INTRODUCTION

This application note describes how to use a PIC16F877A as an I^2^C™ master to communicate with the Microchip MCP23016 I^2^C I/O Expander slave device.

An I/O Expander device is used to increase the I/O capability of a microcontroller (refer to Figure 1). A microcontroller’s I^2^C port can be used as a communication channel with MCP23016(s) to expand the microcontroller’s I/O count. By using two I^2^C pins (and one general-purpose I/O pin, if using the interrupt capability of the MCP23016), 16 to 128 general-purpose I/Os can be gained. The MCP23016 has three address pins which can be used to provide unique addresses for up to eight devices.

Each device attached to the I^2^C bus must be assigned a unique address unless all devices (with the same address) are receiving the same data and do not transmit any data. When the master initiates a data transfer, the address of the slave device is transmitted. Within the address, the Lsb (R/W bit) specifies whether the master reads from, or writes to, the slave. For write operations, a series of bytes would be transmitted from the master. For read operations, the master waits for the bus to be free (i.e., SCL line not pulled low) and then clocks the data to be received from the slave.

What can you do with an I/O Expander?

An I/O Expander can also be used to monitor switches and/or sensors, drive LEDs and/or relays, as well as other general-purpose I/O functions. An I/O Expander can have several uses in a variety of applications. Typical applications include high-side MOSFET load-switch drivers in power-management systems and keyboards.

The I^2^C Bus Specification

This application note does not discuss the I^2^C specification in detail. Refer to the following documents (www.microchip.com) for more information on the I^2^C specification and implementation:

- AN735, “Using the PICmicro MSSP Module for Master I^2^C Comm”, DS00735.
- MCP23016 Datasheet, DS20090.

For complete I^2^C bus specifications, refer to the Philips®/Signetics® document, “The I^2^C Bus and How to Use It”.

FIGURE 1: BLOCK DIAGRAM
INITIATING AND TERMINATING DATA TRANSFER

During times of no data transfer (idle time), both the SCL and SDA lines are pulled high via pull-up resistors. A master device takes control of the bus during bus idle by generating a START condition. A START is defined as a high-to-low transition of SDA when SCL is high. When the master has completed all data, it releases the bus by generating a STOP condition. A STOP is defined as a low-to-high transition of SDA while SCL is high. Because the START and STOP conditions are defined as transitions of the SDA when the SCL line is high, the SDA line can only change when SCL is low during the actual data transmission. Figure 2 shows the relationship between SCL and SDA for the various conditions. A REPEATED-START condition (Figure 3) is generated by the master to maintain control of the bus while switching between write mode and read mode and/or while in multi-master environments. To generate a REPEATED-START, both SDA and SCL start low. SDA is then asserted high, followed by SCL being asserted high. Finally, SDA is asserted low while SCL is high.

Addressing the MCP23016 I/O Expander

I2C devices can be addressed in two different modes: 10-bit addressing or 7-bit addressing modes. The MCP23016 uses the 7-bit addressing, as shown in Figure 4 and Figure 5. Therefore, this application note will only be using the 7-bit addressing mode. To understand how the MCP23016 works, refer to the MCP23016 datasheet, DS20090.

FIGURE 4: THE 7-BIT ADDRESS FOR THE MCP23016

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

FIGURE 5: THE 7-BIT ADDRESS FORMAT

- **S** - Start Condition
- **R/W** - Read/Write bit
- **ACK** - Acknowledge

---

**Notes:**

- MSb: Most Significant Bit
- LSb: Least Significant Bit

---

**FIGURE 2:** START AND STOP CONDITIONS

<table>
<thead>
<tr>
<th>SDA</th>
<th>SCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 3:** REPEATED-START CONDITION

- SDA = 1, SCL = 1
- 1st Bit
- Falling edge of ninth clock
- End of Xmit
- Sr = Repeated START

---

**FIGURE 4:** THE 7-BIT ADDRESS FOR THE MCP23016

0 1 0 0 A2 A1 A0

**FIGURE 5:** THE 7-BIT ADDRESS FORMAT

- **S** - Start Condition
- **R/W** - Read/Write bit
- **ACK** - Acknowledge
WRITING TO THE MCP23016 I/O EXPANDER

