INTRODUCTION

The PIC18C601 and PIC18C801 microcontrollers are the first members of Microchip’s PIC18 family with no on-chip program memory. They offer the PIC18 architecture, with the ability to use different types and sizes of external program memory (up to 2 Mbyte) to exactly fit most applications.

In modern embedded applications, where features and functionality are constantly evolving, FLASH memory is an ideal choice for external program memory. Field upgradability is almost always desirable in these systems, too. Most commonly available FLASH devices, however, disable read access while being programmed or erased. They also require special command sequences for programming, and have longer erase and write times than read times. As a result, systems using FLASH technology require either a second memory device, or a microcontroller with built-in memory space, in order to implement field reprogrammability. PIC18C601/801 controllers do this by allowing part of on-chip data memory to be reconfigured as program memory.

To implement reprogrammability, the user must incorporate into their design, a bootloader — a firmware mechanism that allows a new user application program to be written to the system. The bootloader firmware system must be able to recognize that new user code is available and initiate itself (“invocation”), receive the new code from some communication channel in manageable segments and check it for communication errors (“communication”), and program the memory with the new data and without errors (“programming”). It must also be flexible enough to be able to incorporate new programming methods, as new FLASH devices become available.

PROGRAMMING A ROMLESS SYSTEM: OVERVIEW

PIC18C601/801 controllers offer no on-chip program memory. In normal operation, program instructions are fetched and executed directly from the external memory. These microcontrollers also offer 1.5 Kbytes of on-chip data memory. Of this, the last 512 bytes are designated as “Boot RAM”. This block can be configured to act as either data or program memory; when set as program memory, it provides the system designer a way to program external FLASH devices without the need for additional hardware. The memory maps for the controllers, showing Boot RAM enabled and disabled, are presented in Figures 1 and 2.

When programs are executed from Boot RAM, the system bus and all of its control signals are deactivated. If required, the external system bus may be disabled and turned into I/O port signals. While the Boot RAM is enabled, any attempts to read or write to it are ignored. Any TBLWT instructions attempted to addresses in the Boot RAM space result in an external table write to the external memory, instead. Similarly, TBLRD instructions on the Boot RAM space, are performed on the external memory.
A typical bootloader using the Boot RAM performs the following steps:

1. Disable Boot RAM.
2. Transfer the programmer routine of the bootloader program from the external program memory to the Boot RAM, using TBLRD and MOVWF instructions.
3. Enable the Boot RAM.
4. Execute the programmer routine as a data block is received.
5. Perform the necessary programming on the external memory by either executing the necessary TBLRD and TBLWT instructions, or by switching the system bus to I/O ports.
6. Continue to execute the programmer routine from Boot RAM as data blocks are received.
7. Jump to a known valid external program memory location.
8. Reset the system when all data is programmed.
FIGURE 1: MEMORY MAP AND PROGRAM STACK FOR THE PIC18C801

PGRM = ‘0’ (Boot RAM disabled)

PGRM = ‘1’ (Boot RAM enabled)

RESET Vector 0000h
High Priority Interrupt Vector 0008h
Low Priority Interrupt Vector 0018h

External Program Memory

On-Chip Boot RAM
1FFE00h
1FFFDFFh
1FFFFFh

Internal Memory

External Table Memory
1FFE00h
1FFDFFh
1FFFFFh

User Memory Space
FIGURE 2: MEMORY MAP AND PROGRAM STACK FOR THE PIC18C601

PGRM = '0'
(Boot RAM disabled)

PGRM = '1'
(Boot RAM enabled)
GENERAL REQUIREMENTS FOR THE BOOTLOADER

When implementing any in-system programmer, the most basic requirement is that the system be able to perform a large amount of memory programming without error. Other key points to be considered for the design are:

- Providing an option to enter Bootloader mode or execute the existing application code
- Allowing for the use of the most popular file formats for programming (such as INHX8 and INHX32)
- Implementing a robust communication protocol between the data source and the firmware, to divide the data into manageable packets with the required address and error detection information
- Providing the means for reading and verifying programmed data
- Creating a design that is sufficiently modular and flexible, to support new programming algorithms, as well as override and debug the default programmer

In creating the reference design for this application note, we decided that a flexible and robust system would have three key components.

- **Host software:** This component should reside on a separate (PC) system from the programming target. It should provide a general purpose interface to the target’s on-board programming firmware, to allow the download of user selected Intel® HEX or HEX 32 format files. It should also support other device specific programming commands, such as Device Erase. Finally, it should use a robust communication protocol for error-free data transfer.

- **Core bootloader firmware:** This firmware component should detect if new user code is available for programming. If so, it should manage the receipt of new code from the host software, loading of the appropriate firmware to Boot RAM, and transfer of program execution to Boot RAM. If new code is not available, it should transfer program execution directly to existing user code.

- **Programmer firmware:** This firmware component should handle the actual programming of external memory. If an algorithm other than the default FLASH programmer is required, it should be downloadable from the host software.

THE HOST SOFTWARE

There are many ways to download new user code to a device. To demonstrate the flexibility of the programming system, the reference model uses a host software application, running on an external system (in this case, an IBM® compatible PC). This provides the ability to handle multiple file formats and FLASH device families, as well as take care of other device management tasks. Users may opt to use other methods, such as transferring code from EEPROMs, or downloading by modem from the Internet.

The host software for the reference design is a 32-bit application, designed to run under Microsoft® Windows® operating system. The application runs all commands from one window, using a standard Windows compatible GUI. It is compatible with all 32-bit Microsoft operating systems, and may be installed on Windows NT® and Windows 2000 systems without Administrator privileges.

A brief description of the host software and its user interface is provided in Appendix E. Users interested in further investigation are encouraged to download the application code and experiment further.

