char in W to terminal
0097 0C49  00300  movlw  'I'  ; Place 'I' into W
0098 09AE  00301  call  putchar  ; Send char in W to terminal
0099 0C53  00302  movlw  'S'  ; Place 'S' into W
009A 09AE  00303  call  putchar  ; Send char in W to terminal
009B 0C45  00304  movlw  'E'  ; Place 'E' into W
009C 09AE  00305  call  putchar  ; Send char in W to terminal
009D 0C0D  00306  movlw  .13   ; Value for CR placed into W
009E 09AE  00307  call  printcrlf  ; Print CR and LF
009F 0A1A  00308  goto  calget  ; RESET!
00A0 0C43  00309  movlw  'C'  ; Place 'C' into W
00A1 09AE  00310  call  putchar  ; Send char in W to terminal
00A2 0C53  00311  movlw  'S'  ; Place 'S' into W
00A3 09AE  00312  call  putchar  ; Send char in W to terminal
00A4 0C55  00313  movlw  'U'  ; Place 'U' into W
00A5 09AE  00314  call  putchar  ; Send char in W to terminal
00A6 0C4D  00315  movlw  'M'  ; Place 'M' into W
00A7 09A9  00316  call  printcrlf  ; Print CR and LF
00A8 0A1A  00317  goto  calget  ; RESET!
00A9 0328  00318  printcrlf
00A9 0C0D  00319  movlw  .13   ; Value for CR placed into W
00AA 09AE  00320  call  putchar  ; Send char in W to terminal
00AB 0C0A  00321  movlw  .10   ; Value for LF placed into W
00AC 09AE  00322  call  putchar  ; Send char in W to terminal
00AD 0800  00323  retlw  0     ; Done - return!
00324
00325 ; **********************************************************************************
00326 ;  * printcrlf  *
00327 ;  * Sends char .13 (Carrage Return) and *
00328 ;  * char .10 (Line Feed) to RS-232 port *
00329 ;  * by calling putchar.  *
00330 ;  * RAM used:  W  *
00331 ; **********************************************************************************
00332
00333 ; **********************************************************************************
00334 ;  * putchar  *
00335 ;  * Print out the character stored in W  *
00336 ;  * by toggling the data to the RS-232  *
00337 ;  * output pin in software.  *
00338 ;  * RAM used:  W,DATAREG,TEMP1  *
00339 ; **********************************************************************************
00340
00341 ; **********************************************************************************
00342
00343 ; **********************************************************************************
00344 ;  * putloop1  *
00345 ;  * Send next bit in carry  *
00346 ; **********************************************************************************
00347
00348 ; **********************************************************************************
00349 ;  * rrf  DATAREG,F  *
00350 ;  * Skip if carry was set  *
00351 ; **********************************************************************************
00352
00353 ; **********************************************************************************
00354 ;  * bcf  PORTB,SEROUT  *
00355 ;  * Clear RS-232 serial output bit  *
00356 ;  * Skip if carry was clear  *
00357 ; **********************************************************************************
00B7 0566 00354  bsf PORTB, SEROUT ; Set RS-232 serial output bit
00355  delaybit ; Delay for one bit time
0000  M local dlylabels
M ; 9600 baud, 8 bit, no parity, 104 us per bit, 52 uS per half bit
M ; (8) shift/usage + (2) setup + (1) nop + (3 * 31) literal = (104) 4Mhz
