Implementing Bootloader Firmware for the PIC18C601/801 ROMless Microcontrollers

Authors: *Gaurang Kavaiya and
Nilesh Rajbharti
Microchip Technology Inc.*

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.

This application note discusses the general design requirements for bootloader firmware in a ROMless controller system. To illustrate the key points, a fully-featured reference design, with an interface to external host software, is described in detail. Information on integrating a bootloader with user application code is also covered.

The reader is expected to be familiar with the following:

- General PIC18 architecture
- The PIC18 instruction set
- External memory interface modes of the PIC18 ROMless devices, and
- Interface modes of different non-volatile memory devices

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

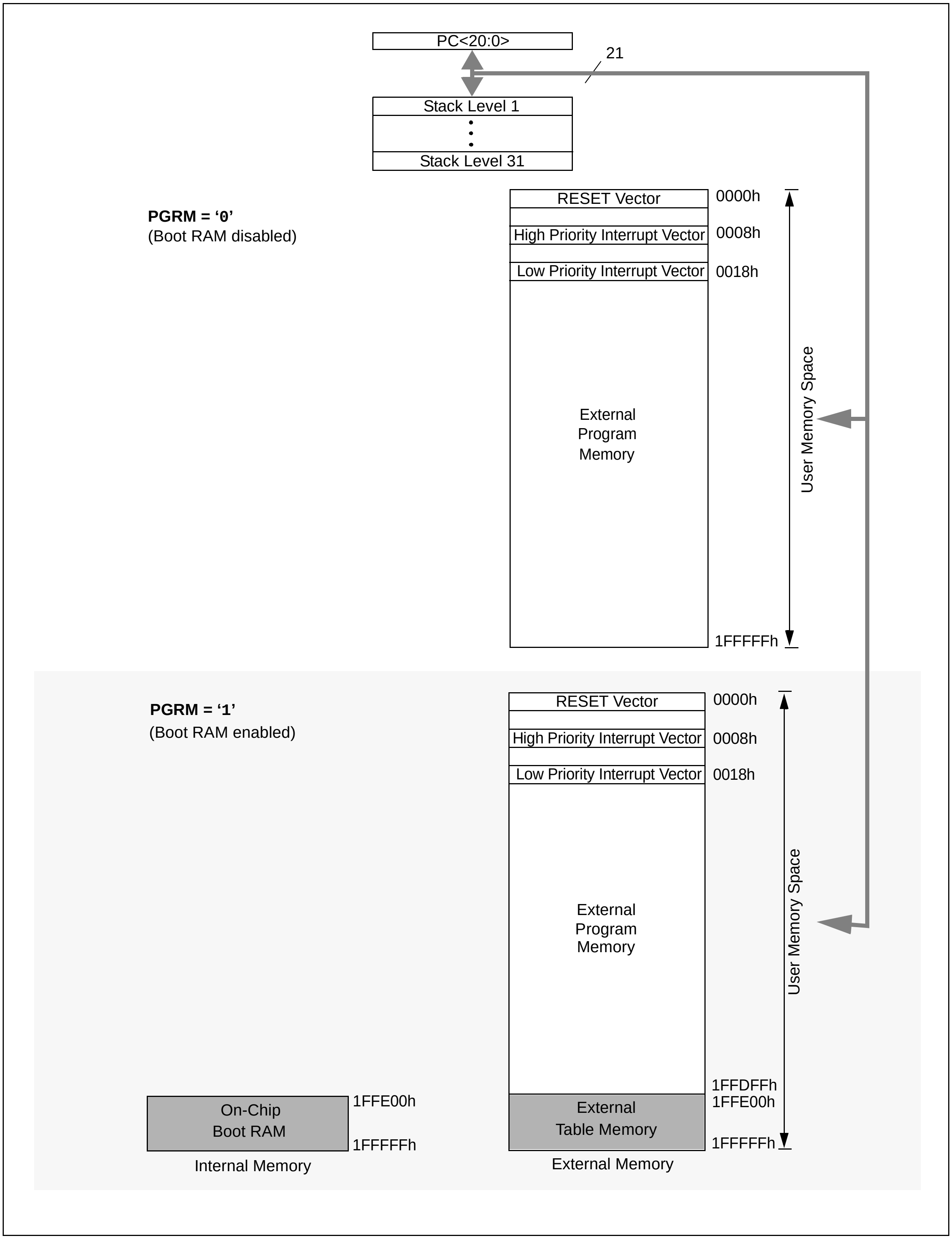
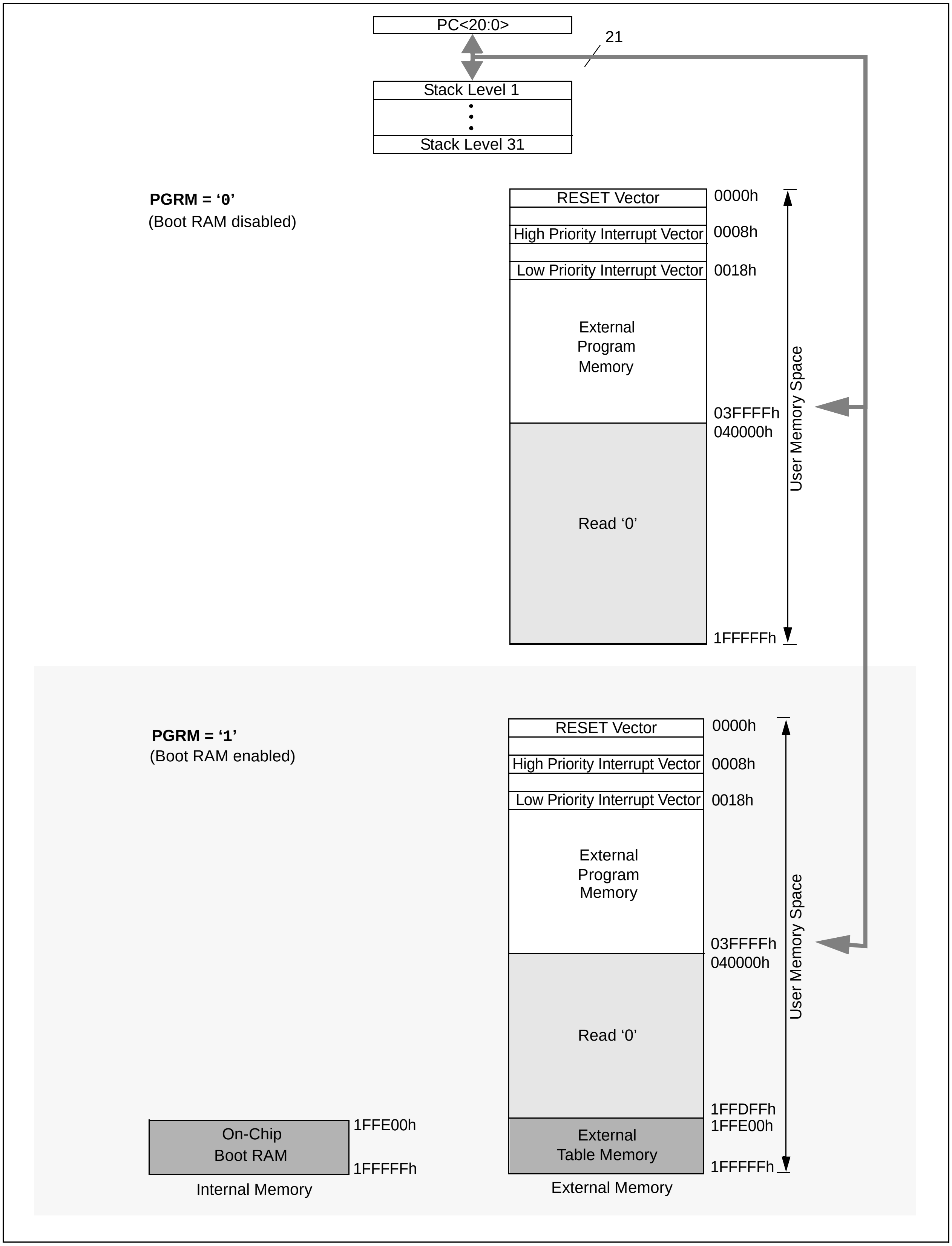


FIGURE 2: MEMORY MAP AND PROGRAM STACK FOR THE PIC18C601



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:

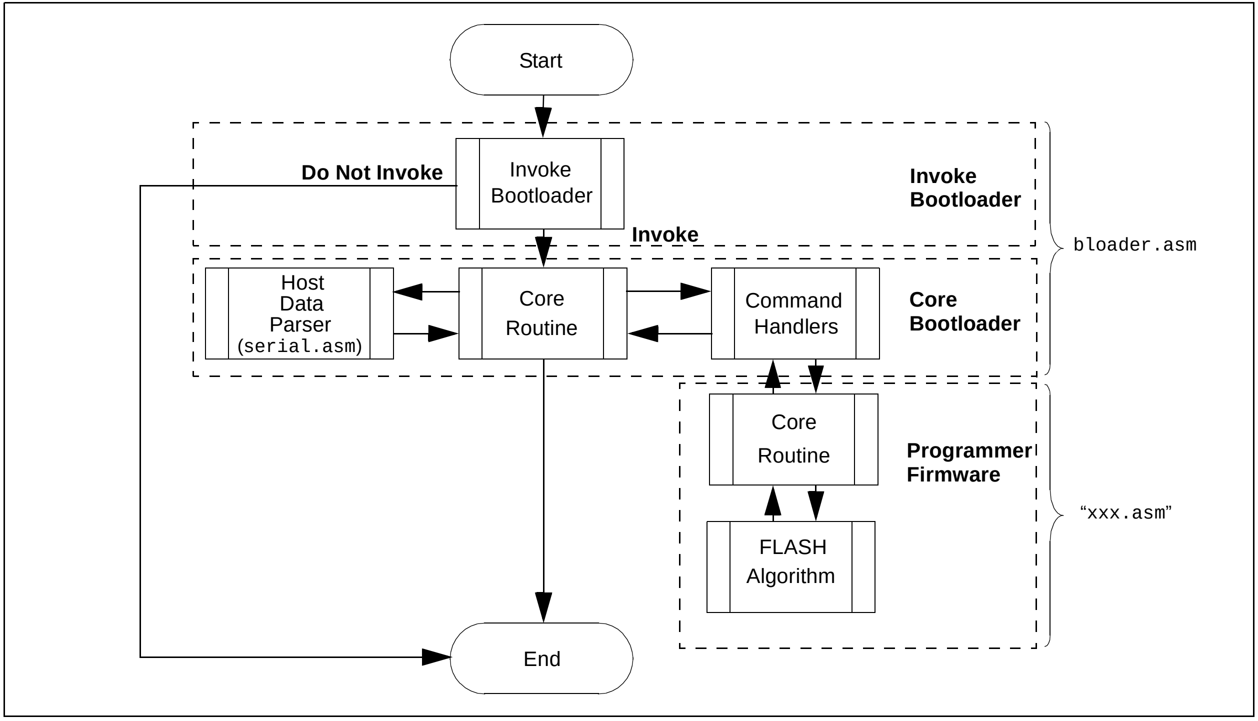
- Code resides at the RESET location
- Code is write protected against any accidental erasure or programming
- Code checks for the availability of new user code through some mechanism
- Code starts execution of existing user code, if no new user code is available for download
- Code receives new user 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

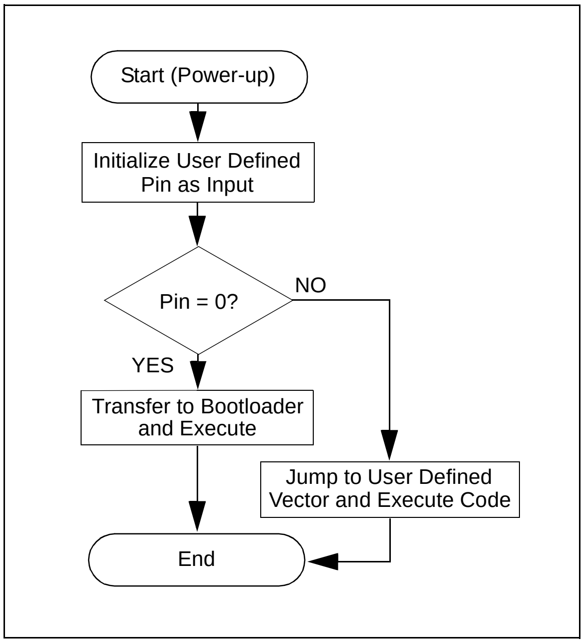
There are many ways to indicate whether new user code should be downloaded. As examples, a designer could use:

- 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 I^2C™ 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 boot loader 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


Start (Power-up)
Initialize User Defined Pin as Input
Pin = 0?
NO
YES
Transfer to Bootloader and Execute
Jump to User Defined Vector and Execute Code
End

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

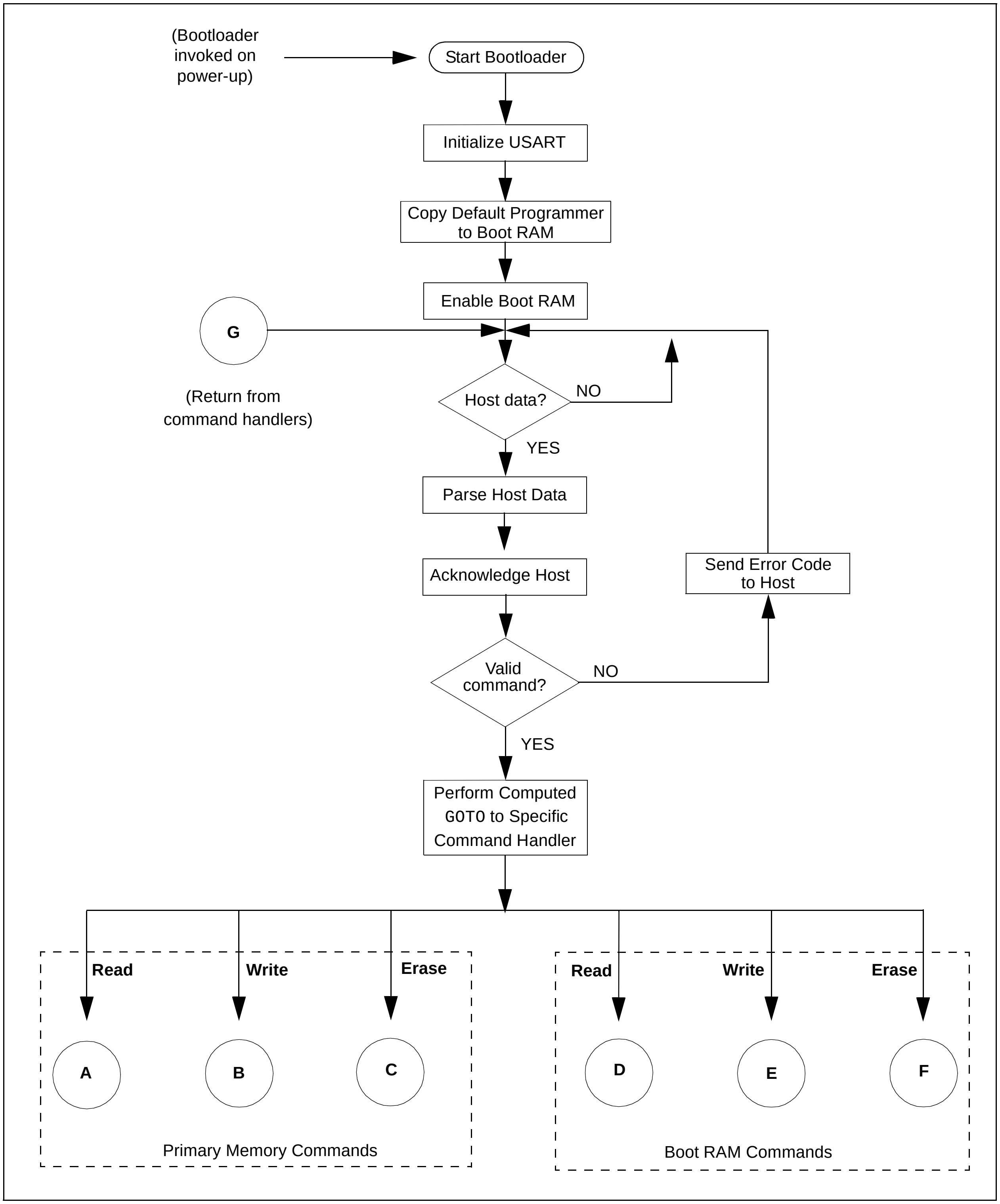


FIGURE 6: READ COMMAND HANDLER

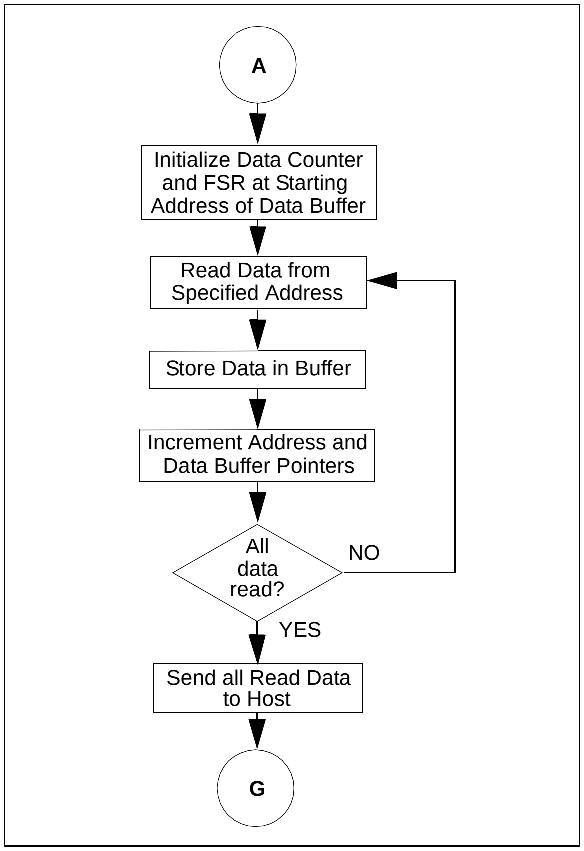

A
Initialize Data Counter and FSR at Starting Address of Data Buffer
Read Data from Specified Address
Store Data in Buffer
Increment Address and Data Buffer Pointers
All data read?
NO
YES
Send all Read Data to Host
G