During write mode (Figure 6), all data is transmitted as bytes, with no limit to the number of bytes transmitted per data transfer. After each byte, the slave (receiver) generates an acknowledge bit (ACK) by pulling the SDA line low. If a slave doesn’t acknowledge the slave address or received data, the master aborts the transfer. Whether the ACK bit is generated or not, the SDA line must be released by the slave so the master can generate the STOP condition. The protocol used to communicate with the MCP23016 is simple for a write operation. First, a START condition is generated, followed by the slave address with the LSb=0 (R/W=0). The command byte is then transmitted (the command byte is like an address pointer. It gives the address of the register to be written to). This is followed by the first data byte which is written to the first byte of the register pair addressed by the command byte. Finally, the second data byte is written to the second byte of the register pair. This can be followed by more data byte pairs or by a STOP condition.

Although writing data from the master to the slave is very simple, some safeguards need to be implemented. To ensure proper functioning of the device, follow the master writing sequence flowchart, as shown in Figure 7.

Note: A 12 µs delay is required after every 9th clock on SCL to allow the MCP23016 time to process the contents on SDA. This is in addition to any SCL hold times the MCP23016 may require (See Figure 6).
FIGURE 6: TYPICAL $I^2C$ WRITE TRANSMISSION FORMAT

- Address
- Command Byte
- Data 1
- Data 2
- Data on GP0
- Data on GP1
- SCL held low until data is processed
- DATA VALID
- $t_{GPV0}$
- $t_{GPV1}$
- DATA
- VALID
- $t_{ADH}$
FIGURE 7: MASTER WRITING SEQUENCE FLOWCHART

Start
- Wait for $^2$C™ bus to be Idle: (Note 1)
  ACKEN=RCEN=PEN=RSEN=SEN=R/W=0

  Issue a START condition:
  BSF SSPCON2,SEN

  Wait for $^2$C bus to be Idle:
  ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
  Then wait 12 µs. (Note 2)

Address
- Load SSPBUF with MCP23016 address (lsb=0)

Command
- Load SSPBUF with Command byte to point the register to be written to

  Wait for $^2$C bus to be Idle:
  ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
  Then wait 12 µs. (Note 2)

Data1
- Load SSPBUF with Data1

  Wait for $^2$C bus to be Idle:
  ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
  Then wait 12 µs. (Note 2)

Data2
- Load SSPBUF with Data2

  Wait for $^2$C bus to be Idle:
  ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
  Then wait 12 µs. (Note 2)

Stop
- Issue a STOP condition:
  BSF SSPCON2,PEN

Note 1: The master needs to wait for $^2$C bus idle to indicate that the MSSP module has finished its last task. The SSPIF interrupt could be used instead of the wait for idle (the interrupt is not used in this application note).

2: A 12 µs delay is inserted, as stated in the MCP23016 datasheet. If a 12 µs delay is not inserted, it can lead to the malfunction of the MCP23016. This could be caused either by receiving false data or by locking up the $^2$C bus all together.

3: A NACK is issued to the slave-transmitter that the master wants to stop receiving data from it. If a NACK is not received by the slave-transmitter, it will not reset the internal state machine following a STOP or a REPEATED-START. This would cause the MCP23016 to lock up.
READING FROM THE MCP23016 I/O EXPANDER

FIGURE 8: TYPICAL READ TRANSMISSION/RECEPTION FORMAT

When the master is receiving data during read mode (Figure 8), it generates an acknowledge signal for each received byte of data except for the last byte. To signal the end of data to the slave-transmitter, the master generates a “not acknowledge” (NACK) condition (ACK=1). This is very important for proper functioning of read mode. The slave then releases the SDA line so the master can generate the STOP condition. If the slave needs to delay the transmission of the next byte, it may hold the SCL line low to force the master into a wait state. Data transfer continues when the slave releases the SCL line. This allows the slave to move the received data or fetch the data it needs to transfer before allowing the clock to start again. This wait state technique can also be implemented at the bit level.