00B8 0C1F  M movlw .31 ; place 31 decimal literal into count
00B9 0027  M movwf COUNT ; Initialize COUNT with loop count
00BA 0000  M nop ; Add one cycle delay
00BB  M dlylabels
00BB 02E7  M decfsz COUNT,F ; Decrement count until done
00BC 0ABB  M goto dlylabels ; Not done delaying - go back!
00BD 02E8 00356  decfsz TEMP1,F ; Decrement bit counter, skip when done!
00BE 0AB3 00357  M goto putloop ; Jump back and send next bit
00BF 0566 00358  bsf PORTB, SEROUT ; Send out stop bit
00359  delaybit ; delay for stop bit
0000  M local dlylabels
M ; 9600 baud, 8 bit, no parity, 104 us per bit, 52 uS per half bit
M ; (8) shift/usage + (2) setup + (1) nop + (3 * 31) literal = (104) 4Mhz
00C0 0C1F  M movlw .31 ; place 31 decimal literal into count
00C1 0027  M movwf COUNT ; Initialize COUNT with loop count
00C2 0000  M nop ; Add one cycle delay
00C3  M dlylabels
00C3 02E7  M decfsz COUNT,F ; Decrement count until done
00C4 0AC3  M goto dlylabels ; Not done delaying - go back!
00C5 0800 00360  retlw 0 ; Done - RETURN
00361
00362 ; ************************************************************
00363 ; * ISP routines from PICSTART-16C *
00364 ; * Converted from PIC17C42 to PIC16C5X code by John Day *
00365 ; * Originally written by Jim Pepping *
00366 ; ************************************************************
0200 00367  ORG 200 ; ISP routines stored on page 1
00368
00369 ; ************************************************************
00370 ; * poweroffisp *
00371 ; * Power off application PIC - turn off VPP and reset device after *
00372 ; * programming pass is complete *
00373 ; ************************************************************
0200 00374  poweroffisp
0200 04A6 00375  bcf PORTB, VPPON ; Turn off VPP 13 volts
0201 0586 00376  bsf PORTB, GNDON ; Apply 0 V to MCLR to reset PIC
0202 0C01 00377  movlw b'11000001' ; RB6,7 set to inputs
0203 0006 00378  tris PORTB ; Move to tris registers
0204 0486 00379  bsf PORTB, GNDON ; Allow MCLR to go back to 5 volts, deassert reset
0205 0526 00380  bcf PORTB, WORKLED ; Turn off WORK LED
0206 0800 00381  retlw 0 ; Done so return!
00382
00383 ; ************************************************************
00384 ; * testmodeisp *
00385 ; * Apply VPP voltage to place application PIC into test mode. *
00386 ; * this enables ISP programming to proceed *
00387 ; * RAM used: TEMP *
00388 ; ************************************************************
0207 00389  testmodeisp
0207 0C08 00390  movlw b'00001000' ; Serial OFF, LEDS OFF, VPP OFF
0208 0026 00391  movwf PORTB ; Place "0" into port b latch register
0209 04A6 00392  bcf PORTB, VPPON ; Turn off VPP just in case!
020A 0586 00393  bsf PORTB, GNDON ; Apply 0 volts to MCLR
020B 0C01 00394  movlw b'00000001' ; RB6,7 set to outputs
020C 0006 00395  tris PORTB ; Move to tris registers
020D 0206 00396  movf PORTB, W ; Place PORT B state into W
020E 002D 00397  movwf TEMP ; Move state to TEMP
020F 048D 00398  bcf TEMP, 4 ; Turn off MCLR GND
0210 05AD 00399  bsf TEMP, 5 ; Turn on VPP voltage
0211 020D 00400  movf TEMP, W ; Place TEMP into W
0212 0026 00401  movwf PORTB ; Turn OFF GND and ON VPP
0213 0546 00402 bsf PORTB,DONELED ; Turn ON GREEN LED
0214 0800 00403 retlw 0 ; Done so return!