BOOTLOADER Firmware COMPONENTS

We can summarize the requirements for the firmware components of the bootloader as follows:

- Code resides at the RESET location
- Code is write protected against any accidental erasure or programming
- Code checks for the availability of new user code through some mechanism
- Code starts execution of existing user code, if no new user code is available for download
- Code receives new user code via some communication channel
- Code erases the memory device (FLASH only)
- Code programs the new user code into memory
- Code verifies the programming of user code

The firmware of the reference design bootloader is divided into two general parts: the **core bootloader firmware**, which initiates and manages operation, and the **programmer firmware**, which actually writes the new information to the memory devices. In this design, they are built from three distinct assembly files:

- **bloader.asm**, which handles bootloader invocation, operation and command decoding and execution
- **serial.asm**, which manages communications with the host software and protocol management
- **"xxx.asm"** (a user assigned name), which manages the memory write and erase processes, and contains the memory specific algorithms
The flow chart in Figure 3 shows the relationship between the firmware components and their assembly file sources.

FIGURE 3: OVERVIEW OF THE BOOTLOADER FIRMWARE

Invoking the Bootloader

There are many ways to indicate whether new user code should be downloaded. As examples, a designer could use:

- a jumper or switch on a port pin
- a particular command sequence on a communication channel
- the presence of a new device

The particular method chosen, depends on the way that user code is to be transferred into the microcontroller. For example, if the new user code is stored on an I²C™ EEPROM that is placed in a socket on a board, then an address in EEPROM could be read to determine whether a new EEPROM is present. Alternatively, the system can look for a bootloader command sequence coming from the serial port; if the command is not received in a specified period of time, the bootloader gives control to the existing user program. While this has the advantage of not using a hardware resource, it has a primary disadvantage that the device will experience a fixed delay every time it is RESET, before running the application.

The reference design uses a “hardware” invocation by monitoring one of the user defined port pins. Figure 4 shows how this is accomplished.

FIGURE 4: INVOKING THE BOOTLOADER
Core Bootloader Firmware

Once invoked, the core bootloader firmware starts execution. It waits for a valid command from the host. Upon receipt, it acknowledges the command back to the host. It then executes the command and sends a response to the host.

The main routine for the core bootloader is shown in the flow chart in Figure 5. The individual command handlers are detailed in Figures 6 through 11.

FIGURE 5: FLOW CHART OF MAIN PROGRAM LOOP FOR THE BOOTLOADER CORE
FIGURE 6: READ COMMAND HANDLER

A
- Initialize Data Counter and FSR at Starting Address of Data Buffer
- Read Data from Specified Address
- Store Data in Buffer
- Increment Address and Data Buffer Pointers

All data read?
- YES
  - Send all Read Data to Host
- NO

G

B
- Initialize Data Counter and FSR at Starting Address of Data Buffer
- Read Data from Data Buffer
- Generate Memory Location Address, Byte Data and Write Flag Information
- Transfer Control to Programmer Firmware in Boot RAM for Write Operation

Write successful?
- YES
  - Send Write Success Code to Host
- NO
  - All data written?
    - YES
      - Send Write Success Code to Host
    - NO
      - Send Write Error Code to Host

G

FIGURE 7: ERASE COMMAND HANDLER

C
- Transfer Control to Programmer Firmware in Boot RAM for Chip Erase

Chip Erase successful?
- NO
  - Send Erase Error Code to Host
- YES
  - Send Erase Success Code to Host

G

FIGURE 8: WRITE COMMAND HANDLER
Configure Boot RAM as General Purpose RAM

Initialize Data Counters at Start of Data Buffer and Boot RAM Address

Read Data from Specified Memory Address

Increment Boot RAM and Data Buffer Pointers

All data read?

YES

Configure Boot RAM as Program Memory

Send all Read Data to Host

NO

FIGURE 9: BOOT RAM READ COMMAND HANDLER

Configure Boot RAM as General Purpose RAM

Initialize Data Counters at Start of Data Buffer and Boot RAM Address

Fill all Locations with FFh Data

All data filled?

YES

Configure Boot RAM as Program Memory

Send Boot RAM Erase Success Code to Host

NO

FIGURE 10: BOOT RAM ERASE COMMAND HANDLER

Configure Boot RAM as General Purpose RAM

Initialize Data Counters at Start of Data Buffer and Boot RAM Address

Write Data to Specified Memory Address

Increment Boot RAM and Data Buffer Pointers

All data written?

YES

Configure Boot RAM as Program Memory

Send Boot RAM Write Success Code to Host

NO

FIGURE 11: BOOT RAM WRITE COMMAND HANDLER
Host Software Communications

The file 'Serial.asm' stores the serial interface code for a particular protocol. The 'Serial.inc' file contains definition of shared parameters for using this file. This file must be included in the "asm" file, where these serial routines are used.

The ParseHostCommand function waits for a valid command from host, and stays in the loop until a valid packet is received. It parses valid commands on receipt, and ignores all invalid packets. A flow chart of this function is shown in Figure 12.

The Send Host Data functions send data to host in defined packet, while Acknowledge Host Function acknowledges the host for command reception. A flow chart of the SendHostData is shown in Figure 13 (page 11).

FIGURE 12: FLOW CHART FOR THE ParseHostCommand ROUTINE

Start

(From Core Bootloader)

Wait for data; STX?

YES

H

Wait for data; STX?

YES

Wait for Data; Save Packet Length

H

Wait for Data; Save Command

Requires DLE de-stuffing

Wait for Data; Save Data Byte and Increment Data Buffer Pointer

All data received?

YES

Wait for Data; Save Checksum

H

Checksum verified?

YES

Wait for Data; DLE?

YES

Return to Core Bootloader

DLE De-stuffing Routine

Is data = DLE?

YES

Return and Process Data Byte

H

Wait for data; ETX?

YES

NO

Wait for data; DLE?