The protocol used to communicate with the MCP23016 for a read operation is as follows: A START condition is generated, followed by the slave address with the LSb=0 (R/W=0 to indicate a write condition). The command byte is then transmitted (the command byte is like an address pointer. It gives the address of the register to read from). This is followed by a REPEATED-START condition. The slave address is transmitted again but with the LSB=1 (R/W=1 to indicate a read condition). The master waits for the slave to release the SCL before beginning to clock the first data byte. After receiving the first data byte, the master waits again for the SCL to be released before clocking the second data byte. This can be followed by more data bytes as long as the master keeps acknowledging after each data byte. At the end of the last data byte, the master needs to generate a NACK before issuing a STOP or REPEATED-START condition.

Although reading data from the slave by the master is very simple, some safeguards need to be implemented.

To ensure proper functioning of the device, follow the master reading sequence flowchart shown in Figure 9.
FIGURE 9: MASTER READ SEQUENCE FLOWCHART

Start

Wait for I²C™ bus to be Idle: (Note 1)
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0

Issue a start condition:
BSF SSPCON2,SEN

Address Write

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
Then wait 12 µs. (Note 2)

Load SSPBUF with MCP23016
address (lsb=0)

Command

Load SSPBUF with
Command byte to point the
register to be written to

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
Then wait 12 µs. (Note 2)

Re-Start

Issue a repeated start condition:
BSF SSPCON2,RSEN

Address Read

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
Then wait 12 µs. (Note 2)

Load SSPBUF with MCP23016
address (lsb=1)

Data1

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
Then wait 12 µs. (Note 2)

Set receive enable bit
BSF SSPCON2,RCEN

Read SSPBUF

Continued on next page

Note 1: The master needs to wait for I²C bus idle to indicate that the MSSP module has finished its last task. The SSPIF interrupt could be used instead of the wait for idle (the interrupt is not used in this application note).

2: A 12 µs delay is inserted, as stated in the MCP23016 datasheet. If a 12 µs delay is not inserted, it can lead to the malfunction of the MCP23016. This could be caused either by receiving false data or by locking up the I²C bus all together.

3: A NACK is issued to the slave-transmitter that the master wants to stop receiving data from it. If a NACK is not received by the slave-transmitter, it will not reset the internal state machine following a STOP or a REPEATED-START. This would cause the MCP23016 to lock up.
FIGURE 10: MASTER READ SEQUENCE FLOWCHART (CONTINUED)

Continued on previous page

Wait for I²C™ bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W
all equal to zero

Issue an acknowledge:
BCF SSPCON2.ACKDT

Enable acknowledge sequence:
BSF SSPCON2.ACKEN

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SE=R/W=0
Then wait 12 µs. (Note 2)

Set Receive Enable Bit
BSF SSPCON2.RCEN

Read SSPBUF

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
Then wait 12 µs. (Note 2)

Issue a not-acknowledge: (Note 3)
BSF SSPCON2.ACKDT

Enable acknowledge sequence:
BSF SSPCON2.ACKEN

Wait for I²C bus to be Idle:
ACKEN=RCEN=PEN=RSEN=SEN=R/W=0
Then wait 12 µs.

Issue a stop condition:
BSF SSPCON2,PEN

Note 1: The master needs to wait for I²C bus idle to indicate that the MSSP module has finished its last task. The SSPIF interrupt could be used instead of the wait for idle (the interrupt is not used in this application note).

2: A 12 µs delay is inserted, as stated in the MCP23016 datasheet. If a 12 µs delay is not inserted, it can lead to the malfunction of the MCP23016. This could be caused either by receiving false data or by locking up the I²C bus all together.

3: A NACK is issued to the slave-transmitter that the master wants to stop receiving data from it. If a NACK is not received by the slave-transmitter, it will not reset the internal state machine following a STOP or a REPEATED-START. This would cause the MCP23016 to lock up.
INTERFACE CODE

For this application, the PIC16F877A provides an MSSP module for I\(^2\)C communication.