00404

00405 ; ********************************************************************************
00406 ; * p16cispout                                                      *
00407 ; * Send 14-bit data word to application PIC for writing this data *
00408 ; * to it’s program memory. The data to be sent is stored in both *
00409 ; * HIBYTE (6 MSBs only) and LOBYTE.                             *
00410 ; * RAM used: TEMP, W, HIBYTE (inputs), LOBYTE (inputs)   *

00411 ; ********************************************************************************

0215 00412 P16cispout

0215 0C0E 00413 movlw .14             ; Place 14 into W for bit counter
0216 002D 00414 movwf TEMP            ; Use TEMP as bit counter
0217 04C6 00415 bcf PORTB,ISPCLOCK    ; Clear CLOCK line
0218 04E6 00416 bcf PORTB,ISPDATA      ; Clear DATA line
0219 0C01 00417 movlw DATISPOUT        ; Place tris value for data output
021A 0006 00418 tris PORTB            ; Set tris latch as data output
021B 04E6 00419 bcf PORTB,ISPDATA      ; Send a start bit (0)
021C 05C6 00420 bcf PORTB,ISPCLOCK    ; Set CLOCK output
021D 04C6 00421 bcf PORTB,ISPCLOCK    ; Clear CLOCK output (clock start bit)

021E 00422 P16cispoutloop

021E 0403 00423 bcf STATUS,C           ; Clear carry bit to start clean
021F 04C6 00424 bcf PORTB,ISPCLOCK    ; Clear CLOCK bit
0220 04E6 00425 rrf HIBYTE,F           ; Rotate HIBYTE output
0221 032A 00426 rrf LOBYTE,F           ; Rotate LOBYTE output
0222 0603 00427 btfsc STATUS,C         ; Skip if data bit is zero
0223 05C6 00428 bcf PORTB,ISPCLOCK    ; Set CLOCK output
0224 04C6 00429 bcf PORTB,ISPCLOCK    ; Clear CLOCK output (clock stop bit)
0225 04E6 00430 bcf PORTB,ISPDATA      ; Clear DATA bit to start (assume 0)
0226 0006 00431 decfsz TEMP,F          ; Decrement bit counter, skip when done
0227 0A1E 00432 goto P16cispoutloop     ; Jump back and send next bit
0228 04E6 00433 bcf PORTB,ISPDATA      ; Send a stop bit (0)
0229 05C6 00434 bcf PORTB,ISPCLOCK    ; Set CLOCK output
022A 04C6 00435 bcf PORTB,ISPCLOCK    ; Clear CLOCK output (clock stop bit)
022B 0800 00436 retlw 0               ; Done so return!

00437

00438 ; ********************************************************************************

00439 ; * p16cispin                                                      *
00440 ; * Receive 14-bit data word from application PIC for reading this *
00441 ; * data from it’s program memory. The data received is stored in *
00442 ; * both HIBYTE (6 MSBs only) and LOBYTE.                          *
00443 ; * RAM used: TEMP, W, HIBYTE (output), LOBYTE (output)   *

00444 ; ********************************************************************************

022C 00445 P16cispin

022C 0C0E 00446 movlw .14             ; Place 14 data bit count value into W
022D 002D 00447 movwf TEMP            ; Init TEMP and use for bit counter
022E 0069 00448 clrwf HIBYTE           ; Clear recieved HIBYTE register
022F 006A 00449 clrwf LOBYTE           ; Clear recieved LOBYTE register
0230 0403 00450 bcf STATUS,C          ; Clear carry bit to start clean
0231 04C6 00451 bcf PORTB,ISPCLOCK    ; Clear CLOCK output
0232 04E6 00452 bcf PORTB,ISPDATA      ; Clear DATA output
0233 0C01 00453 movlw DATISPIN         ; Place tris value for data input into W
0234 0006 00454 tris PORTB            ; Set up tris latch for data input
0235 05C6 00455 bcf PORTB,ISPCLK      ; Send a single clock to start things going
0236 04C6 00456 bcf PORTB,ISPCLOCK    ; Clear CLOCK to start receive

0237 00457 P16cispinloop

0237 05C6 00458 bcf PORTB,ISPCLOCK    ; Set CLOCK bit
0238 0000 00459 nop                   ; Wait one cycle
0239 0403 00460 bcf STATUS,C          ; Clear carry bit, assume 0 read
023A 06E6 00461 btfsc PORTB,ISPDATA    ; Check the data, skip if it was zero
023B 0503 00462 bcf STATUS,C          ; Set carry bit if data was one
023C 0329 00463 rrf HIBYTE,F           ; Move received bit into HIBYTE
023D 032A 00464 movf LOBYTE,F          ; Update LOBYTE
023E 04C6 00465 bcf PORTB,ISPCLK       ; Clear CLOCK line
023F 0000 00466 nop                   ; Wait one cycle
0240 0000 00467 nop                   ; Wait one cycle
0241 02ED 00468     decfsz TEMP,F     ; Decrement bit counter, skip when zero
0242 0A37 00469     goto P16cispinloop ; Jump back and receive next bit
0243 05C6 00470     bsf PORTB,ISPCLOCK ; Clock a stop bit (0)
0244 0000 00471     nop                       ; Wait one cycle
0245 04C6 00472     bcf PORTB,ISPCLOCK     ; Clear CLOCK to send bit
0246 0000 00473     nop                       ; Wait one cycle
0247 0403 00474     bcf STATUS,C           ; Clear carry bit
0248 0329 00475     rrf HIBYTE,F           ; Update HIBYTE with the data
0249 032A 00476     rrf LOBYTE,F           ; Update LOBYTE
024A 0403 00477     bcf STATUS,C           ; Clear carry bit
024B 032A 00478     rrf LOBYTE,F           ; Update LOBYTE with the data
024C 0329 00479     rrf HIBYTE,F           ; Update HIBYTE with the data
024D 04C6 00480     bcf PORTB,ISPCLOCK     ; Clear CLOCK line
024E 04E6 00481     bcf PORTB,ISPDATA       ; Clear DATA line
024F 0C01 00482     movlw DATISPOUT        ; Place tris value for data output into W
0250 0006 00483     tris PORTB            ; Set tris to data output
0251 0800 00484     retlw 0               ; Done so RETURN!

