AN910

PICmicro® Device Programming: What You Always Wanted to Know (But Didn't Know Who to Ask)

Author: *Eric Somerville*
Microchip Technology Inc.

INTRODUCTION

There is a lot material out there about microcontroller programming. Most of it focuses on the software side of things – orthogonal instruction sets, code optimization, high-level language programming tricks and tweaks in assembler environments, even taking advantage of a device's peculiarities to make it do what you want. This is, of course, a very good thing, as microcontrollers end up doing more complex tasks in more sophisticated applications, the ability to write elegant code becomes more and more valuable.

What doesn't get mentioned as often is the last key part of the process: actually getting that elegant code into the microcontroller itself. The author still remembers well his first experience, in the days before Microchip even existed, of trying to piece together several hardware specifications in order to figure out how the programming process was supposed to work. True, device programming is a vital step, but it doesn't always get the attention it deserves.

That brings us to the point of this application note. For more than a decade, Microchip has published a lot of information about programming its 8-bit microcontrollers. There is now so much information out there, in fact, that it's sometimes hard for engineers and technicians who are unfamiliar with Microchip products to know where to start. The purpose of this paper is to provide the entry point for information on programming Microchip's 8-bit microcontrollers. While we can't cover every programming specification in detail, we can give you a good idea of how the process works and show you what to look for in a specification. We'll also touch briefly on how in-system programmability can affect an application's design and finish with other programming and diagnostic innovations that make applications more reliable.

THE BASICS

If you've had previous experience with programming microcontrollers, you're already familiar with the basic issue: a microcontroller in an embedded application is not a computer. There is no convenient built-in GUI for loading a program. You can't just insert a tiny floppy in the side and press any pin to continue. Instead, you use a hardware protocol to get the device's attention and present it with the intended program in an unambiguous fashion. If the designers have done their job correctly, the device will interpret the incoming data and write it to memory exactly as you intended.

For the vast majority of PICmicro devices, the protocol is known as In-Circuit Serial Programming™ (better know as ICSP™). Since Microchip introduced it in the early 1990s, serial programming methods of some form have become the standard for in-system programming for most microcontroller manufacturers. The protocol allows programming functions to be multiplexed with existing device pins, avoiding the need of tying up precious I/O real estate with ports that might only be used occasionally, if at all.

Although the exact implementation varies from one PICmicro family of devices to another, the basic ICSP protocol remains the same. When the device is supplied with a normal power supply and a specific voltage on the Master Clear ($\overline{MCLR}$) pin, a state machine built into the core architecture takes control. It accepts serial data and clock on two of the port pins and writes the information to the appropriate target memory space. The entire process is controlled by a set of special commands that accompany the serial data stream.

That sounds simple, doesn't it? In principle, the whole system can be reduced to those basic concepts. There are, of course, lots of details that must be observed for things to work correctly and many of those differ depending on the particular device family. It is those details that we will discuss here.

Keep in mind if you're using one of Microchip's device programmers, such as PRO MATE® II or a programmer offered by one of Microchip's approved third party partners, that all of these details will be handled for you automatically. It is those cases where the ICSP operation is used as it was named – that is, programming the microcontroller in the system – that knowing how the process works will help you to understand the issues and make the application work for you.

The ICSP Hardware Protocol

Generally speaking, all PICmicro microcontrollers are placed in Serial Programming mode by doing these three things in sequence:

1. Applying the appropriate power source and ground (V_{DD} and V_{SS}) to the device;
2. Raising the voltage on the $\overline{MCLR}$ pin to the programming voltage level (in general, around 13V), while at the same time:
3. Pulling the two designated I/O port pins to logic low and holding them there.

A few details are worth mentioning here. First, applying V_{DD} and V_{SS} means all digital and analog supply and ground pins, including AV_{DD} and AV_{SS}, not just those pins that are convenient. This is usually stated clearly in the first page or two of each programming specification, which is a separate document from the data sheet. If it isn't, it is safe to assume that the device requires it.