YES

NO

Wait for Data; STX?

NO

Wait for data; DLE?

YES

NO

Wait for data; STX?

YES

NO
FIGURE 13: FLOW CHART FOR THE SendHostData ROUTINE

Start (from Core Bootloader)

Send STX - DLE - STX Sequence

Send Packet Length; Initialize Checksum Value to Length

Send Command; Find new Checksum

Send New Data; Increment Data Pointer, find New Checksum

All data sent? NO

Send Checksum

Send DLE-ETX Sequence

Return to Core Bootloader

Requires DLE stuffing

DLE Stuffing Routine

Is data = DLE? NO

Send Extra DLE Byte

YES

Return and Send Data Byte
FIRMWARE/SOFTWARE INTERFACE

The data received by the core boot firmware will usually contain more than just program memory data. It will normally also contain the address to which data is to be written, the number of bytes transmitted and a checksum to detect errors. The firmware must decode, verify and store the data, before writing it into program memory. If the data is not verified, it should again ask source to retransmit it.

Because the available data RAM on-chip is limited in comparison to the maximum possible program size (2 MByte for the PIC18C801), the data to be programmed must be divided in small blocks. The bootloader must be able to control the reception of blocks, since it cannot process any data sent to it while it is writing to its own memory. As data is transferred in blocks, an error correction mechanism to take care of transmission errors becomes a requirement.

To identify transmission errors, a data communication protocol is required. The protocol in the reference design uses three instructions for the interface:

• Command, for instructions from host software to the firmware
• Acknowledge, as a “return receipt” by the firmware, for an instruction from the host software
• Response, containing the results of an instruction after decoding and execution by the firmware

**Command Format:**

<STX><DLE><Len><Command>[<Data>…]  
<Checksum><DLE><ETX>

where

<STX> is the “Start of TeXt” byte, used to synchronize the start of a packet (literal value of 02h)
<DLE> is the Data Link Escape byte, used to delimit the frame header or footer (literal value of 04h)
<Len> is the number of data bytes in the packet
<Command> is the encoded command byte
<Data> represents the parameter byte(s) for the command, with a length of <Len> bytes
<Checksum> is the 8-bit 2’s complement of sum of <Len>, <Command> and <Data>
<ETX> is the “End of TeXt” byte, used to mark the end of the packet (literal value of 03h)

If the <Len>, <Command>, <Data> or <Checksum> portion of the packet resembles DLE (i.e., has a value of 04h), an extra DLE will be stuffed before that byte. The stuffed DLE(s) will not change <Len> or <Checksum> value.

The receiver of the packet verifies the integrity of the data by adding the <Len>, <Command>, <Data> and <Checksum> bytes, excluding any stuffed DLEs. This sum must be 00h in order to confirm the integrity of received packet.

**Acknowledge Format:**

<STX><DLE><Len><ACK>[<Command>]  
<Checksum><DLE><ETX>

where

<STX> is the “Start of TeXt” byte, used to synchronize the start of a packet (literal value of 02h)
<DLE> is the Data Link Escape byte, used to delimit the frame header or footer (literal value of 04h)
<ACK> is the Acknowledge byte (literal value of 06h)
<Command> is the encoded command byte
<Len> is a single byte of literal value 01h
<Checksum> is the 8-bit 2’s complement of sum of <Len>, <Command> and <Data>
<ETX> is the “End of TeXt” byte, used to mark the end of the packet (literal value of 03h)

If the <Len>, <Command>, <Data> or <Checksum> portion of the packet resembles DLE (i.e., has a value of 04h), an extra DLE will be stuffed before that byte. The stuffed DLE(s) will not change <Len> or <Checksum> value.

The receiver of the packet verifies the integrity of the data by adding the <Len>, <Command>, <Data> and <Checksum> bytes, excluding any stuffed DLEs. This sum must be 00h in order to confirm the integrity of received packet.

**Response Format:**

<STX><DLE><Len><Result>[<Data>…]  
<Checksum><DLE><ETX>

where

<STX> is the “Start of TeXt” byte, used to synchronize the start of a packet (literal value of 02h)
<DLE> is the Data Link Escape byte, used to delimit the frame header or footer (literal value of 04h)
<Result> is the encoded binary result byte
<Data> represents the parameter byte(s) for the result, with a length of <Len> bytes
<Checksum> is the 8-bit 2’s complement of sum of <Len>, <Command> and <Data>
<ETX> is the “End of TeXt” byte, used to mark the end of the packet (literal value of 03h)

If the <Len>, <Command>, <Data> or <Checksum> portion of the packet resembles DLE (i.e., has a value of 04h), an extra DLE will be stuffed before that byte. The stuffed DLEs will not change <Len> or <Checksum> value.

The receiver of the packet verifies the integrity of the data by adding the <Len>, <Command>, <Data> and <Checksum> bytes, excluding any stuffed DLEs. This sum must be 00h in order to confirm the integrity of received packet.
Table 1 lists the preliminary commands included in the reference design, as well as their parameters. Additional commands can be added if and when required.