; *******************************************************************
; * commandisp                                                      *
; * Send 6-bit ISP command to application PIC. The command is sent *
; * in the W register and later stored in LOBYTE for shifting.     *
; *     RAM used: LOBYTE, W, TEMP                                   *
; *******************************************************************

0252 0002A 00493     movwf LOBYTE           ; Place command into LOBYTE
0253 0C06 00494     movlw CMDISPCNT         ; Place number of command bits into W
0254 002D 00495     movwf TEMP             ; Use TEMP as command bit counter
0255 04E6 00496     bcf PORTB,ISPDATA       ; Clear DATA line
0256 04C6 00497     bcf PORTB,ISPCLOCK      ; Clear CLOCK line
0257 0C01 00498     movlw DATISPOUT        ; Place tris value for data output into W
0258 0006 00499     tris PORTB            ; Set tris to data output
0259 00500 00500 P16cispccmdoutloop

; ********************************************************************
; * programpartisp                                                   *
; * Main ISP programming loop. Reads data starting at STARTCALBYTE   *
; * and calls programming subroutines to program and verify this     *
; * data into the application PIC.                                   *
; *     RAM used: LOADDR, HIADDR, LODATA, HIDATA, FSR, LOBYTE, HIBYTE*
; ********************************************************************

026B 0027 00501 00507 programpartisp
026B 0907 00507 00511 call testmodelisp    ; Place PIC into test/program mode
026C 0064 00512 00517 clrf FSR             ; Point to bank 0
026D 0210 00518 00519 movf STARTCALBYTE,W ; Upper order address of data to be stored into W
026E 0027 00519 00520 movf HIADDR          ; place into counter
026F 0211 00520 00521 movf STARTCALBYTE+1,W ; Lower order address byte of data to be stored
0270 0028 00521 00522 movf LOADDR          ; place into counter
0271 00E8 00534 decf LOADDR,F ; Subtract one from loop constant
0272 02A7 00535 incf HIADDR,F ; Add one for loop constant
0273 0C06 00536 programsetptr
0274 0952 00537 movlw CMDISPINCRADDR ; Increment address command load into W
0275 02E8 00538 call commandisp ; Send command to PIC
0276 0A73 00539 decfsz LOADDR,F ; Decrement lower address
0277 02E7 00540 goto programsetptr ; Go back again
0278 0A73 00541 decfsz HIADDR,F ; Decrement high address
0279 0C03 00542 movlw .3 ; Place start pointer into W, offset address
027A 008B 00544 subwf TIMEHIGH,W ; Restore byte count into W
027B 002F 00545 movwf BYTECOUNT ; Place into byte counter
027C 0C12 00546 movlw STARTCALBYTE+2 ; Place start of REAL DATA address into W
027D 002E 00547 movwf ADDRPTR ; Update pointer
027E 00548 programisloop
027F 0C34 00549 movlw UPPER6BITS ; retlw instruction opcode placed into W
0280 0C10 00550 movlw STARTCALBYTE ; Place base address into W
0281 008E 00551 subwf ADDRPTR,w ; Offset by STARTCALBYTE
0282 0024 00552 movf FSR,5 ; Shift bits 4,5 to 5,6
0283 0584 00553 bsf FSR,4 ; Shift bits 4,5 to 5,6
0284 05A4 00554 bcf FSR,5
0285 06A4 00555 btfsc FSR,4
0286 05A4 00556 bsf FSR,5
0287 0584 00557 bcf FSR,4
0288 0209 00558 movf HIBYTE, w ; Subtract programmed and read data
0289 0208 00559 movf LOBYTE, w ; Subtract programmed and read data
028A 0040 00560 clrw ; Clear W reg
028B 01CB 00561 addwf PULSECNT,W ; now do 3 times overprogramming pulses
028C 01CB 00562 addwf PULSECNT,W
028D 01CB 00563 addwf PULSECNT,W
028E 002B 00564 movf ADDRPTR,w ; Offset by ADDRPTR
028F 002A 00565 movlw ADDRPTR ; Set up FSR to point to next value
0290 05B3 00566 baf STATUS, VFYES ; Set verify flag
0291 09B1 00567 call pgmvyfisp ; Program and verify this byte
0292 02AB 00568 incf PULSECNT,F ; Increment pulse counter
0293 0C19 00569 movlw .25 ; Place 25 count into W
0294 008B 0056A subwf PULSECNT,w ; Subtract pulse count from 25
0295 0643 0056B btfsc STATUS,Z ; Skip if NOT 25 pulse counts
0296 0A9A 0056C goto pgmispfail ; Jump to program failed - only try 25 times
0297 0209 0056D movf HIBYTE, w
0298 0087 0056E subwf HIBYTE, w
0299 0743 0056F btfsc STATUS,Z ; Skip if programmed is OK
029A 0A90 00570 goto pgmispcktloop ; Miscompare - program it again!
029B 020A 00571 movf LOBYTE, w
029C 0088 00572 subwf LOBYTE, w
029D 0743 00573 btfsc STATUS,Z ; Skip if programmed is OK
029E 0A90 00574 goto pgmispcktloop ; Miscompare - program it again!
029F 0040 00575 clrw ; Clear W reg
02A0 01CB 00576 addwf PULSECNT, W
02A1 01CB 00577 addwf PULSECNT, W
02A2 01CB 00578 addwf PULSECNT, W
02A3 002B 00579 movwf PULSECNT ; Add 3X pulse count to pulsecount
02A4 00580 pgmispcktloop
02A5 04E3 00581 bcf STATUS, VFYES ; Clear verify flag
02A6 0B91 00582 call pgmvyfisp ; Program this byte
02A7 02E8 00583 decfsz PULSECNT,F ; Decrement pulse counter, skip when done
02A8 0A90 00584 goto pgmispcktloop ; Miscompare - program it again!
02A9 0AAA 00585 goto pgmnextbyte ; Loop back and program again!
02AA 04E3 00586 pgmispfail
02AB 0952 00587 goto pgmnextbyte
02AC 00588 pgmnextbyte
02AD 0C06 00589 movlw CMDISPINCRADDR ; Increment address command load into W
02AE 0952 00590 call commandisp ; Send command to PIC