FIGURE 7: ERASE COMMAND HANDLER

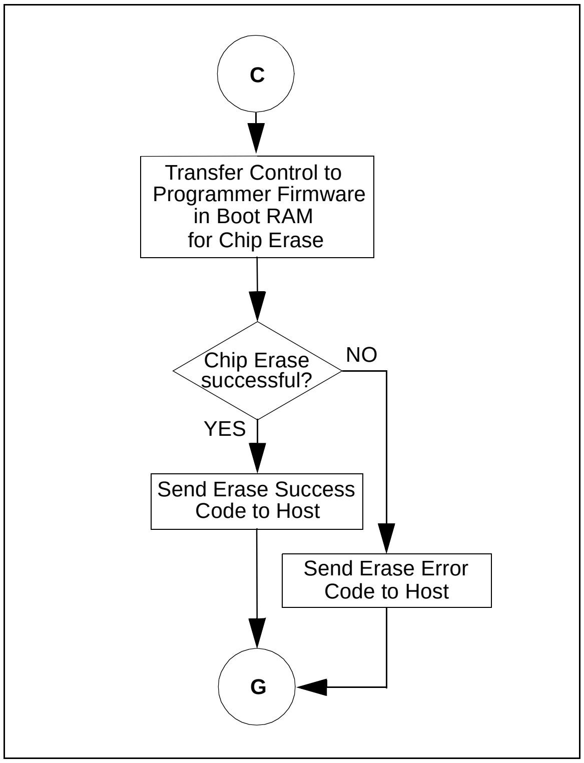

C
Transfer Control to Programmer Firmware in Boot RAM for Chip Erase
Chip Erase successful?
NO
YES
Send Erase Success Code to Host
Send Erase Error Code to Host
G

FIGURE 8: WRITE COMMAND HANDLER

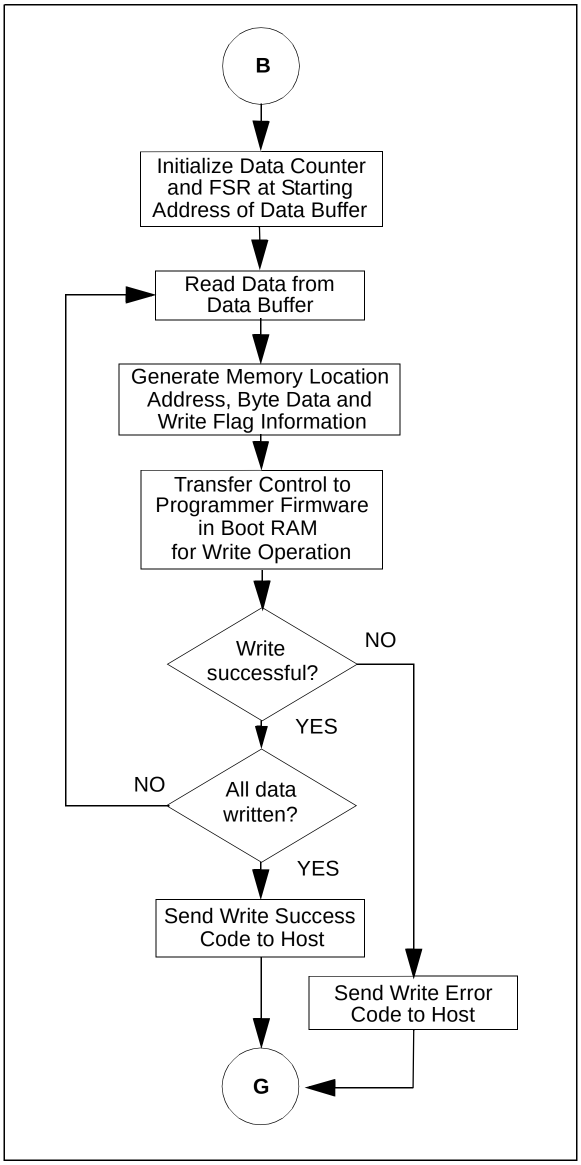

B
Initialize Data Counter and FSR at Starting Address of Data Buffer
Read Data from Data Buffer
Generate Memory Location Address, Byte Data and Write Flag Information
Transfer Control to Programmer Firmware in Boot RAM for Write Operation
Write successful?
NO
YES
NO
All data written?
YES
Send Write Success Code to Host
Send Write Error Code to Host
G

FIGURE 9: BOOT RAM READ COMMAND HANDLER

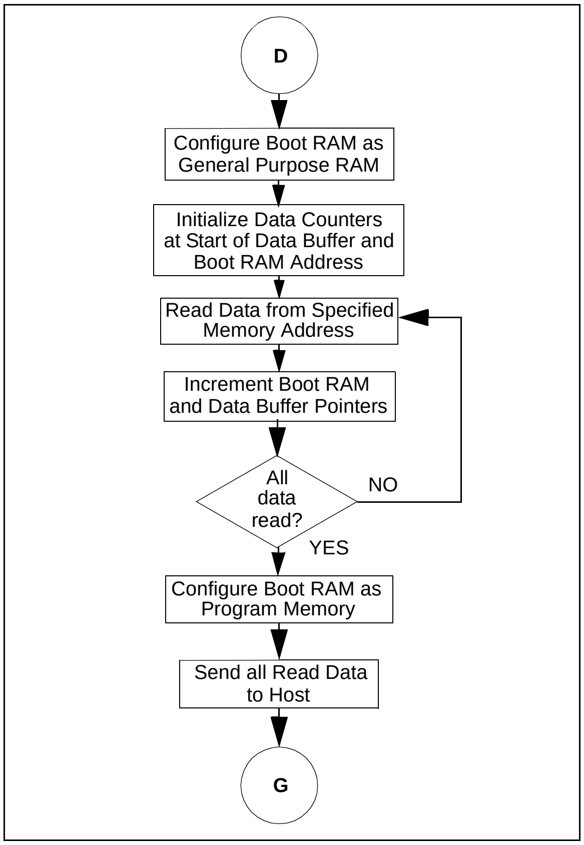

D
Configure Boot RAM as General Purpose RAM
Initialize Data Counters at Start of Data Buffer and Boot RAM Address
Read Data from Specified Memory Address
Increment Boot RAM and Data Buffer Pointers
All data read?
NO
YES
Configure Boot RAM as Program Memory
Send all Read Data to Host
G

FIGURE 10: BOOT RAM ERASE COMMAND HANDLER

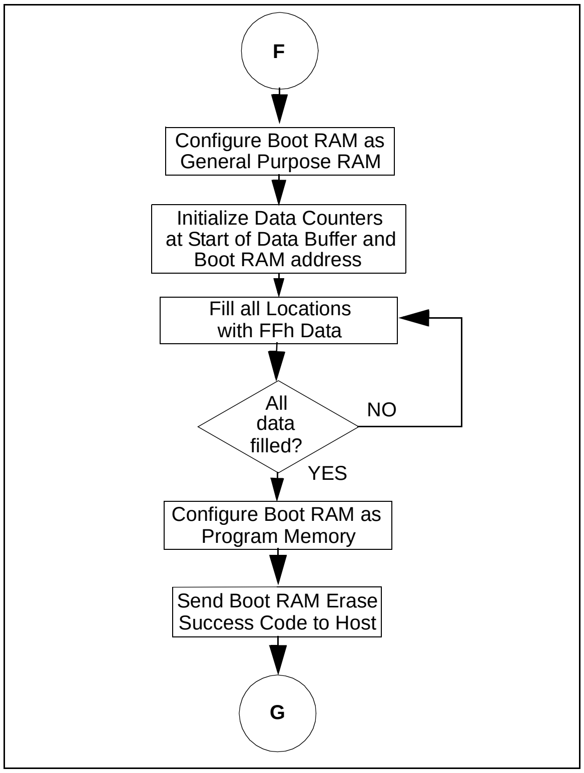
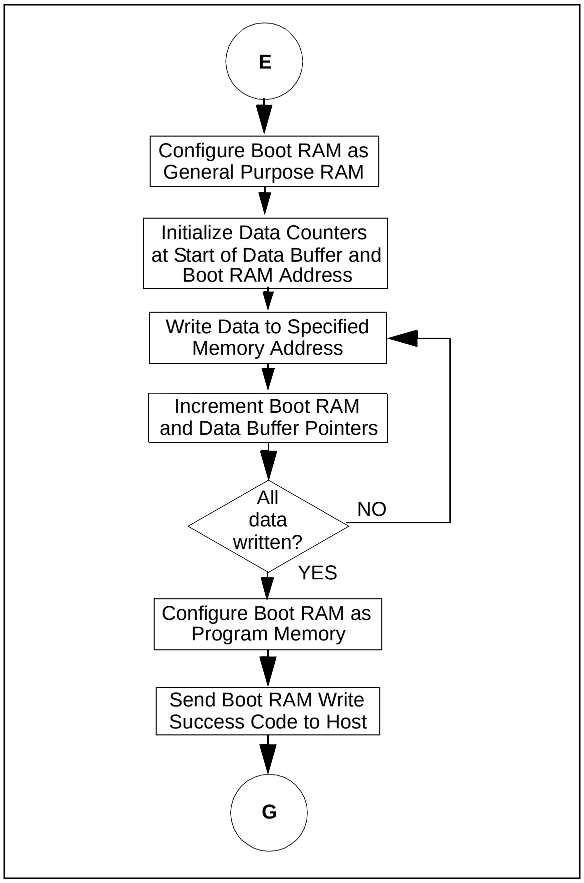

F
Configure Boot RAM as General Purpose RAM
Initialize Data Counters at Start of Data Buffer and Boot RAM address
Fill all Locations with FFh Data
All data filled?
NO
YES
Configure Boot RAM as Program Memory
Send Boot RAM Erase Success Code to Host
G