**TABLE 1: BOOTLOADER FIRMWARE COMMAND SET**

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Parameters</th>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD_VER</td>
<td>00h</td>
<td>Len = 0</td>
<td>Len = 1</td>
<td>Returns firmware version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = RD_VER</td>
<td>Result = RD_VER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = Version</td>
<td></td>
</tr>
<tr>
<td>RD_MEM</td>
<td>01h</td>
<td>Len = 5</td>
<td>Len = 5 + Len</td>
<td>Returns memory content from given address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = RD_MEM</td>
<td>Result = RD_MEM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = AddrLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[1] = AddrLH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[2] = AddrUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[3] = AddrUH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[4] = Len</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[5…5+Len] = Memory Data</td>
<td></td>
</tr>
<tr>
<td>WR_MEM</td>
<td>02h</td>
<td>Len = 6 + Len</td>
<td>Len = 1</td>
<td>Writes given memory contents to given address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = WR_MEM</td>
<td>Result = WR_MEM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = Number of bytes written</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[1] = AddrLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[2] = AddrLH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[3] = AddrUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[4] = AddrUH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[5] = Flag</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[6..6+Len] = Data</td>
<td></td>
</tr>
<tr>
<td>WR_CLR</td>
<td>03h</td>
<td>Len = 0</td>
<td>Result = WR_CLR</td>
<td>Erases memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = WR_CLR</td>
<td>Len = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = Result Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>’0’ = Success</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>’1’ = 0 Error Code</td>
<td></td>
</tr>
<tr>
<td>RD_MEM_BOOT</td>
<td>0Bh</td>
<td>Len = 5</td>
<td>Len = 5 + Len</td>
<td>Returns memory content from boot memory at given address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = RD_MEM</td>
<td>Result = RD_MEM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = AddrLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[1] = AddrLH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[2] = AddrUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[3] = AddrUH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[4] = Len</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[5…5+Len] = Memory Data</td>
<td></td>
</tr>
<tr>
<td>WR_MEM_BOOT</td>
<td>0Ch</td>
<td>Len = 6 + Len</td>
<td>Len = 1</td>
<td>Writes boot memory contents to given address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = WR_MEM</td>
<td>Result = WR_MEM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = Number of bytes written</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[1] = AddrLL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[2] = AddrLH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[3] = AddrUL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[4] = AddrUH</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[5] = Flag</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[6..6+Len] = Data</td>
<td></td>
</tr>
<tr>
<td>WR_CLR_BOOT</td>
<td>0Dh</td>
<td>Len = 0</td>
<td>Result = WR_CLR</td>
<td>Erases boot memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command = WR_CLR</td>
<td>Len = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data[0] = Result Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>’0’ = Success</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>’1’ = 0 Error Code</td>
<td></td>
</tr>
</tbody>
</table>

© 2001 Microchip Technology Inc.
Programming Firmware

The file ‘xxx.asm’ stores the code for FLASH programming. This file is memory specific, so the user may need to change it depending on their specific requirement. The default FLASH programmer can be attached with the bootloader; any FLASH programmer can be downloaded on demand at a later time.

The actual implementation will vary, depending on the memory device and interface mode used. Broadly, FLASH devices can be divided into four families, depending on their programming algorithm. In addition to programming algorithm, implementation will change based on external interface. The general programmer algorithm is described in Figure 14. Examples of specific algorithms for different FLASH families are outlined in Figures 15 through 18.

It is important to note that these flow charts do not include all programming algorithms for all FLASH device families available on the market. Additional information on FLASH families and programming commands is provided in Appendixes C and D.

APIs FOR EXTERNAL MEMORY PROGRAMMING AND ERASE FUNCTIONS

To provide a simple method for interfacing user designed FLASH programming algorithms to the rest of the code, Application Program Interfaces (or APIs) have been designed for FLASH Erase and FLASH Write routines. These APIs also allow the core bootloader and programmer firmware to share information, as described later. The interfaces are described below.

Erase Function

Purpose: Erase all available memory locations.
Prototype: WREG Erase()
Input: None
Output:
WREG: Result code of this function
If WREG == 00h
Function was successful
Else
There was an error, which may be explained by the non-zero value

Write Function

Purpose: Write an 8-bit value to a memory location defined by 32-bit value.
Prototype: WREG Write (DWORD Address, BYTE Data, BYTE Flag)
Input:
Address: 32-bit address of the location being written
Data: 8-bit data value to be written to given address
Flag: Specifies whether this is a first, intermediate, last or only byte of total data to be written. The following table describes the valid values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>This is the first byte being written. User may setup “Write” mode for external memory in beginning of this function.</td>
</tr>
<tr>
<td>01h</td>
<td>This is a last byte being written. User must change external memory mode to “Read Array”.</td>
</tr>
<tr>
<td>02h</td>
<td>This is the only byte being written. User may set up “Write” mode for external memory in beginning of this function and must change it to “Read Array” mode before returning from this function.</td>
</tr>
<tr>
<td>All other values</td>
<td>This is an intermediate byte being written. User may not need to change external memory mode during this call.</td>
</tr>
</tbody>
</table>

Output:
WREG: Result code for this function
If WREG == 00h,
Function was successful
Else
There was an error, which may be explained by the non-zero value
FIGURE 14: FLOW CHART FOR THE GENERAL PROGRAMMER FIRMWARE

1. Start Programmer

ERASE

2. Branch to Command Executor Based on Jump Table

3. Initialize Device (Set for TBLWT mode or Disable External Bus and Enable I/O Ports)

BYTE WRITE

4. Call FLASH Write Routine for Byte Write at Address

5. Even address?
   - YES: Make Word from this Byte and Previously Stored Byte
   - NO: Call Chip Erase Function

WRITE

6. Call FLASH Write Routine for Word Write to Address

7. Store Byte until Odd Byte (MSByte) is Received

8. Save pending data in buffer

SECTOR WRITE

9. First byte?
   - YES: Initialize Data Pointer Buffer
   - NO: Does supplied address match with expected next address?
     - YES: Copy FFh into Buffer
     - NO: Store Address for this Data

10. Copy Data into Buffer

11. Increment Data Buffer Pointer

12. Flag Address Mismatch

13. All sector data received?
   - YES: Call FLASH Write Routine to Program Sector
   - NO: Copy FFh into Buffer

14. Last byte?
   - YES: Increment Data Buffer Pointer
   - NO: Call FLASH Write Routine to Program Sector

15. Did address mismatch occur?
   - YES: Restore Device Settings for External Execution mode
   - NO: Store Write/Erase Success/Error Code in W Register