The firmware code for this application is written in C using the Hi-Tech PICC™ C Compiler and is available on Microchip’s website (www.microchip.com). Table 1 provides a list of source code files.

**TABLE 1: MASTER I\(^2\)C ‘C’ SOURCE CODE FILES**

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16chaser.c</td>
<td>Main code loop and interrupt handler.</td>
</tr>
<tr>
<td>delay.c</td>
<td>Delay routines.</td>
</tr>
<tr>
<td>delay.h</td>
<td>Delay function prototypes.</td>
</tr>
<tr>
<td>I2Crxtx.h</td>
<td>Hardware I(^2)C master routines for PIC16F877A.</td>
</tr>
<tr>
<td>pic.h</td>
<td>Required by compiler for SFR declarations.</td>
</tr>
<tr>
<td>I2Crxtx.h</td>
<td>Hardware I(^2)C master function prototypes.</td>
</tr>
</tbody>
</table>

The CCP2 module in the PIC16F877A is being used to generate an interrupt on every falling edge of the MCP23016 interrupt pin.

**Master implementation**

The master device (PIC16F877A), upon completion of the internal power-up cycle, performs some basic peripheral and variable initialization. The ADC module is disabled, I/O ports are configured, the MSSP module is configured for master I\(^2\)C mode with 400 kHz baud rate and slew rate is enabled and the CCP2 module is configured to interrupt on the falling edge. Once the peripheral initialization is completed, peripheral and global interrupts are enabled. A small delay is then introduced to allow the MCP23016 to complete its internal power-up cycle. The I/O Expander is then initialized by setting the ports to predefined values, changing the direction of the port pins and changing the polarity. The main code execution loop is then entered (see Figure 11). In the main loop, the *Check_CCP_status()* routine is called, followed by a small delay. The variables “i” and “j” are transmitted to the MCP23016. They are then shifted one to the right and the other to the left and vice versa to get an LED chaser light effect.

When an interrupt occurs on the CCP2 module, the CCP2IF bit is cleared and a software flag is set (see Figure 12). The software flag is used in case of an interrupt occurring while the master is transmitting or receiving, the task being performed will finish before servicing the interrupt.

*Check_CCP_status()* checks if the software interrupt has been set. If the flag is not set, the code goes back to “main”. If the flag is set, the flag is cleared and *GetNewValue()* is called (see Figure 13). *GetNewValue()* is a function that reads the MCP23016 port values and then displays them on the LEDs (see Figure 14). Then it checks if the interrupt pin is still low. If low, the code reads the MCP23016 port values again. If high, the code goes back to the main loop. Refer to Table 2 for a description of all the functions used in the source code.
### TABLE 2: FUNCTIONS USED IN THE SOURCE CODE