FIGURE 11: BOOT RAM WRITE COMMAND HANDLER


E
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?
NO
YES
Configure Boot RAM as Program Memory
Send Boot RAM Write Success Code to Host
G

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

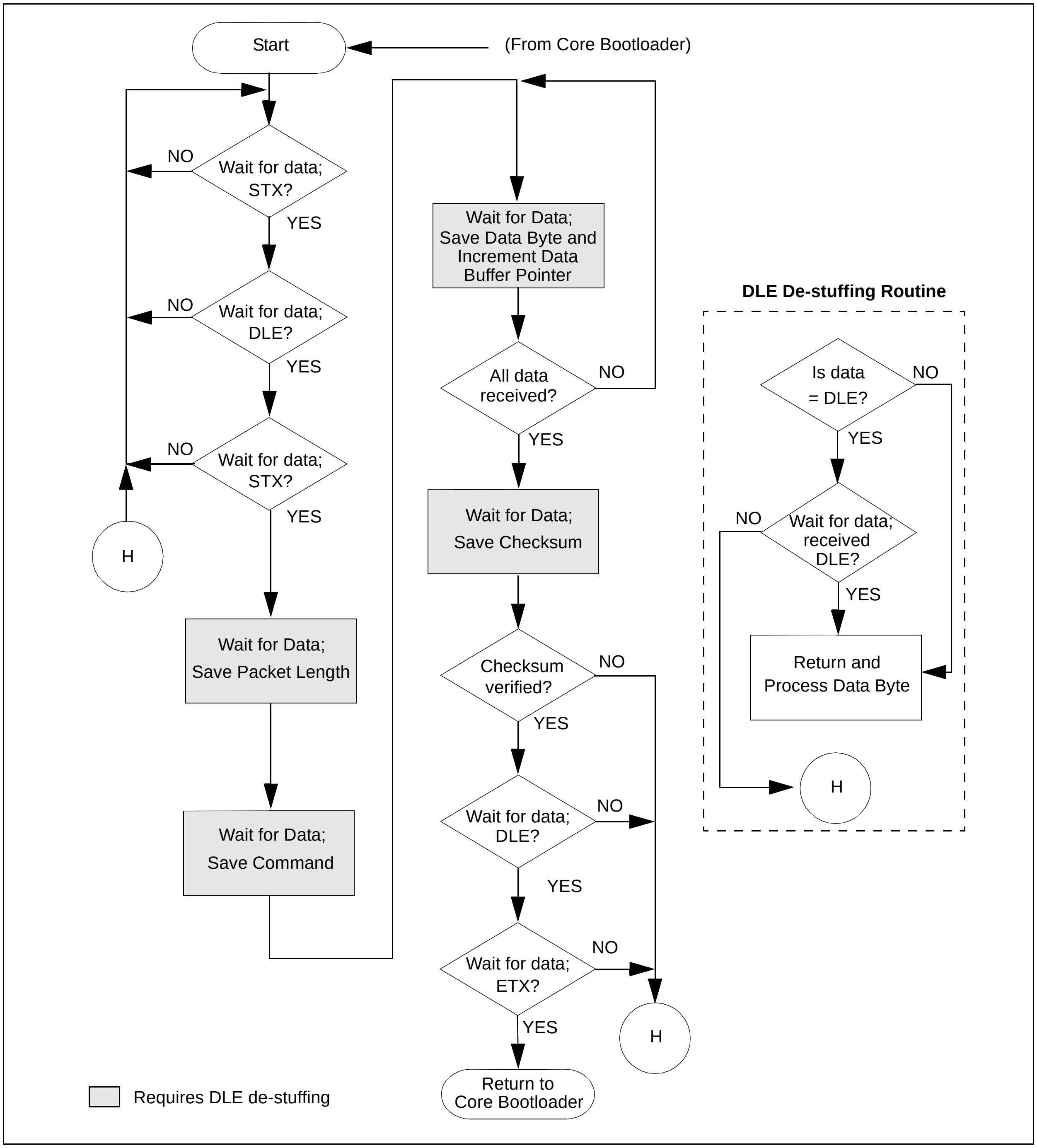
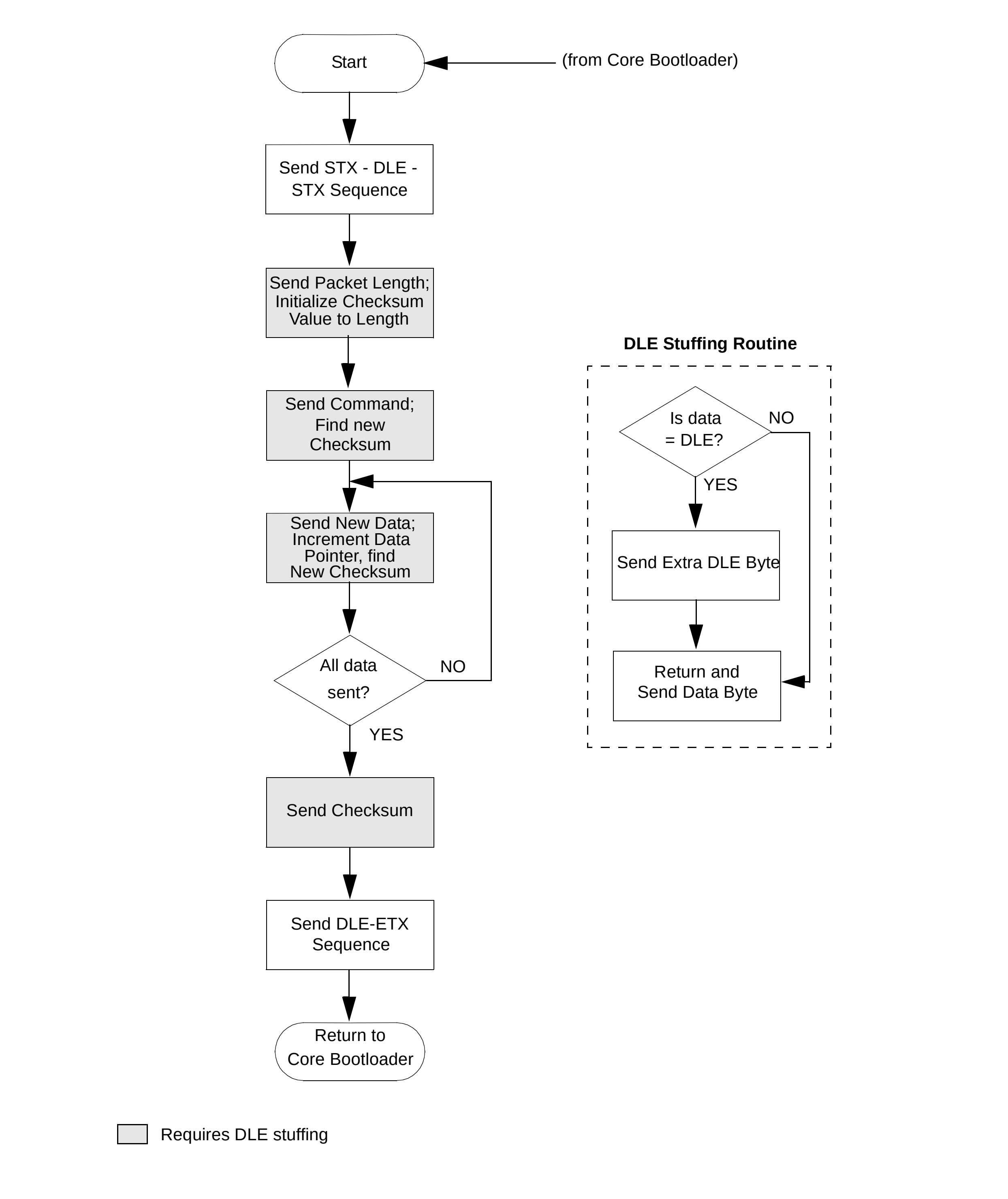


FIGURE 13: FLOW CHART FOR THE SendHostData ROUTINE


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
YES
Send Checksum
Send DLE-ETX Sequence
Return to Core Bootloader
DLE Stuffing Routine
Is data = DLE?
NO
YES
Send Extra DLE Byte
Return and Send Data Byte
Requires DLE stuffing

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 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.