Return to Command Handlers
**FIGURE 15:** COMMON “WAIT FOR END-OF-WRITE” ROUTINES FOR FLASH DEVICES

**FIGURE 16:** TYPICAL WRITE CYCLE ROUTINE FOR SECTOR-PROGRAM FLASH

---

**Internal Timer Routine**
- **Start**
- Wait for T WC
- **Return**

**Toggle Bit Routine**
- **Start**
- Read Byte from Page
- Read Same Byte
- Do
  - **NO**
  - **Return**
- **YES**
- **Return**

**Data # Polling Routine**
- **Start**
- Read DQ7 (Data for Last Byte Loaded)
- Is DQ7 = true data?
  - **NO**
  - **Return**
  - **YES**

**Start**
- (Call from Programmer)
- Load Data AAh to Address 5555h
- Unlock Sequence
- Load Data 55h to Address 2AAAAh
- Load Data 0Ah to Address 5555h (Write Command)
- Set Page Address
- Set Address to Beginning of Sector
- Load Data
- Increment Address Counter by 1
- All sector data written?
  - **NO**
  - **Wait for End of Write (Memory Specific)(1)**
  - **YES**
  - Return to Programmer

**Note 1:** See Figure 15 for common examples.
FIGURE 17: TYPICAL WRITE AND ERASE SEQUENCES FOR “A” AND “B” FLASH FAMILIES

Note 1: For FLASH family A:
Unlock address 1 — 555h
Unlock address 2 — 2AAh
Command Initiate — 555h
For FLASH family B:
Unlock address 1 — 5555h
Unlock address 2 — 2AAAh
Command Initiate — 5555h

2: See Figure 15 for common examples.
FIGURE 18: TYPICAL WRITE AND ERASE ROUTINES FOR “C” FAMILY FLASH DEVICES

Byte/Word Write

Start (Call from Programmer)

Write 70h

Read Status Register

SR<7> = 1? NO

YES

Write 40h or 10h (Location to be Written)

Write Word/Byte and Supplied Address

Read Status Register

SR<7> = 1? NO

YES

Full Status Check (if desired)

Return to Programmer

Chip Erase

Start (Call from Programmer)

Initialize Block Address (next Block after Bootloader)

Write 70h

Read Status Register

SR<7> = 1? NO

YES

Write 20h to any Address of Block to be Erased

Write D0h to any Address of Block to be Erased

Read Status Register

SR<7> = 1? NO

YES

Full Status Check (if desired)

Last Block?

NO

YES

Return to Programmer

Increment to Next Block
PARAMETER PASSING MECHANISM FOR ASSEMBLY LANGUAGE

Normally, the default flash programmer is attached with the core bootloader; an alternate programmer can be downloaded from the host software later, if required. This ability to change programmer firmware is why the the core bootloader and programmer are built as separate projects. In doing this, however, it becomes necessary to provide a mechanism for sharing data and functions between the two. It is also essential to prevent the firmware components from using overlapping areas of RAM. This can best be done by using an absolute addressing scheme.

To enforce reasonable type checking, the generic code portion will define and export certain variables. These are listed in Example 1.

Generic code will populate these variables before calling user supplied Write function. The user supplied Write function will import these variables and use them as needed.

WRITING NEW FLASH MEMORY ROUTINES FOR THE BOOTLOADER

User supplied Write functions can use the provided 'memrtnes.inc' file, which contains the definition of these parameters, as shown in Example 2. This way, the core bootloader and programmer firmware can share the data.

If memory routines are built separately from the bootloader, always use the "memrtnes.lkr" file (included in the Zip archive available at the Microchip website) to build them. This makes sure that the FLASH routines do not overlap with monitor data RAM area.

Now that a system for sharing the data is established, we need a mechanism to share the functions as well. One solution is to fix the location for Write and Erase functions themselves. This may create a problem if all the required firmware does not fit in the allotted space. We need some mechanism, so a user can place their firmware at anywhere in available area. FLASH programmer specific code contains a "jump table" at the beginning of code, which is what the bootloader uses to call appropriate routines. This jump table allows user to locate their actual functions anywhere in the 512-byte area; they do not have to "origin" their functions at hard coded addresses. When FLASH routines are downloaded by the host software, it "relocates" them at beginning of Boot RAM; for this reason, users must only use bra and rcall instructions for jumps. Example 3 shows how this is done.

The best way to embed the memory routine code is to use the template file "memrtnes.tpl", which takes care of all the definitions. The template is also included in the Zip archive available at the Microchip website.

Note: For more information on using the templates, please refer to the User’s Manual and on-line help for the MPLAB® development system.

EXAMPLE 1: DEFINING COMMON VARIABLES FOR PASSING PARAMETERS

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDATA_ACS</td>
<td>.00</td>
</tr>
<tr>
<td>Address</td>
<td>RES .04    ; Parameter #1 for Write function</td>
</tr>
<tr>
<td>Byte</td>
<td>RES .01    ; Parameter #2 for Write function</td>
</tr>
<tr>
<td>Flags</td>
<td>RES .01    ; Parameter #3 for Write function</td>
</tr>
</tbody>
</table>

EXAMPLE 2: EXAMPLE CODE FOR memrtnes.inc

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>EQU .00    ; 32 bit Address of the location being written</td>
</tr>
<tr>
<td>Byte</td>
<td>EQU .04    ; 8-bit data value to be written</td>
</tr>
<tr>
<td>Flags</td>
<td>EQU .05    ; Specifies whether this is a first, intermediate, last or only one byte of total data to be written</td>
</tr>
</tbody>
</table>
EXAMPLE 3: PROVIDING FOR RELOCATABLE WRITE AND ERASE FUNCTIONS

Bootloader Code section for Calling Write and Erase APIs

```assembly
call 1FFE00h + @Command ;Write Command =0, Erase Command=2

In this instance, the FLASH Write function called is located at 1FFE00h, while the Erase function is located at 1FFE02h. The command handler must be located at these locations. This is done as follows:

Programmer CODE ;This section when copied to Boot RAM makes address 1FFE00
bra Write ;Branch to Write function
bra Erase ;Branch to Erase function

Write:
 ;(Insert Write routine here)
  return

Erase:
 ;(Insert Erase routine here)
  return
```

INTEGRATING THE BOOTLOADER WITH USER CODE

The bootloader code usually uses the RESET location and some additional program memory. It can also use the interrupt; but, if an interrupt occurs while the code is executing from Boot RAM, it will jump to the interrupt service vector in FLASH program memory. This could be dangerous if programming the new code into external memory has not been completed. Thus, the onboard programmer must not use interrupt driven code. It should disable interrupts until it finishes programming external memory.