© 2000 Microchip Technology Inc.

DS00656B-page 4-15
02AC 02AE 00591   incf   ADDRPTR,F           ; Increment pointer to next address
02AD 02EF 00592   decfsz  BYTECOUNT,F         ; See if we sent last byte
02AE 0A7E 00593   goto    programisploop      ; Jump back and send next byte
02AF 0900 00594   call    poweroffisp         ; Done - power off PIC and reset it!
02B0 00595 self   goto    self                ; Done with programming - wait here!
00596
00597
00598
00599
00600 ; *******************************************************************
00601 ; * pgmvfyisp                                                       *
00602 ; * Program and/or Veryify a word in program memory on the          *
00603 ; * application PIC.  The data to be programmed is in HIDATA and    *
00604 ; * LODATA.                                                         *
00605 ; * RAM used: HIBYTE, LOBYTE, HIDATA, LODATA, TEMP                 *
00606 ; *******************************************************************
02B1 00607 pgmvfyisp
02B1 00608 loadcisp
02B2 0C02 00609   movlw   CMDISPLOAD          ; Place load data command into W
02B3 0952 00610   call    commandisp          ; Send load data command to PIC
02B4 0000 00611   nop                         ; Wait one cycle
02B5 0000 00612   nop                         ; Wait one cycle
02B6 0208 00613   nop                         ; Wait one cycle
02B7 0207 00614   movf    LODATA,w            ; Place LODATA byte into W
02B8 0000 00615   movwf   LOBYTE              ; Move it to LOBYTE reg
02B9 0207 00616   movf    HIDATA,w            ; Place HIDATA byte into W
02BA 0915 00617   movwf   HIBYTE              ; Move it to HIBYTE reg
02BB 0C08 00618   call    P16cispout          ; Send data to PIC
02BC 0C08 00619   movlw   CMDISPPGMSTART      ; Place start programming command into W
02BD 0952 00620   call    commandisp          ; Send start programming command to PIC
02BE 0000 00621   delay100us
02BF 0C20 00622   movlw   .32                 ; Place 32 into W
02C0 0000 00623   nop                         ; Wait one cycle
02C1 002D 00624   movwf   TEMP                ; Move it to TEMP for delay counter
02C2 02ED 00625   decfsz  TEMP,F              ; Decrement TEMP, skip when delay done
02C3 0C0E 00626   movlw   CMDISPPGMEND        ; Place stop programming command into W
02C4 0952 00627   call    commandisp          ; Send stop programming command to PIC
02C5 07E3 00628   btfss   STATUS,VFYYES     ; Skip if we are supposed to verify this time
02C6 0800 00629   retlw 0                     ; Done - return!
02C7 0000 00630   nop                         ; Wait one cycle
02C8 0000 00631   readcisp
02C9 0C04 00632   movlw   CMDISPREAD          ; Place read data command into W
02CA 0952 00633   call    commandisp          ; Send read data command to PIC
02CB 092C 00634   call    P16cispin           ; Read programmed data
02CC 0800 00635   retlw 0                     ; Done - return!
00636
00637
00638
MEMORY USAGE MAP ('X' = Used, '-' = Unused)