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

Command	Code	Parameters	Response	Description

RD_VER	00h	Len = 0
		Command = RD_VER
			Len = 1
			Result = RD_VER
			Data[0] = Version
				Returns firmware version

RD_MEM	01h	Len = 5
		Command = RD_MEM
		Data[0]=AddrLL
		Data[1]=AddrLH
		Data[2]=AddrUL
		Data[3]=AddrUH
		Data[4]=Len
			Len = 5 + Len
			Result = RD_MEM
			Data[0]=AddrLL
			Data[1]=AddrLH
			Data[2]=AddrUL
			Data[3]=AddrUH
			Data[4]=Len
			Data[5…5+Len]=Memory Data
				Returns memory content from given address

WR_MEM	02h	Len = 6 + Len
		Command = WR_MEM
		Data[0]=AddrLL
		Data[1]=AddrLH
		Data[2]=AddrUL
		Data[3]=AddrUH
		Data[4]=Len
		Data[5]=Flag
		Data[6..6+Len]=Data
			Len = 1
			Result = WR_MEM
			Data[0] = Number of bytes written
				Writes given memory contents to given address

WR_CLR	03h	Len=0
		Command = WR_CLR
			Result = WR_CLR
			Len = 1
			Data[0] = Result Code
			'0' = Success
			'1' = 0 Error Code
				Erases memory

RD_MEM_BOOT	0Bh	Len = 5
		Command = RD_MEM_BOOT
		Data[0]=AddrLL
		Data[1]=AddrLH
		Data[2]=AddrUL
		Data[3]=AddrUH
		Data[4]=Len
			Len = 5 + Len
			Result = RD_MEM
			Data[0]=AddrLL
			Data[1]=AddrLH
			Data[2]=AddrUL
			Data[3]=AddrUH
			Data[4]=Len
			Data[5…5+Len]= Memory Data
				Returns memory content from boot memory at given address

WR_MEM_BOOT	0Ch	Len = 6 + Len
		Command = WR_MEM_BOOT
		Data[0]=AddrLL
		Data[1]=AddrLH
		Data[2]=AddrUL
		Data[3]=AddrUH
		Data[4]=Len
		Data[5]=Flag
		Data[6..6+Len]=Data
			Len = 1
			Result = WR_MEM
			Data[0] = Number of bytes written
				Writes boot memory contents to given address

WR_CLR_BOOT	0Dh	Len=0
		Command = WR_CLR_BOOT
			Result = WR_CLR
			Len = 1
			Data[0] = Result Code
			'0' = Success
			'1' = 0 Error Code
				Erases boot memory

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:

Value	Meaning
00h	This is the first byte being written. User may setup “Write” mode for external memory in beginning of this function.
01h	This is a last byte being written. User **must** change external memory mode to “Read Array”.
02h	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.
All other values	This is an intermediate byte being written. User **may** not need to change external memory mode during this call.

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

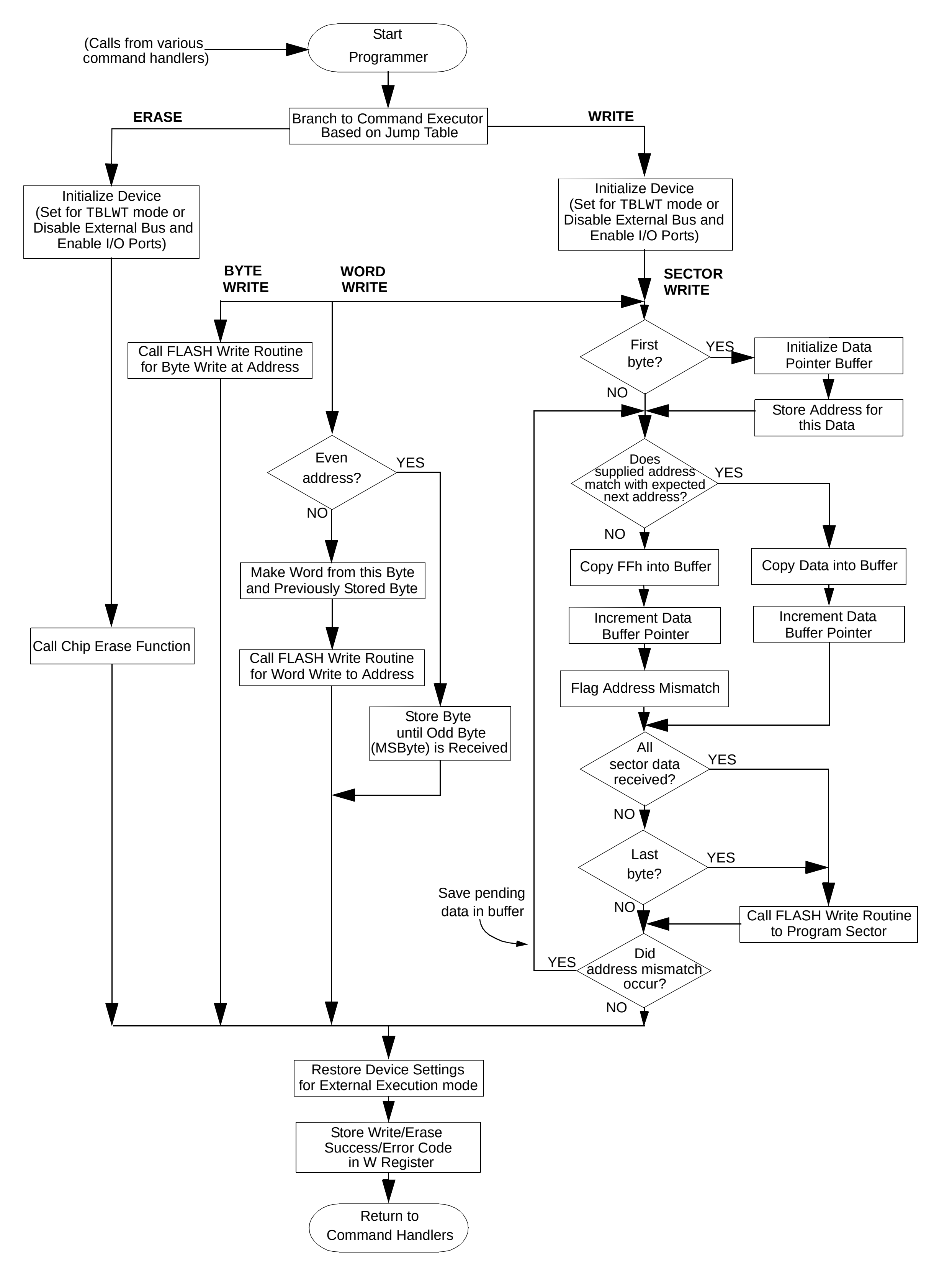

(Calls from various command handlers)
Start Programmer
Branch to Command Executor Based on Jump Table
ERASE
WRITE
Initialize Device (Set for TBLWT mode or Disable External Bus and Enable I/O Ports)
Initialize Device (Set for TBLWT mode or Disable External Bus and Enable I/O Ports)
BYTE WRITE
WORD WRITE
SECTOR WRITE
Call FLASH Write Routine for Byte Write at Address
First byte?
YES
NO
Initialize Data Pointer Buffer
Store Address for this Data
Even address?
YES
NO
Does supplied address match with expected next address?
YES
NO
Make Word from this Byte and Previously Stored Byte
Copy FFh into Buffer
Copy Data into Buffer
Call Chip Erase Function
Call FLASH Write Routine for Word Write to Address
Increment Data Buffer Pointer
Increment Data Buffer Pointer
Flag Address Mismatch
Store Byte until Odd Byte (MSByte) is Received
All sector data received?
YES
NO
Last byte?
YES
NO
Save pending data in buffer
Call FLASH Write Routine to Program Sector
YES
Did address mismatch occur?
NO
Restore Device Settings for External Execution mode
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