The bootloader starts at the RESET location. To avoid accidental erasure, this entire sector of program memory must be protected. As the interrupt vector also falls in this range, the bootloader must relocate it. Additionally, the bootloader must know where the application code starts, to be able to execute it. Similarly, users may want to change other bootloader related configuration items for different systems, such as the pin monitored to invoke the firmware, or the oscillator frequency used to calculate the baud rate for serial communications.

All of these user code related parameters are defined in the 'UserCode.inc' file. Users can edit this file to quickly modify the firmware to suit their particular requirements. An example is shown in Example 4 (page 21). Users should store their code in the next sector after the Bootloader code. This location address is defined in the UserCode label. In the same fashion, interrupt vector relocation addresses are defined at 'HighPriorIntServ' and 'LowPriorIntServ', for high priority and low priority Interrupt Service Routines, respectively. The pin monitored for bootloader invocation is defined by BootLoadChkPin. Both the Port name and bit number should be defined here.

The bootloader code coexists with the user code on the device and many of the resources used by the boot code can also be used by the user code. The core bootloader and programmer firmware uses the resources listed in Table 2.

### TABLE 2: RESOURCES NEEDED FOR THE BOOTLOADER

<table>
<thead>
<tr>
<th>Resource</th>
<th>Bootloader Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Core Boot Firmware</td>
</tr>
<tr>
<td>Program memory (bytes)</td>
<td>1024</td>
</tr>
<tr>
<td>Data memory (access RAM, bytes)</td>
<td>32</td>
</tr>
<tr>
<td>Data memory (general purpose RAM, bytes)</td>
<td>255</td>
</tr>
<tr>
<td>I/O pins</td>
<td>1</td>
</tr>
<tr>
<td>Peripherals</td>
<td>USART</td>
</tr>
</tbody>
</table>

* Requirements vary by specific implementation and FLASH programming algorithm.
The program memory used by the bootloader cannot be used for user code. However, actual memory consumption will depend on the sector size, as the sector containing bootloader code must be protected (and therefore, cannot contain user code). Larger sector sizes mean greater memory consumption; smaller sectors mean lesser consumption. As the bootloader firmware can consume significant data memory resources, it is not likely that developers will want to reserve these on an ongoing basis for code that is infrequently called. As all the code is written in relocatable format, MPLINK™ Object Linker will not allow the re-use of resources used by the bootloader code, if application code is merged with bootloader code to make a single project. Therefore, combining user code and bootloader code into a single project should be avoided.

In a production environment, however, it is desirable to program the entire FLASH device with the bootloader firmware and user application code in a single shot. In this case, the developer should build two separate HEX files (bootloader and user code), then merge the two to create a single HEX file. This allows the developer to re-use the data memory resources used by the bootloader.

To avoid overlap of program memory, the developer should use the appropriate linker script file. User code should use a linker script file similar to the one shown in Figure 5. This will prevent overlap of user code with Bootloader. The text in **bold** defines the sector requirement. The text in **bold italics** shows the maximum available program memory with device; this is modified according to the physical memory connected to the device. If more than one memory device is connected, this file should reflect memory map of the system. The use of a proper linker script file will ensure that the linker places code and variables in the proper places.

Note: For additional information on linker scripts, please refer to the Microchip MPLINK User’s guide.

The USART can be used by the user code. Any I/O pin(s) monitored to invoke the bootloader can be used as an output, by isolating their switches or jumpers with a resistor.

In summary, all resources used by the bootloader, except program memory, can also be used by the user application code. Figure 19 shows the final combined memory map of user code and bootloader firmware.

### EXAMPLE 5: SAMPLE LINKER SCRIPT FILE

```plaintext
// File: UserCode.lkr
// Sample linker command file for User code

LIBPATH.

CODEPAGE NAME=vectors START=0x0 END=0x29 PROTECTED
CODEPAGE NAME=Bootloader START=0x3A END=0xFF PROTECTED
CODEPAGE NAME=page START=0x1000 END=0x200000 PROTECTED
CODEPAGE NAME=config START=0x300000 END=0x300007 PROTECTED
CODEPAGE NAME=idlocs START=0x3FFFFE END=0x3FFFFF PROTECTED

ACCESSBANK NAME=accessram START=0x0 END=0x7F
DATABASE NAME=gpr0 START=0x80 END=0x8FF
DATABASE NAME=gpr1 START=0x100 END=0x1FF
DATABASE NAME=gpr2 START=0x200 END=0x2FF
DATABASE NAME=gpr3 START=0x300 END=0x3FF
DATABASE NAME=gpr4 START=0x400 END=0x4FF
DATABASE NAME=gpr5 START=0x500 END=0x5FF
DATABASE NAME=sfr START=0xF00 END=0xF7F PROTECTED
ACCESSBANK NAME=accesssfr START=0xF80 END=0xFFF PROTECTED
```

Note: Users can merge HEX files by using the facilities available in some programmers.
CONCLUSION

Incorporating bootloader firmware into a microcontroller based design allows for easy and efficient field upgrades of a product, which in turn, can enhance its functionality and value. Designs using the PIC18C601/801 ROMless microcontrollers can easily incorporate a bootloader to enhance their flexibility.