<table>
<thead>
<tr>
<th>Functions</th>
<th>Input</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I²C_init()</td>
<td>None</td>
<td>None</td>
<td>Initializes the SSP module for I²C master communication.</td>
</tr>
<tr>
<td>I²C_waitForIdle()</td>
<td>None</td>
<td>None</td>
<td>Waits for the I²C bus to be idle.</td>
</tr>
<tr>
<td>I²C_start()</td>
<td>None</td>
<td>None</td>
<td>Issues a START condition.</td>
</tr>
<tr>
<td>I²C_repStart()</td>
<td>None</td>
<td>None</td>
<td>Issues a REPEATED-START condition.</td>
</tr>
<tr>
<td>I²C_stop()</td>
<td>None</td>
<td>None</td>
<td>Issues a STOP condition.</td>
</tr>
<tr>
<td>I²C_read(ack)</td>
<td>ack,</td>
<td>Read data</td>
<td>Enables Receive mode, reads received data from SSPBUF. ack=0, don't acknowledge - ack=1, acknowledge</td>
</tr>
<tr>
<td></td>
<td>acknowledge bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I²C_write(I²CWriteData)</td>
<td>I²CWriteData, byte to be transmitted</td>
<td>1 if ACK 0 if NACK</td>
<td>Loads data to be transmitted to the slave, into the SSPBUF and returns a ‘1’ if transmission acknowledged.</td>
</tr>
<tr>
<td>write_to_MCP(address,cmd ,data1,data2)</td>
<td>Start,address, Command, Data1, Data2,Stop</td>
<td>None</td>
<td>Implements a full write sequence from START to STOP.</td>
</tr>
<tr>
<td>GetNewValue()</td>
<td>None</td>
<td>None</td>
<td>Reads value from slave and displays it on PORTB and PORTD.</td>
</tr>
<tr>
<td>interrupt isr()</td>
<td>None</td>
<td>None</td>
<td>Interrupt service routine for CCP module.</td>
</tr>
<tr>
<td>Check_CCP_status()</td>
<td>None</td>
<td>None</td>
<td>Checks if a CCP interrupt occurred and, if so, calls GetNewValue().</td>
</tr>
<tr>
<td>DelayUs(x)</td>
<td>x = # of µs</td>
<td>None</td>
<td>Delay in microseconds (used in Delay.c only).</td>
</tr>
<tr>
<td>DelayUsx(x)</td>
<td>x = # of µs</td>
<td>None</td>
<td>Delay in microseconds with x = number of microseconds.</td>
</tr>
<tr>
<td>DelayMs(x)</td>
<td>x = # of msec.</td>
<td>None</td>
<td>Delay in milliseconds with x = number of milliseconds.</td>
</tr>
<tr>
<td>DelaySec(x)</td>
<td>x = # of sec.</td>
<td>None</td>
<td>Delay in seconds with x = number of seconds.</td>
</tr>
</tbody>
</table>
FIGURE 11: MAIN LOOP FLOWCHART

1. Initialize the PIC16F877
2. Wait for MCP23016
3. Initialize the MCP23016 Registers
   - I=1, J=128
4. While (I<=128)
   - (check_CCP_status(); write I and J to MCP23016; DelayMs(100); Shift I to the left; Shift J to the right;)
5. While (I<=1)
   - (check_CCP_status(); write I and J to MCP23016; DelayMs(100); Shift I to the right; Shift J to the left;)

FIGURE 12: INTERRUPT SERVICE ROUTINE FLOWCHART

1. Is CCP2IE=1 & CCP2IF=1?
   - Yes
   - Clear Hardware Flag: BSF PIR2, CCP2IF2
   - And Set Software Flag to indicate that an interrupt occurred
   - Flags = 0x01
   - Return
   - No

2. Is Flags=0x01?
   - Yes
   - Call GetNewValue() sub-routine
   - No
   - Return

3. Clear Software Flag: Flags=0x00;

FIGURE 13: CHECK_CCP_STATUS() SUB-Routine FLOWCHART

1. Is RC1=0 (is the interrupt still set?)
   - Yes
   - DelayUs(250);
   - No

2. Is RC1=0 (is the interrupt still set?)
   - Yes
   - Make PORTB=LSB;
   - Make PORTD=MSB;
   - No
   - Return

FIGURE 14: GETNEWVALUE() SUB-Routine FLOWCHART

1. Save LSB_old, Save MSB_old
2. Read new LSB, and MSB
3. Is LSB_old=LSB; and MSB_old=MSB;
   - Yes
   - Is RC1=0 (is the interrupt still set?)
     - Yes
     - Make PORTB=LSB; Make PORTD=MSB;
     - No
     - Return
   - No

© 2003 Microchip Technology Inc.
TEST BOARD
See Appendix A for a complete schematic diagram of the test board.

SUMMARY
When more I/O ports are needed by a microcontroller (e.g., keyboard, sensors, LEDs, relays, high-side MOS-FET load-switch drivers, etc.), the MCP23016 is a low-cost, simple solution. The Master Synchronous Serial Port (MSSP) embedded on the PIC16F877A, and many of the PICmicro® devices, provide I²C communication for use with the MCP23016 16-bit I/O Expander.