The programming voltage applied to $\overline{MCLR}$, also known as V_{PP}, varies with device architecture. The voltage level (referred to throughout the literature as "V_{IHH}") can be specified as a fixed range for older EPROM devices, or as an offset of V_{DD} in Flash devices. A safe generalization for all PICmicro microcontrollers is a V_{IHH} of 13V; however, it is always best to check the particular device's programming specification first for its particular values. This is why all programming specifications call for a well regulated voltage source with a resolution of ±0.25V.

Besides the level of V_{IHH}, there is a requirement for the transition time to that level. The text in some specifications may be non-specific, but there is almost always a defined and very short interval. There is a very good reason for the brevity: the device may begin to execute whatever is in its program memory if V_{DD} is applied for sufficient time before V_{PP} reaches V_{IHH}. Although this is not an issue with OTP parts, which are shipped blank and will only perform `NOP` instructions, it may cause concerns if a Flash device is being reprogrammed while in the application.

Premature code execution is primarily a concern with applications that use a fast-starting RC oscillator or the device's internal RC oscillator; however, it could be an issue for any oscillator type if the V_{PP} rise time is sufficiently long. In fact, some earlier devices are even more explicit, requiring that V_{IHH} already be on $\overline{MCLR}$ before V_{DD} is provided to the part. This is good practice for any application that uses an RC oscillator and where external Master Clear functionality is disabled. In all cases, it means that V_{PP} not only requires a regulated supply, but an output with adequate drive behind it to bring the level up sufficiently fast. This is particularly important where the circuit uses added capacitance or strong pull-up resistors.

There's also the matter of the ports. Depending on the device, the two programming pins are usually multiplexed with functions on PORTB, most often RB6 and RB7 and are designated "PGC" (Program Clock) and "PGD" (Program Data). For really low pin count devices with only one I/O port (like those in the PIC12 family), the programming pins are GP0 and GP1. If there's any doubt as to which pins are associated with serial programming, refer to the pinout diagram in the device data sheet or programming specification.

As a final clarification on ports: some newer PICmicro devices have a $\overline{MCLR}$ pin that can be turned into something else. Specifically, the pin (usually GP3, RA5 or RE3, depending on the pin count) can function as an input-only port when the MCLRE configuration bit is set; in these cases, external Master Clear functionality is disabled. Even if the pin is configured as a port, however, it can still function as the trigger for ICSP operation. Applying V_{PP} to this pin when it is configured as a digital port still invokes the ICSP protocol as before, provided the other conditions are met.

When all conditions are met for the appropriate setup interval, the on-board serial programming state machine takes control. The device enters an Idle state, where the CPU and all the peripherals are unclocked and then waits for a clock signal to appear on the PGC pin to start the programming process. The programming voltage on $\overline{MCLR}$ is also used by EPROM-based devices (and a few earlier Flash devices) to charge the program memory array and get it ready for the incoming data.

Programming itself is a synchronous process, starting as soon as a clock train is applied to PGC; there is no latency. This is very important to note, as the state of PGD on each and every falling edge of PGC is latched and interpreted as a data bit. If the data stream gets a late start, the net effect is to left-shift the incoming data or commands, with accordingly bad results. Starting the data stream before the program clock is ready will end up right-shifting your data, which is just as bad.

PGC also controls the way that data and commands are clocked in and out. Unlike some serial protocols with a constant clock train, the ICSP protocol differentiates data items by using intervals between short pulse trains. For a command to receive data, as an example, some number of pulses are sent to clock in the accompanying data on PGD. PGC is then held low for an interval longer than one clock cycle; then PGC is pulsed a certain number of times to clock in more data. These longer "PGC held low" intervals mark the boundaries between commands; they are used by the state machine to complete the current command and prepare for the next. The intervals themselves vary, based on their purpose and the particular programming algorithm used by a device family and are spelled out in the programming specifications.

Timely introduction of the data stream is not the only concern. The setup and hold times for data on PGD must also fall within certain specifications to actually be recognized as valid data (that is to say, a valid logic '1' or '0'). There are also minimum intervals that must be observed in separating programming commands from their accompanying data and commands from each other. All of these intervals vary from family to family and are detailed in the appropriate programming specification.

Besides those we've just discussed, there are many more specified voltage levels and timing intervals in ICSP operation. The most commonly used terms are defined in Tables 1 and 2.