The reference design demonstrated in this note provides a flexible and modular framework for bootloader firmware. To recap, some of the features included are:

- External host software with a simple GUI and the flexibility to handle the most popular HEX file formats
- A serial communications interface with a robust data communication protocol, making it possible to identify and correct communication errors
- Downloadable programmer firmware, which allows for the development and substitution of new FLASH programming algorithms
- Prewritten linker scripts, templates, and “include” files for the efficient development of new memory routines, the ability to share device resources and overlap multiple code pieces, and the ability to customize the firmware to user requirements
- The ability to map the external bus to I/O ports, to allow the implementation of any memory programming algorithm

Using the key components of the reference design will allow developers to create their own custom bootloader firmware, specifically tailored to their application’s resources and requirements.
APPENDIX A: REFERENCES

Readers with additional questions on Microchip ROM-less microcontrollers, the external memory interface and FLASH memory programming, are referred to the documents listed below for more information. They may be downloaded from the Microchip corporate website, at

www.microchip.com

• DS39541, “PIC18C601/801 Data Sheet”
• DS00778, “Implementing the External Memory Interface on PIC18C601/801 MCUs”

APPENDIX B: SOFTWARE DISCUSSED IN THIS APPLICATION NOTE

Because of the overall length of all components, a complete source file listing for the bootloader reference design is not provided. Those users who are interested in further exploring the bootloader firmware are encouraged to download the project files for their examination.

The software discussed in this application note (the Host Software executable file and project files and templates for the bootloader firmware) are available as a single WinZip archive file. The archive may be downloaded from the Microchip corporate Web site at:

www.microchip.com
# APPENDIX C: SUMMARY OF MEMORY DEVICES

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Part ID</th>
<th>Programming Algorithm Family&lt;sup&gt;(2)&lt;/sup&gt;</th>
<th>Organization</th>
<th>Basic Byte/Word Addressing&lt;sup&gt;(4)&lt;/sup&gt;</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>29F series</td>
<td>A</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29F series</td>
<td>A</td>
<td>x16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29F series</td>
<td>A</td>
<td>x8/x16</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>ATMEL</td>
<td>29 Series</td>
<td>B&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>x8</td>
<td></td>
<td>Sector Programming</td>
</tr>
<tr>
<td></td>
<td>29 Series</td>
<td>B&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>x16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 Series</td>
<td>B</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 Series</td>
<td>B</td>
<td>x16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>49 Series</td>
<td>B</td>
<td>x8/x16</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>INTEL</td>
<td>Boot Block</td>
<td>C</td>
<td>x8/x16</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strata FLASH/FLASH File</td>
<td>C</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strata FLASH/FLASH File</td>
<td>C</td>
<td>x8/x16</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>SHARP</td>
<td>28F series</td>
<td>C</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>28F series</td>
<td>C</td>
<td>x16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>28F series</td>
<td>C</td>
<td>x8/x16</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td>29F series</td>
<td>A</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29F series</td>
<td>A</td>
<td>x16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29F series</td>
<td>A</td>
<td>x8/x16</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>Samsung</td>
<td>FLASH products in this family have multiplexed address/data/command lines, and are incompatible with PIC18C601/801 devices.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalyst</td>
<td>Boot Block FLASH</td>
<td>C</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bulk Erase FLASH</td>
<td>(5)</td>
<td>x8</td>
<td>x16</td>
<td></td>
</tr>
</tbody>
</table>

**Note** 1: This listing is provided only as an example of typical memory devices available. It is not meant to be exhaustive.

2: Details of each programming algorithm family are provided in Appendix B.

3: For these devices, users must provide all data in the sector. The device will first erase the entire sector, then program it. These devices do not support Sector Erase commands.

4: Applicable only to x8/x16 selectable devices.

5: These devices have a unique set of programming algorithms. They are omitted for the sake of brevity.
APPENDIX C: SUMMARY OF MEMORY DEVICES\(^{(1)}\) (CONTINUED)

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Part ID</th>
<th>Programming Algorithm Family(^{(2)})</th>
<th>Organization</th>
<th>Basic Byte/Word Addressing(^{(4)})</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyundai</td>
<td>29F series</td>
<td>A</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29F series</td>
<td>A</td>
<td>x8/x16</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>Micron</td>
<td>Boot Block</td>
<td>C</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Boot Block</td>
<td>C</td>
<td>x8/x16</td>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Even Sectored</td>
<td>C</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Even Sectored</td>
<td>C</td>
<td>x8/x16</td>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>SST</td>
<td>39F Series</td>
<td>B</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>29EE Series</td>
<td>B(^{(3)})</td>
<td>x8</td>
<td></td>
<td>Sector Programming</td>
</tr>
<tr>
<td>NexFlash</td>
<td>29F series</td>
<td>B</td>
<td>x8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note 1:** This listing is provided only as an example of typical memory devices available. It is not meant to be exhaustive.

**2:** Details of each programming algorithm family are provided in Appendix B.

**3:** For these devices, users must provide all data in the sector. The device will first erase the entire sector, then program it. These devices do not support Sector Erase commands.