This application note demonstrates how to interface the MCP23016 with the PIC16F877A, how the device protocol functions and how to implement it in software. The code used in this application note will work with any PICmicro microcontroller containing an MSSP module. Some code modification may be necessary. The MCP23016 is also compatible with other MCUs that have I²C modules. The test board example (Appendix A) helps in testing and understanding the way the MCP23016 I/O Expander works.
APPENDIX A

I/O EXPANDER TEST BOARD

INTRODUCTION

The I/O Expander test board is equipped with a Microchip MCP23016 I^2C I/O Expander slave device connected to the I^2C port and the interrupt pin (SDA, SCL, and INT) of the PIC16F877A, which is configured as an I^2C master.

Inputs / Outputs

MCP23016

The MCP23016 has 16 bidirectional I/Os (GP0.0-GP0.7 and GP1.0-GP1.7). Pins GP0.0 to GP0.7 are connected to LEDs D1 through D8 and Switch S1. Pins GP1.0 to GP1.7 are connected to LEDs D9 through D16 and Switch S3. This configuration allows the use of the pins as either inputs or outputs. The only drawback with this configuration is that it makes the I/O pins active low for both inputs and outputs (refer to Figure 16 and Figure 17). However, this scheme makes it more convenient for testing. All the pins have 4.7 kΩ pull-up resistors (RP1 and RP2).

PIC16F877A

The PIC16F877A is connected to LEDs in a similar fashion to the MCP23016. PORTB is used as the LSB and is connected to LEDs D17 through D24 and switch S4. PORTD is used as the MSB and is connected to LEDs D25 through D32 and switch S6.

ADDRESS PINS

The MCP23016 has three hardware address pins that are used to allow up to eight MCP23016 devices on the same I^2C bus. The MCP23016 address pins are connected to switch S2. This is done for testing purposes to allow change of addresses using the switch. These pins can be tied permanently high or low.

INTERRUPT PIN

The interrupt pin from the MCP23016 is tied to the capture input pin of the PIC16F877A (RC1/CCP2) and is pulled high using a 1 kΩ resistor (R98).

I^2C PINS

The I^2C pins (SDA and SCL) from all devices (U1 and U2) are connected together to form the I^2C bus. Both SDA and SCL are pulled high using two 4.7 kΩ resistors (R100 and R101).

FIGURE 15: TEST BOARD CONNECTIONS

![Diagram of test board connections](https://example.com/diagram.png)
FIGURE 16: MCP23016 I/O EXPANDER SCHEMATIC (SHEET 1 OF 2)
FIGURE 17: MCP23016 I/O EXPANDER SCHEMATIC (SHEET 2 OF 2)
APPENDIX B

I/O EXPANDER INTERFACE CODE

Master Firmware

The master firmware initializes the variables, ports, A/D module, I²C module and CCP module of the PIC16F877A, then waits for the MCP23016 Power-On-Reset to initialize the I/O Expanders registers. The code enters an infinite loop which checks if a CCP2 interrupt has occurred, then transmits two bytes ("i" and "j") which implement an "LED chaser" type of pattern.

If an interrupt occurs, the code jumps to the interrupt subroutine which clears the CCP2 interrupt flag (CCP2IF=0) and sets a software flag (Flags=0x01) so that the interrupt can be serviced later. When the Check_CCP_status() routine is called, it checks the software flag. If the software flag is clear, the code returns to the "main". If the software flag is set, the code jumps to GetNewValue() routine which reads the port values from the slave device and displays the read values on PORTB (LSB) and PORTD (MSB).

The firmware is setup to read from and write to the same address. It is possible to have the MCP23016 connected with another I/O Expander device on the same address and write to both of them at the same time. A second MCP23016 with a different address can be added with some code modification.
Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products.

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip’s products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Trademark

The Microchip name and logo, the Microchip logo, KEELLOQ, MPLAB, PIC, PICmicro, PICSTART, PRO MATE and PowerSmart are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, microID, MXDEV, MXLAB, PICMASTER, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Accuron, dsPIC, dsPICDEM.net, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, PICC, PICT, PICDEM, PICDEM.net, PowerCal, PowerInfo, PowerTool, rPIC, Select Mode, SmartSensor, SmartShunt, SmartTel and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2003, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.