TABLE 1: COMMON DC CHARACTERISTIC DEFINITIONS IN MICROCHIP PROGRAMMING SPECIFICATIONS(1)

Symbol	Characteristic	Comments
VDD	Supply voltage for device during programming	Some Flash specifications may give multiple values or ranges under VDD, depending on the operation (program/verify, erase, bulk erase, etc.). Regardless of the application's operating range, the device's VDD must be held in the appropriate range during the programming operation.
VDDP	Supply voltage for device during programming	Usually specified for OTP devices (Flash devices specify one or more VDD ranges; see above). Regardless of the application's operating range, the microcontroller's VDD must be held in this range during programming.
VDDV	Supply voltage during verify	Usually a range specified for OTP devices. Most algorithms require verification at the minimum and maximum ends of the range to ensure proper programming.
VIHH (VIHH1)	Voltage on $\overline{\text{MCLR}}$ to enable high-voltage ICSP™ programming	For most devices, this serves as the hardware trigger and the source for charging the program memory array. Specified as a range.
VIHL	Voltage on $\overline{\text{MCLR}}$ and/or PGM to enable single-supply ICSP programming	This is the secondary trigger and the voltage source for charging the program memory array in single-supply ICSP programming.
VIH (VIH1)	Input high-level (logic '1') limit on PGC:PGD	Defines the lower limit of what will be recognized as a logic '1'. This term may be used in some specifications in the same sense as VIHL. It is not necessarily the same value. "VIH1" is the earlier usage.
VIL (VIL1)	Input low-level (logic '0') limit on PGC:PGD	Defines the upper limit of what will be recognized as a logic '0'. "VIL1" is the earlier usage.
IDD	Current requirement for $\overline{\text{MCLR}}$/VPP during programming	Not always specified; more likely to be seen on OTP devices.
IDDP	Current requirement for device during programming	Total current drawn during programming for all VDD pins; expressed as a typical or maximum.
CIO	Loading capacitance on I/O pins	General requirement for all devices for meeting AC specifications listed in their data sheets. In programming, applies primarily to PGC and PGD.

Note 1: This list represents the most commonly defined DC specifications for ICSP programming and is not exhaustive. Specifications not listed here are defined in the programming specification where they occur.

TABLE 2: COMMON AC TIMING PARAMETER DEFINITIONS IN MICROCHIP PROGRAMMING SPECIFICATIONS(1)

Symbol(2,3)	Characteristic	Comments
TVHHR (tVHHR and TR)	Maximum rise time for $\overline{MCLR}$ (Vss to VIHH) to enter Programming mode	TR is more commonly used with OTP devices. $\overline{MCLR}$ must rise to VPP faster than this value.
TF	Maximum fall time for $\overline{MCLR}$ (VIHH to VSS)	$\overline{MCLR}$ must return to VSS faster than this value.
TSET0	PGC:PGD pattern setup time (minimum time from start of logic '0' to start of $\overline{MCLR}$ rise to VIHH)	Definition for Flash devices.
TSET1	Minimum setup (rise) time for PGD before PGC falling edge	
THLD0	Minimum time to hold PGC and PGD low after $\overline{MCLR}$ rises to VIHH to enter Programming mode	
THLD1	Minimum hold time for PGD before PGC falling edge	Defines the time that PGD must be in a particular state to be clocked in as valid.
Tdly1 (TDLY1(4))	Minimum required time between PGD not being driven and next rising edge of PGC	Reflects the minimum time between adjacent commands, or a data payload and the next command from PGD's point of view.
Tdly2 (TDLY2(4))	Command separation interval (minimum time from falling edge of clock to next rising edge)	Also reflects the difference between commands, or between commands and data payload, from PGC's point of view. PGC is held low during this interval. TDLY1 is always smaller than TDLY2.
Tdly3 (TDLY3(4))	Data out valid time (minimum time from rising clock edge to valid data out)	Defines the time that PGC must be high before PGD is read as valid.
TERA	Erase cycle time (Flash memory only)	May be specified in several ways depending on the erase operation (program memory, entire device, etc.).
TPROG	Programming cycle time (Flash memory only)	May be specified in several ways depending on the programming operation (program-with-erase, program only, low-voltage program, etc.).
TPW	Programming pulse width (EPROM memory only)	Represents the interval between "Begin Programming" and "End Programming" commands, where PGC must be held low. Although always required, this interval is not explicitly defined in many specifications.