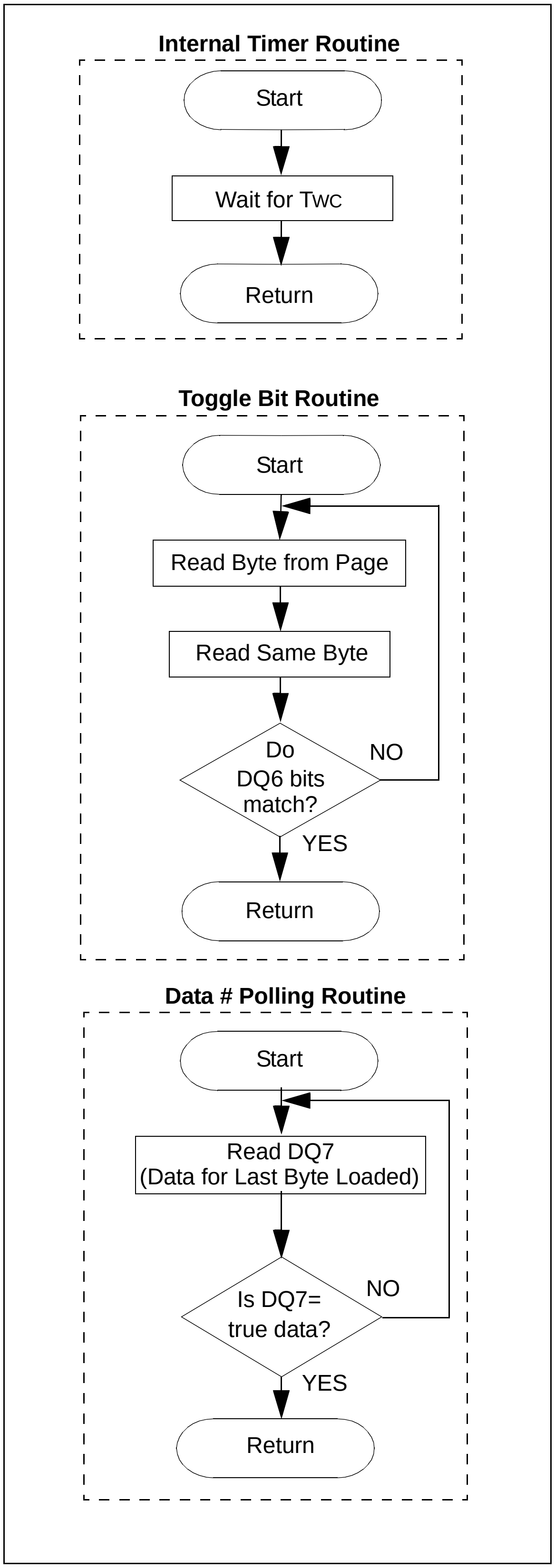
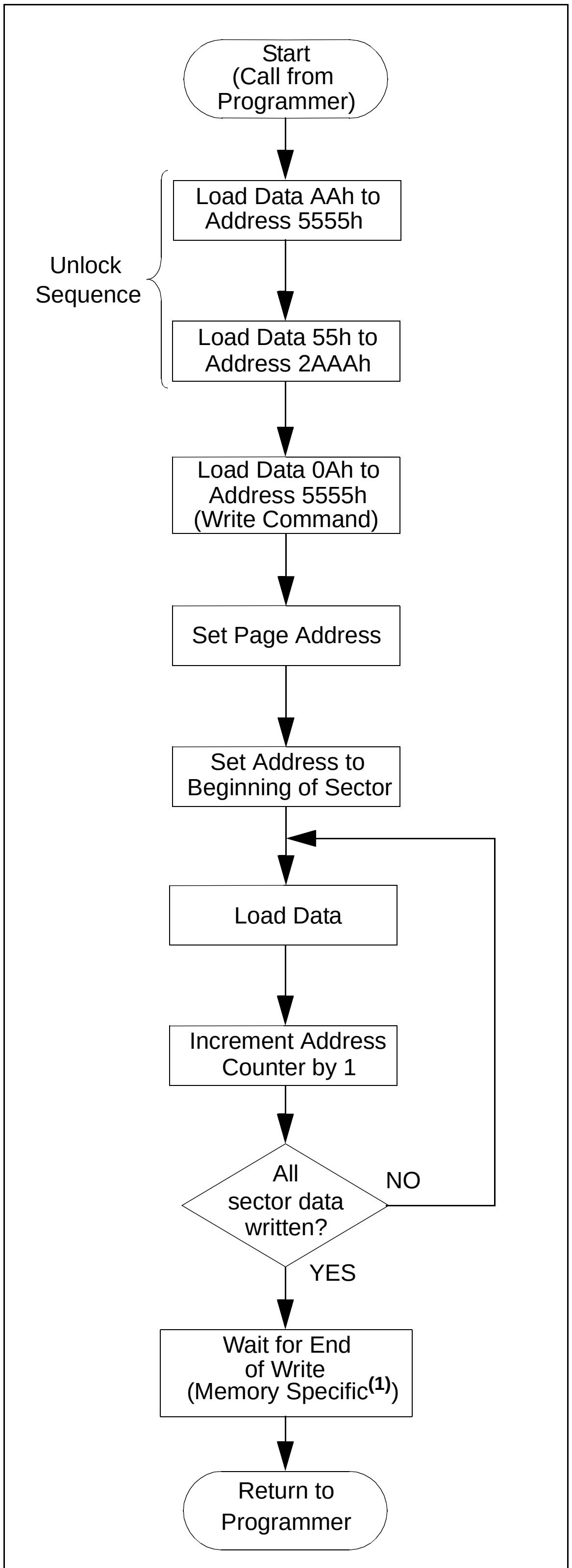
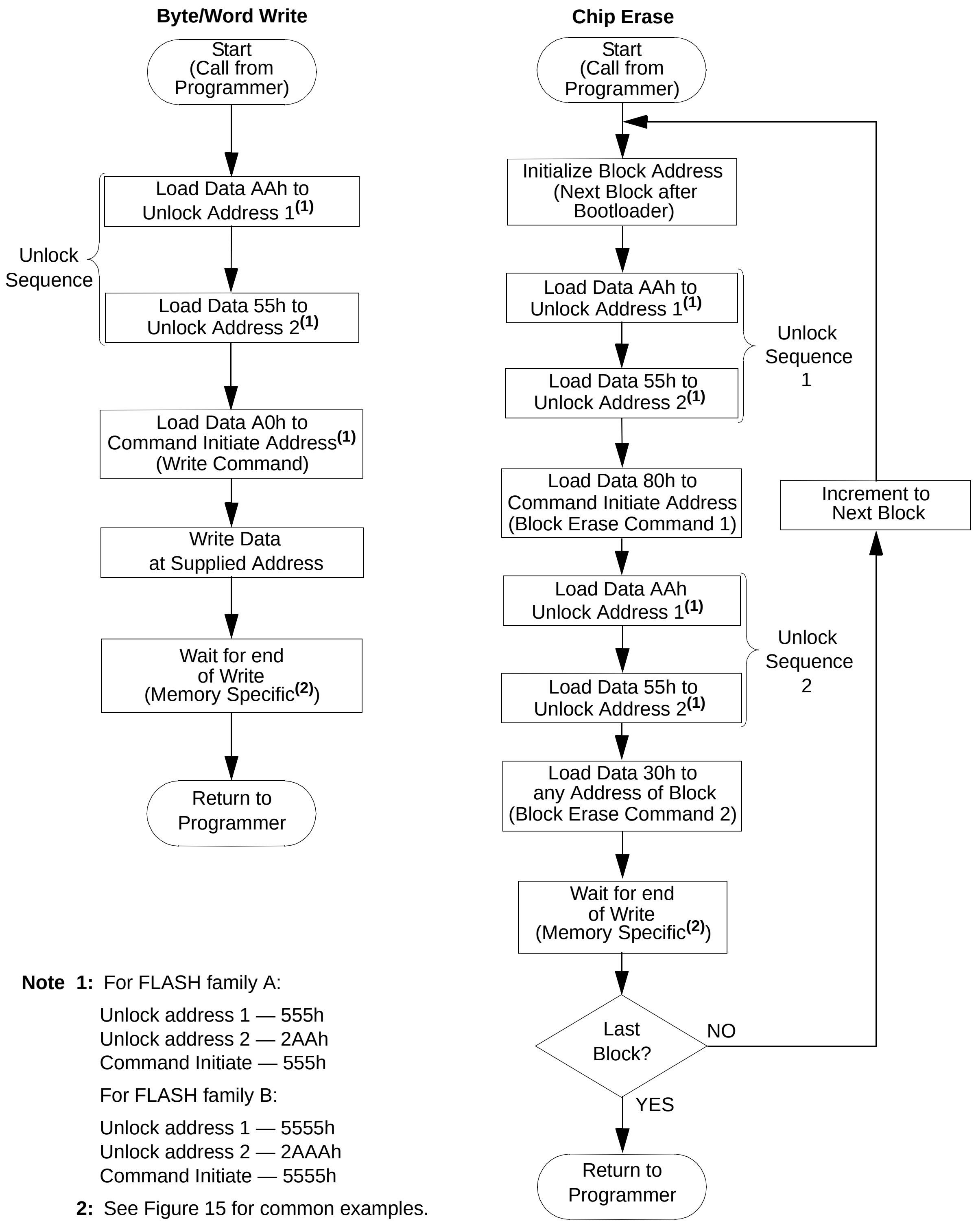


FIGURE 16: TYPICAL WRITE CYCLE ROUTINE FOR SECTOR-PROGRAM FLASH


Start (Call from Programmer)
Unlock Sequence
Load Data AAh to Address 5555h
Load Data 55h to Address 2AAAh
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
YES
Wait for End of Write (Memory Specific(1))
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:

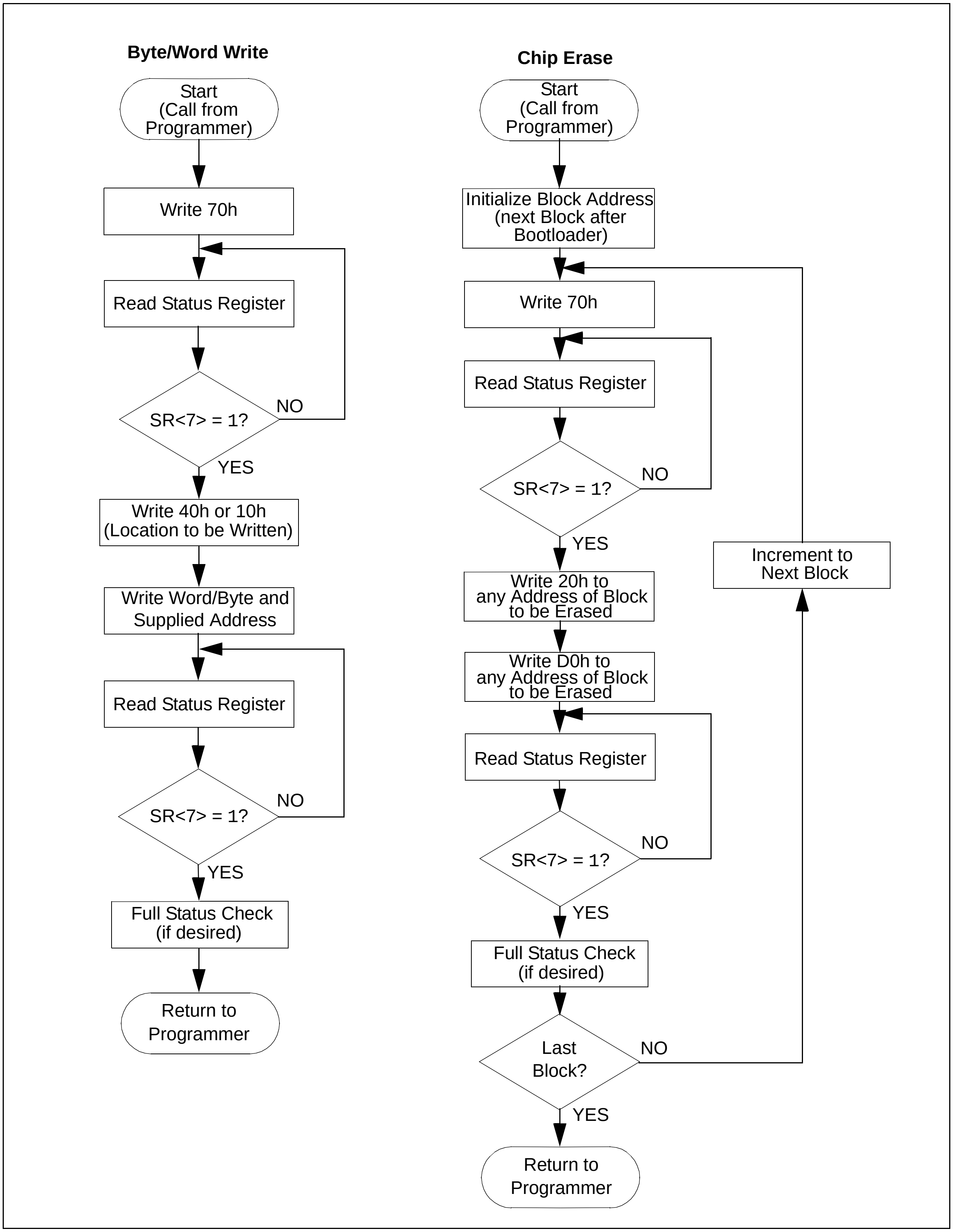
Unlock address 1 — 555h
Unlock address 2 — 2AAh
Command Initiate — 555h