<table>
<thead>
<tr>
<th>Address</th>
<th>Bits Used/Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>XX</td>
</tr>
<tr>
<td>0040</td>
<td>XX</td>
</tr>
<tr>
<td>0080</td>
<td>XX</td>
</tr>
<tr>
<td>00C0</td>
<td>X</td>
</tr>
<tr>
<td>0200</td>
<td>XX</td>
</tr>
<tr>
<td>0240</td>
<td>XX</td>
</tr>
<tr>
<td>0280</td>
<td>XX</td>
</tr>
<tr>
<td>02C0</td>
<td>-</td>
</tr>
<tr>
<td>07C0</td>
<td></td>
</tr>
<tr>
<td>0FC0</td>
<td></td>
</tr>
</tbody>
</table>

All other memory blocks unused.

Program Memory Words Used: 402
Program Memory Words Free: 1646

Errors : 0
Warnings : 0 reported, 0 suppressed
Messages : 2 reported, 0 suppressed
APPENDIX B:
MPASM 01.40.01 Intermediate ISPTEST.ASM 3-31-1997 10:55:57 PAGE 1

LOC OBJECT CODE LINE SOURCE TEXT

00001 ; Filename: ISPTEST.ASM
00002 ; **********************************************
00003 ; * Author: John Day *
00004 ; * Sr. Field Applications Engineer *
00005 ; * Microchip Technology *
00006 ; * Revision: 1.0 *
00007 ; * Date August 25, 1995 *
00008 ; * Part: PIC16CXX *
00009 ; * Compiled using MPASM V1.40 *
0010; **********************************************
0011 ; * Include files: *
0012 ; * P16CXX.ASM *
0013 ; **********************************************
0014 ; * Fuses: OSC: XT (4.0 Mhz xtal) *
0015 ; * WDT: OFF *
0016 ; * CP: OFF *
0017 ; * PWRT: OFF *
0018 ; **********************************************
0019 ; * This program is intended to be used as a code example to *
0020 ; * show how to communicate with a manufacturing test jig that *
0021 ; * allows this PIC16CXX device to self program. The RB6 and RB7 *
0022 ; * lines of this PIC16CXX device are used to clock the data from *
0023 ; * this device to the test jig (running ISPPRGM.ASM). Once the *
0024 ; * PIC16C58 running ISPPRGM in the test jig receives the data, *
0025 ; * it places this device in test mode and programs these parameters. *
0026 ; * The code with comments "TEST -" is used to create some fakecalibration *
0027 ; * parameters that are first written to addresses STARTCALBYTE through *
0028 ; * ENDCALBYTE and later used to call the self-programming algorithm. *
0029 ; * Replace this code with your parameter calculation procedure, *
0030 ; * placing each parameter into the STARTCALBYTE to ENDCALBYTE *
0031 ; * file register addresses (16 are used in this example). The address *
0032 ; * "lookuptable" is used by the main code later on for the final lookup *
0033 ; * table of calibration constants. 16 words are reserved for this lookup *
0034 ; * table *
0035 ; **********************************************
0036 ; * Program Memory: *
0037 ; * 49 Words - communication with test jig *
0038 ; * 17 Words - calibration look-up table (16 bytes of data) *
0039 ; * 13 Words - Test Code to generate Calibration Constants *
0040 ; * RAM Memory: *
0041 ; * 16 Bytes -Temporary- Store 16 bytes of calibration constant *
0042 ; * 4 Bytes -Temporary- Store 4 bytes of temp variables *
0043 ; **********************************************