**4:** Applicable only to x8/x16 selectable devices.

**5:** These devices have a unique set of programming algorithms. They are omitted for the sake of brevity.
APPENDIX D: PROGRAMMING ALGORITHMS FOR REPRESENTATIVE MEMORY DEVICES (1)

<table>
<thead>
<tr>
<th>Command</th>
<th>Program Algorithm</th>
<th>Cycles needed</th>
<th>Bus Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>First</td>
<td>Second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr</td>
<td>Data</td>
</tr>
<tr>
<td>Read mode/RESET</td>
<td>A</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>Read Mfg. ID</td>
<td>A</td>
<td>4</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>Read Device ID</td>
<td>A</td>
<td>4</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
<td>5555</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>Write</td>
<td>A</td>
<td>4</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
<td>5555</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>(WA)</td>
</tr>
<tr>
<td>Block Erase</td>
<td>A</td>
<td>6</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>6</td>
<td>5555</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>(BA)</td>
</tr>
<tr>
<td>Erase Suspend</td>
<td>A</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erase Resume</td>
<td>A</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>Chip Erase</td>
<td>A</td>
<td>6</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>6</td>
<td>5555</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>Sector Protect</td>
<td>A</td>
<td>4</td>
<td>555</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>4</td>
<td>5555</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend: WA = Write Address, WD = Write Data, IA = Identifier Address, ID = Identifier Data, BA = Block Address, SGA = Sector Group Address, X = Don't Care

Note 1: The information provided in this table is for reference only, and is not meant to be a comprehensive description of the device programming algorithms. For complete information, please refer to the manufacturer’s data sheet.

Note 2: Instruction unimplemented in this programming algorithm family.
APPENDIX E: THE HOST SOFTWARE WINDOW

As previously described, the host software for the reference design bootloader is implemented using a single window (Figure E-1). All commands are available from both the menu bar; most are also available from either the icon-based toolbar (Figure E-2), or keyboard shortcuts. File commands (New, Open, Save, etc.) invoke the standard Windows dialog boxes for file location, name and file format. A complete summary of all available commands is given in Table E-1.

Also available on the Tool Bar is the option to change the COM port setting used by the host system to communicate with the target. The selector is not duplicated as a command menu option. The default port is COM1.

It is important to note that the host software is not a HEX file editor; the display in the main window only shows the current HEX file or memory device contents loaded into the buffer. Developers who want to make changes to a programmed device will still need to follow the usual steps of the software development cycle, using the appropriate software tools for code design and compilation to a HEX file. Only then can the HEX file be loaded into the host software and reprogrammed into the device.
## TABLE E-1: SUMMARY OF HOST SOFTWARE COMMANDS

<table>
<thead>
<tr>
<th>Menu</th>
<th>Command</th>
<th>Keyboard Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File</strong></td>
<td>New</td>
<td>&lt;Ctrl-N&gt;</td>
<td>Clears the host software buffer and prepares for a new HEX file to be loaded.</td>
</tr>
<tr>
<td></td>
<td>Open</td>
<td>&lt;Ctrl-O&gt;</td>
<td>Opens an existing Intel HEX file residing on the host system and displays it in the main window. The software will prompt for the file name and its location.</td>
</tr>
<tr>
<td></td>
<td>Close</td>
<td></td>
<td>Closes the currently open HEX file and clears the host software buffer. If changes have occurred and have not been saved, the user will be asked if they wish to save the changes.</td>
</tr>
<tr>
<td></td>
<td>Save</td>
<td>&lt;Ctrl-S&gt;</td>
<td>Saves the currently displayed data to the open HEX file. If no HEX file is open, invokes the “Save As” function.</td>
</tr>
<tr>
<td></td>
<td>Save As</td>
<td>&lt;F12&gt;</td>
<td>Save the currently displayed data as an Intel HEX file. A dialog box will prompt for location and new file name.</td>
</tr>
<tr>
<td></td>
<td>Exit</td>
<td></td>
<td>Exit the host software without changing or saving the currently displayed data.</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>Program</td>
<td>&lt;Ctrl-P&gt;</td>
<td>Programs the contents of the host software buffer to the target memory device. For FLASH devices, this includes erasing the target, writing to the target, then verifying the data written.</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>&lt;Ctrl-W&gt;</td>
<td>Downloads the current contents of the host software buffer to the target device, without performing Erase or Verify operations.</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>&lt;Ctrl-R&gt;</td>
<td>Reads the code from the target memory device and displays it in the main window. The software will prompt for a range of addresses to be read.</td>
</tr>
<tr>
<td></td>
<td>Erase</td>
<td>&lt;Ctrl-E&gt;</td>
<td>Erases the target memory device.</td>
</tr>
<tr>
<td></td>
<td>Verify</td>
<td>&lt;Ctrl-V&gt;</td>
<td>Verifies the current contents of the memory device against the displayed file.</td>
</tr>
<tr>
<td></td>
<td>Download Memory Routines</td>
<td>&lt;Ctrl-M&gt;</td>
<td>Downloads the contents of the host software buffer to Boot RAM of the target controller, and verifies after download.</td>
</tr>
<tr>
<td></td>
<td>Read Memory Routines</td>
<td></td>
<td>Reads the current contents of Boot RAM from the target controller, and displays it in the main window.</td>
</tr>
<tr>
<td></td>
<td>Verify Memory Routines</td>
<td></td>
<td>Verifies the current contents of Boot RAM against the displayed file.</td>
</tr>
<tr>
<td></td>
<td>Abort</td>
<td>&lt;Ctrl-A&gt;</td>
<td>Terminates the current operation.</td>
</tr>
<tr>
<td></td>
<td>About</td>
<td></td>
<td>Displays the current revision of the host software.</td>
</tr>
</tbody>
</table>

**Note:** Interrupting a Program or Write operation with the **Abort** command can cause unpredictable memory states, which may result in erratic operation. This may require erasing and reprogramming the target memory device.
Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products 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 PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be 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 product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

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.

Trademarks

The Microchip name and logo, the Microchip logo, FilterLab, KEELoQ, MPLAB, PIC, PICmicro, PICMASTER, PICSTART, PROMATE, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

dsPIC, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microID, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, MXDEV, PICC, PICDEM, PICDEM.net, rPIC, Select Mode and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A.

Serialized Quick Term 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.

© 2001, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.