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
Increment to Next Block
Return to Programmer

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

```
            UDATA_ACS  .00
Address     RES        .04     ; Parameter #1 for Write function
Byte        RES        .01     ; Parameter #2 for Write function
Flags       RES        .01     ; Parameter #3 for Write function
```

EXAMPLE 2: EXAMPLE CODE FOR memrtnes.inc

```
Address     EQU        .00     ; 32 bit Address of the location being written
Byte        EQU        .04     ; 8-bit data value to be written
Flags       EQU        .05     ; Specifies whether this is a first,
                               ; intermediate, last or only one byte of
                               ; total data to be written
```

EXAMPLE 3: PROVIDING FOR RELOCATABLE WRITE AND ERASE FUNCTIONS

Bootloader Code section for Calling Write and Erase APIs

```
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 on-board 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

Resource	Bootloader Requirements	
	Core Boot Firmware	**Programmer Firmware***
Program memory (bytes)	1024	Up to 512
Data memory (access RAM, bytes)	32	0 to 3
Data memory (general purpose RAM, bytes)	255	Up to 512
I/O pins	1	System Bus or I/O pins
Peripherals	USART	Generally none

* Requirements vary by specific implementation and FLASH programming algorithm.

EXAMPLE 4: SAMPLE UserCode.inc CONFIGURATION FILE

```
UserCode              EQU     0x1000         ;User Code jump location
HighPriorIntServ      EQU     0x1008         ;Higher priority interrupt
                                             ;service routine jump location
LowPriorIntServ       EQU     0x1018         ;Lower priority interrupt
                                             ;service routine jump location
#define BootLoadChkPin        PORTF,2        ;Boot load Checking port pin number
FOSC                  EQU     D’16000000’    ;Oscillator Frequency
```

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.

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

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

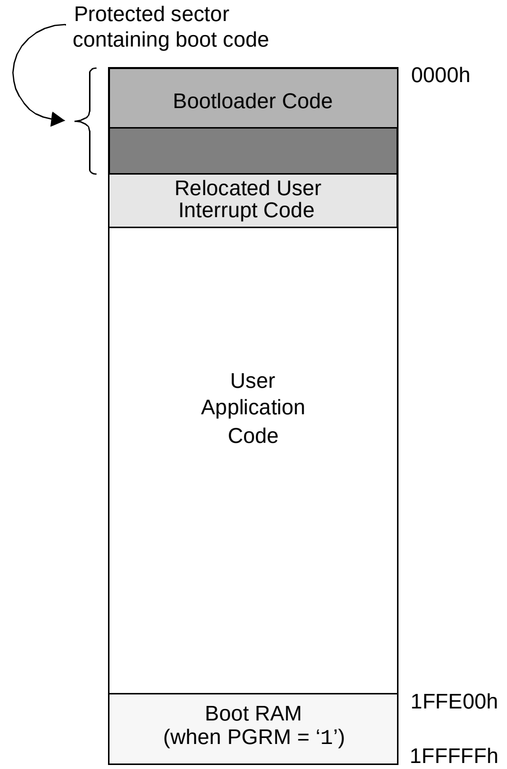
```
// File: UserCode.lkr
// Sample linker command file for User code

LIBPATH .

CODEPAGE   NAME=vectors      START=0x0         END=0x29       PROTECTED
CODEPAGE   NAME=Bootloader   START=0x2A        END=0xFFF      PROTECTED
CODEPAGE   NAME=page         START=0x1000      END=0x200000
CODEPAGE   NAME=config       START=0x300000    END=0x300007   PROTECTED
CODEPAGE   NAME=idlocs       START=0x3FFFFE    END=0x3FFFFF   PROTECTED

ACCESSBANK NAME=accessram    START=0x0         END=0x7F
DATABANK   NAME=gpr0         START=0x80        END=0xFF
DATABANK   NAME=gpr1         START=0x100       END=0x1ff
DATABANK   NAME=gpr2         START=0x200       END=0x2FF
DATABANK   NAME=gpr3         START=0x300       END=0x3FF
DATABANK   NAME=gpr4         START=0x400       END=0x4FF
DATABANK   NAME=gpr5         START=0x500       END=0x5FF
DATABANK   NAME=sfr          START=0xF00       END=0xF7F      PROTECTED
ACCESSBANK NAME=accesssfr    START=0xF80       END=0xFFF      PROTECTED
```

FIGURE 19: MEMORY MAP FOR COMBINED BOOTLOADER AND USER CODE



Note: Sizes of code areas not shown to scale.

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[1]

Manufacturer	Part ID	Programming Algorithm Family[2]	Organization	Basic Byte/Word Addressing[4]	Remarks
AMD	29F series	A	x8		
	29F series	A	x16		
	29F series	A	x8/x16	Byte	
ATMEL	29 Series 29 Series	B[3] B[3]	x8 x16		Sector Programming
	49 Series	B	x8		
	49 Series	B	x16		
	49 Series	B	x8/x16	Byte	
INTEL	Boot Block	C	x8/x16	Byte	
	Strata FLASH/ FLASH File	C	x8		
	Strata FLASH/ FLASH File	C	x8/x16	Word	
SHARP	28F series	C	x8		
	28F series	C	x16		
	28F series	C	x8/x16	Word	
ST	29F series	A	x8		
	29F series	A	x16		
	29F series	A	x8/x16	Byte	
Samsung	FLASH products in this family have multiplexed address/data/command lines, and are incompatible with PIC18C601/801 devices.				
Catalyst	Boot Block FLASH	C	x8		
	Bulk Erase FLASH	[5]	x8 x16		

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)

Manufacturer	Part ID	Programming Algorithm Family[2]	Organization	Basic Byte/Word Addressing[4]	Remarks
Hyundai	29F series	A	x8		
	29F series	A	x8/x16	Byte	
Micron	Boot Block	C	x8		
	Boot Block	C	x8/x16	Byte	
	Even Sectored	C	x8		
	Even Sectored	C	x8/x16	Word	
SST	39F Series	B	x8		
	29EE Series	B[3]	x8		Sector Programming
NexFlash	29F series	B	x8		

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]

Command	Program Algorithm	Bus Cycles												
		Cycles needed	First		Second		Third		Fourth		Fifth		Sixth	
			Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read mode/ RESET	A	1	X	F0	—	—	—	—	—	—	—	—	—	—
	B	1	X	F0	—	—	—	—	—	—	—	—	—	—
	C	1	X	FF	—	—	—	—	—	—	—	—	—	—
Read Mfg. ID	A	4	555	AA	2AA	55	555	90	X00	01	—	—	—	—
	B	4	5555	AA	2AAA	55	5555	90	XX00	01	—	—	—	—
	C	2	X	90	(IA)	(ID)	—	—	—	—	—	—	—	—
Read Device ID	A	4	555	AA	2AA	55	555	90	X01	AD	—	—	—	—
	B	4	5555	AA	2AAA	55	5555	90	XX01	20	—	—	—	—
	C	2	X	90	(IA)	(ID)	—	—	—	—	—	—	—	—
Write	A	4	555	AA	2AA	55	555	A0	(WA)	(WD)	—	—	—	—
	B	4	5555	AA	2AAA	55	555	A0	(WA)	(WD)	—	—	—	—
	C	2	(WA)	40	(WA)	(WD)	—	—	—	—	—	—	—	—
Block Erase	A	6	555	AA	2AA	55	555	80	555	AA	2AA	55	(BA)	30
	B	6	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	(BA)	30
	C	2	(BA)	20	(BA)	D0	—	—	—	—	—	—	—	—
Erase Suspend	A	1	X	B0	—	—	—	—	—	—	—	—	—	—
	B	(2)												
	C	1	X	B0	—	—	—	—	—	—	—	—	—	—
Erase Resume	A	1	X	30	—	—	—	—	—	—	—	—	—	—
	B	(2)												
	C	1	X	D0	—	—	—	—	—	—	—	—	—	—
Chip Erase	A	6	555	AA	2AA	55	555	80	555	AA	2AA	55	555	10
	B	6	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	5555	10
	C	2	X	30	X	D0	—	—	—	—	—	—	—	—
Sector Protect Verify	A	4	555	AA	2AA	55	555	90	(SGA)	00/01	—	—	—	—
	B	4	5555	AA	2AAA	55	5555	90	(SGA)	00/01	—	—	—	—
	C	(2)												

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.

2: Instruction unimplemented in this programming algorithm family.

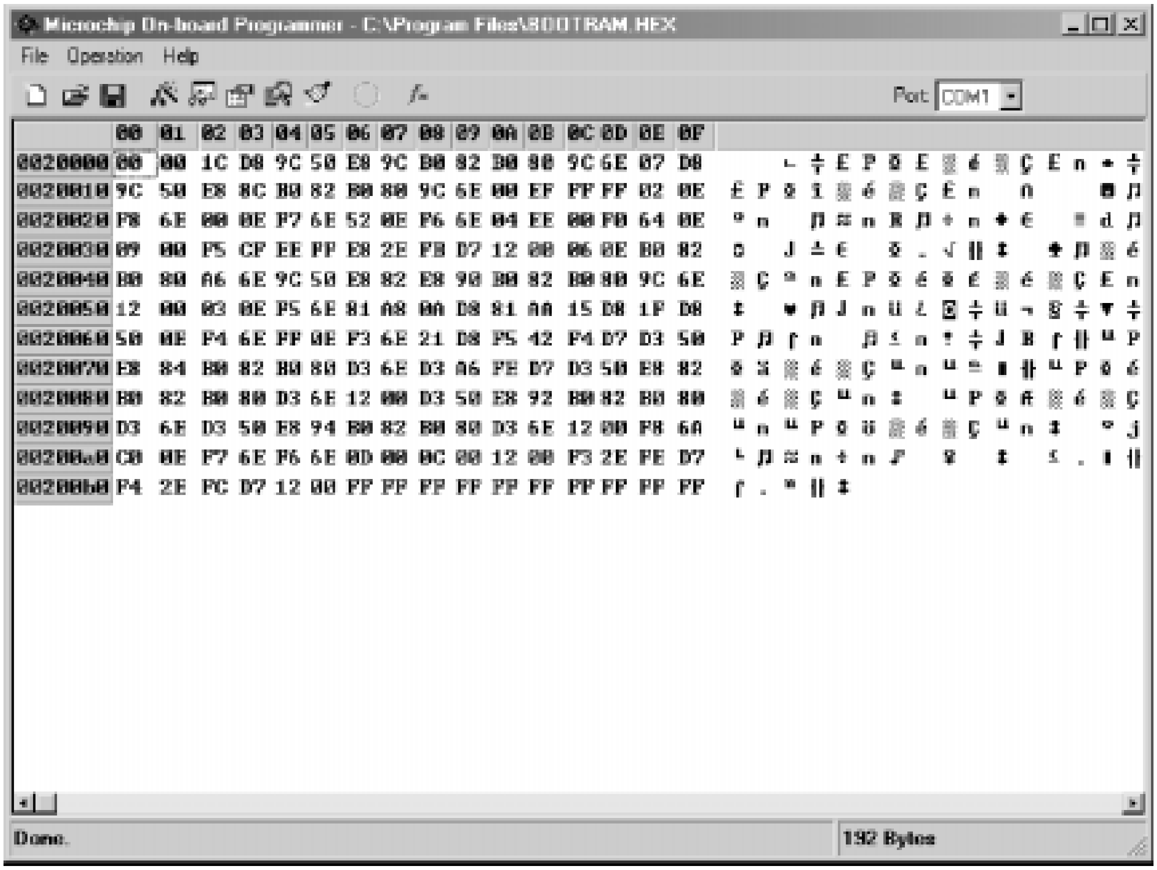
APPENDIX E: THE HOST SOFTWARE WINDOW

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.

FIGURE E-1: HOST SOFTWARE WINDOW



Microchip On-board Programmer - C:\Program Files\BOOTRAM.HEX
File
Operation
Help
Port: COM1