Warning[217]: Hex file format specified on command line.
0045 list p=16C71,f=inhx8m
0046 include <p16C71.inc>
0051 LIST
0052 ; P16C71.INC  Standard Header File, Version 1.00 Microchip Technology, Inc.
0054 LIST
2007 3FF1 0047 __CONFIG _CP_OFF&_WDT_OFF&_XT_OSC&_PWRTzheimer"
00055; * Port B (RB0-RB7) bit definitions *
00056; ***********************************************************
00057#define     CLOCK   6 ; clock line for ISP
00058#define     DATA    7 ; data line for ISP
00059; Port pins RB0-5 are not used in this test program
00060
00061; ***********************************************************
00062; * RAM register usage definition *
00063; ***********************************************************
00064 CSUMTOTAL   EQU 0Ch ; Address for checksum var
00065 COUNT     EQU 0Dh ; Address for COUNT var
00066 DATAREG   EQU 0Eh ; Address for Data output register var
00067 COUNTDLY  EQU 0Fh ; Address for clock delay counter
00068
00069; These two symbols are used for the start and end address
00070; in RAM where the calibration bytes are stored. There are 16 bytes
00071; to be stored in this example; however, you can increase or
00072; decrease the number of bytes by changing the STARTCALBYTE or ENDCALBYTE
00073; address values.
00074
00075 STARTCALBYTE    EQU 10h     ; Address pointer for start CAL byte
00076 ENDCALBYTE      EQU 2Fh     ; Address pointer for end CAL byte
00077
00078; Table length of lookup table (number of CAL parameters to be stored)
00079
00080 CALTABLELENGTH EQU ENDCALBYTE - STARTCALBYTE + 1
00081
000     ORG 0
00082; ***********************************************************
00083; * testcode routine *
00084; ***********************************************************
00085; * TEST code - sets up RAM register with register address as data *
00086; * Uses file register STARTCALBYTE through ENDCALBYTE to store the*
00087; * calibration values that are to be programmed into the lookup *
00088; * table by the test jig running ISPPRGM. *
00089; * Customer would place calibration code here and make sure that *
00090; * calibration constants start at address STARTCALBYTE *
00091; ***********************************************************
0000     testcode
00010     movlw STARTCALBYTE ; TEST -
00011     movwf FSR             ; TEST - Init FSR with start of RAM addres
00012     looptestram
00013     movf FSR,W           ; TEST - Place address into W
00014     movwf INDF            ; TEST - Place address into RAM data byte
00015     incf FSR,F           ; TEST - Move to next address
00016     sublw ENDCALBYTE+1    ; TEST - Subtract from end of RAM
00017     btfss STATUS,Z        ; TEST - Skip if at END of ram
00018     goto looptestram     ; TEST - Jump back and init next RAM byte
00019     goto mainloop        ; TEST - Jump back to self since CAL is done
00020     lookuptable
00021     movf FSR,W           ; TEST - Place current address into W
00022     movwf INDF            ; TEST - Place address into RAM data byte
00023     incf FSR,F           ; TEST - Move to next address
00024     sublw ENDCALBYTE+1    ; TEST - Subtract from end of RAM
00025     btfss STATUS,Z        ; TEST - Skip if at END of ram
00026     goto looptestram     ; TEST - Jump back and init next RAM byte
00027     goto mainloop        ; TEST - Jump back to self since CAL is done
00028     lookuptable
00029     call lookuptable    ; TEST - Get first CAL value from lookup table
00030     sublw 0FFh           ; TEST - Check if lookup CAL table is blank
00031     btfsc STATUS,Z        ; TEST - Skip if table is NOT blank
00032     goto calsend         ; TEST - Table blank - send out cal parameters
00033     goto mainloop        ; TEST - Jump back to self since CAL is done
00034     mainloop
00035     goto mainloop        ; TEST - Jump back to self since CAL is done
00036; ***********************************************************
00037; * lookuptable *
00038; ***********************************************************
00039; * Calibration constants look-up table. This is where the CAL *
00040; * Constants will be stored via ISP protocol later. Note it is *
00041; * blank, since these values will be programmed by the test jig *
00042; * running ISPPRGM later. *
00043; * Input Variable:  W stores index for table lookup *
00044; * Output Variable: W returns with the calibration constant *
00119 ; * NOTE: Blank table when programmed reads "FF" for all locations *
00120 ; ********************************************************************************