```
         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00200000 00 00 1C D8 9C 50 E8 9C B0 82 B0 80 9C 6E 07 D8
00200010 9C 50 E8 8C B0 82 B0 80 9C 6E 00 EF FF FF 02 0E
00200020 F8 6E 00 0E F7 6E 52 0E F6 6E 04 EE 00 F0 64 0E
00200030 09 00 F5 CF EE FF E8 2E FB D7 12 00 06 0E B0 82
00200040 B0 80 A6 6E 9C 50 E8 82 E8 90 B0 82 B0 80 9C 6E
00200050 12 00 03 0E F5 6E 81 A8 0A D8 81 AA 15 D8 1F D8
00200060 50 0E F4 6E FF 0E F3 6E 21 D8 F5 42 F4 D7 D3 50
00200070 E8 84 B0 82 B0 80 D3 6E D3 A6 FE D7 D3 50 E8 82
00200080 B0 82 B0 80 D3 6E 12 00 D3 50 E8 92 B0 82 B0 80
00200090 D3 6E D3 50 E8 94 B0 82 B0 80 D3 6E 12 00 F8 6A
002000a0 C0 0E F7 6E F6 6E 0D 00 0C 00 12 00 F3 2E FE D7
002000b0 F4 2E FC D7 12 00 FF FF FF FF FF FF FF FF FF FF
```

Done.

192 Bytes

FIGURE E-2: HOST SOFTWARE TOOLBAR

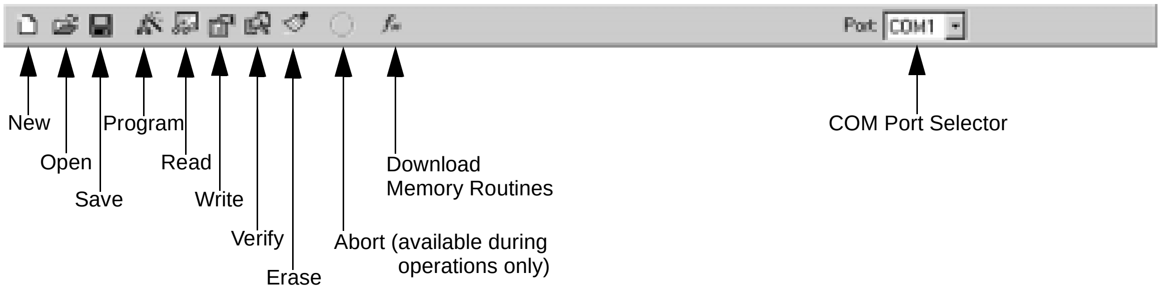


TABLE E-1: SUMMARY OF HOST SOFTWARE COMMANDS

Menu	Command	Keyboard Shortcut	Description
File	New	<Ctrl-N>	Clears the host software buffer and prepares for a new HEX file to be loaded.
	Open	<Ctrl-O>	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.
	Close	—	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.
	Save	<Ctrl-S>	Saves the currently displayed data to the open HEX file. If no HEX file is open, invokes the “Save As” function.
	Save As	<F12>	Save the currently displayed data as an Intel HEX file. A dialog box will prompt for location and new file name.
	Exit	—	Exit the host software without changing or saving the currently displayed data.
Operation	Program	<Ctrl-P>	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.
	Write	<Ctrl-W>	Downloads the current contents of the host software buffer to the target device, without performing Erase or Verify operations.
	Read	<Ctrl-R>	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.
	Erase	<Ctrl-E>	Erases the target memory device.
	Verify	<Ctrl-V>	Verifies the current contents of the memory device against the displayed file.
	Download Memory Routines	<Ctrl-M>	Downloads the contents of the host software buffer to Boot RAM of the target controller, and verifies after download.
	Read Memory Routines	—	Reads the current contents of Boot RAM from the target controller, and displays it in the main window.
	Verify Memory Routines	—	Verifies the current contents of Boot RAM against the displayed file.
	Abort	<Ctrl-A>	Terminates the current operation.
Help	About	—	Displays the current revision of the host software.

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

♻ Printed on recycled paper.

DNV Certification, Inc.
USA

ANSI • RAB
QMS
ACCREDITED

DNV

DNV MSC
The Netherlands
Accredited by the RvA

ISO 9001 / QS-9000
REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs and microperipheral products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.

MICROCHIP

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-7456

Atlanta
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Dayton
Two Prestige Place, Suite 130
Miamisburg, OH 45342
Tel: 937-291-1654 Fax: 937-291-9175

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

New York
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5305 Fax: 631-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC

Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai)
Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai)
Co., Ltd., Chengdu Liaison Office
Rm. 2401, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-6766200 Fax: 86-28-6766599

China - Fuzhou
Microchip Technology Consulting (Shanghai)
Co., Ltd., Fuzhou Liaison Office
Rm. 531, North Building
Fujian Foreign Trade Center Hotel
73 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7557563 Fax: 86-591-7557572

China - Shanghai
Microchip Technology Consulting (Shanghai)
Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai)
Co., Ltd., Shenzhen Liaison Office
Rm. 1315, 13/F, Shenzhen Kerry Centre,
Renminnan Lu
Shenzhen 518001, China
Tel: 86-755-2350361 Fax: 86-755-2366086

Hong Kong
Microchip Technology Hongkong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

India
Microchip Technology Inc.
India Liaison Office
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan
Microchip Technology Taiwan
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Gustav-Heinemann Ring 125
D-81739 Munich, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

10/01/01