000F 00121 lookuptable
000F 0782 00122 addwf PCL,F ; Place the calibration constant table here!
00123
002F 00124 ORG lookuptable + CALTABLELENGTH
002F 34FF 00125 retlw 0FFh ; Return FF at last location for a blank table
00126
00127 ; ********************************************************************************
00128 ; * calsend subroutine                                             *
00129 ; * Send the calibration data stored in locations STARTCALBYTE     *
00130 ; * through ENDCALBYTE in RAM to the programming jig using a serial* *
00131 ; * clock and data protocol                                        *
00132 ; *     Input Variables:   STARTCALBYTE through ENDCALBYTE         *
00133 ; ********************************************************************************
0030 00134 calsend
0030 018C 00135 clrf CSUMTOTAL ; Clear CSUMTOTAL reg for delay counter
0031 018D 00136 clrf COUNT ; Clear COUNT reg to delay counter
0032
0033 00138 delayloop ; Delay for 100 mS to wait for prog jig wakeup
0034 303F 00139 movlw b'00111111' ; RB6,7 set to outputs
0035 0186 00140 movwf TRISB ; Move to TRIS registers
0036 1683 00141 bsf STATUS,RP0 ; Switch to bank 1
0037 3001 00142 movlw high lookuptable+1 ; place MSB of first addr of cal table into W
0038 204D 00143 call sendcalbyte ; Send the high address out
0039 3010 00144 movlw low lookuptable+1 ; place LSB of first addr of cal table into W
0040 204D 00145 call sendcalbyte ; Send low address out
0041 3010 00146 movlw STARTCALBYTE ; Place RAM start address of first cal byte
0042 0084 00147 movwf FSR ; Place this into FSR
0043 00148 loopcal
0043 0800 00149 movf INDF,W ; Place data into W
0044 204D 00150 call sendcalbyte ; Send the byte out
0045 0D8E 00151 rlf DATAREG,F ; Rotate to next bit
0046 1C03 00152 btfss STATUS,C ; Skip if the data bit was high
0047 0186 00153 clrf PORTB ; clear out port pins
0048 00154 loopcalsend
0048 0800 00155 movf INDF,W ; Place data into W
0049 204D 00156 call sendcalbyte ; Send the byte out
004A 0084 00157 incf FSR,F ; Move to the next cal byte
004B 0080 00158 movf FSR,W ; Place byte address into W
004C 3C30 00159 sublw ENDCALBYTE+1 ; Set Z bit if we are at the end of CAL data
004D 1D03 00160 btfs STATUS,Z ; Skip if we are done
004E 2842 00161 goto loopcal ; Go back for next byte
004F 00C 00162 movf CSUMTOTAL,W ; place checksum total into W
0050 0D3 00163 call sendcalbyte ; Send the checksum out
0051 0186 00164 clrf PORTB ; clear out port pins
0052 00165 calsenddone
0052 284C 00166 goto calsenddone ; We are done - go home!
0053
0054 00168 ;================================================================================
0055 00169 ; * sendcalbyte subroutine                                          *
0056 00170 ; * Send one byte of calibration data to the programming jig       *
0057 00171 ; *     Input Variable: W contains the byte to be sent              *
0058 00172 ;================================================================================
0059 00173 sendcalbyte
0059 008E 00174 movwf DATAREG ; Place send byte into data register
005A 078C 00175 addwf CSUMTOTAL,F ; Update checksum total
005B 3008 00176 movlw .8 ; Place 8 into W
005C 008D 00177 movwf COUNT ; set up counter register
005D 0178 loopsendcal
005D 1706 00179 bsf PORTB,CLOCK ; Set clock line high
005E 205C 00180 call delaysend ; Wait for test jig to synch up
005F 0D8E 00181 rlf DATAREG,F ; Rotate to next bit
0060 1786 00182 bsf PORTB,DATA ; Assume data bit is high
0061 1C03 00183 btfs STATUS,C ; Skip if the data bit was high
bcf PORTB,DATA ; Set data bit to low
bcf PORTB,CLOCK ; Clear clock bit to clock data out
call delaysend ; Wait for test jig to synch up
decfsz COUNT,F ; Skip after 8 bits
goto loopsendcal ; Jump back and send next bit
return ; We are done with this byte so return!

; ******************************************************************
; * delaysend subroutine                                           *
; * Delay for 50 ms to wait for the programming jig to synch up    *
; ******************************************************************
movlw 10h ; Delay for 16 loops
movwf COUNTDLY ; Use COUNTDLY as delay count variable
decfsz COUNTDLY,F ; Decrement COUNTDLY and skip when done
go to loopsendcal ; Jump back for more delay
return

MEMORY USAGE MAP ('X' = Used, '-' = Unused)

Program Memory Words Used:   66
Program Memory Words Free:   958

Errors :  0
Warnings : 1 reported,  0 suppressed
Messages : 1 reported,  0 suppressed