Note 1: This list represents the most commonly defined AC specifications for ICSP programming and is not exhaustive. Specifications not listed here are defined in the programming specification where they occur. Specifications exclusive to Parallel Programming modes are not included.

2: The most common typographic representations are shown with alternate versions in parentheses. Except where noted, these symbols correspond to equivalent intervals between timing waveforms for all ICSP programming specifications.

3: Many programming specifications label diagrams by parameter numbers (intervals labelled "P1", "P2", etc.), which are cross-referenced to the listed symbols. The correspondence between parameter numbers and symbols may vary between the specifications for different families. For consistency, it is better to describe AC timing intervals in terms of the listed symbols and not parameter numbers.

4: The most common usages are indicated. All of the TDLY intervals (with the variant representation) have slightly different definitions in PIC17CXXX and PIC18CXXX. Also, TDLY3 is not defined for PIC18CXXX devices; it is called "TVALID" instead.

SINGLE-SUPPLY PROGRAMMING

An additional advantage of many devices with Flash program memory is that they are capable of in-system programming without the regulated 13V source. This method is known as *single-supply programming*, since only VDD is required. Historically, Microchip has referred to this mode as *low-voltage programming* to contrast it with standard ICSP operation and its "high-voltage" VPP requirement. Using this method does involve some trade-offs, however.

Single-supply ICSP programming uses an additional programming pin, labeled PGM, with the other pins required by the normal (High-Voltage) mode. The programming voltage for the memory array is generated by an internal charge pump when normal operating voltage (VDD) is placed on the pin. When the PGC and PGD pins are held to logic low at the same time as VDD is applied to both PGM and $\overline{\text{MCLR}}$, the microcontroller enters Programming mode.

The normal Programming mode is still available to users and can be used just as before. In fact, except for the method of entering the Single-Supply mode, all other programming specifications remain the same. The primary difference is the elimination of the regulated 13V supply requirement, which can be a distinct advantage in adding in-circuit and in-the-field reprogrammability to an application.

The Single-Supply mode is available as an option, which is controlled by the configuration bit, LVP. The default, unprogrammed state of this bit (= 1) enables single-supply programming. Since PGC, PGD and PGM are typically multiplexed with port pins (generally RB3/RB5, RB6 and RB7), this means that the I/O function of these pins is lost when single-supply ICSP programming is enabled. This also means that when single-supply programming is used, additional precautions must be taken to ensure that PGM is always pulled down to a logic low level, in order to prevent inadvertent entry into Programming mode.

For those users who don't need single-supply programming, this mode can be easily disabled by clearing the LVP bit (= 0). However, this can only be done in the standard ICSP mode. Since most applications don't require single-supply programming, a good practice in using devices with this capability is to perform the initial programming in normal mode, setting the LVP bit to '0' in the process. If singly-supply programming becomes necessary later on, it can be restored by setting the bit (again, using normal ICSP mode) back to '1'.

Single-supply ICSP programming is available on select PICmicro devices with Flash program memory. In most cases, devices that implement self-programmable memory (see page 15) also implement the single-supply option.

The Building Blocks of Programming

Although we've established a protocol for getting the information into the part, we still have to successfully write it to the proper memory space. Regardless of the PICmicro device that we're programming, there are several steps we must always perform.

1. **Load the data**. This involves giving a command that unambiguously tells the device, "The stuff following me is a word of data. Get ready to record it." The state machine knows that a specified number of bits after the command are serial data to be serially shifted into a buffer (LSb first).
2. **Write the data to memory**. This command says, "Take the contents of that buffer and put it here." Just where "here" is may be defined by the command itself (for example, program memory or data EEPROM), or may include a program counter by reference. It could also be based on what data was loaded in the last command.
3. **Read back the data from memory**. This is not saying "Give me a copy of what was just in the buffer", but rather, "Give me a copy of what is now in the location that you just wrote to". That may seem like a minor difference, but it can, in fact, be significant; it works with the next step to make sure that data was written successfully.
4. **Verify the data against the source**. This is an external process that compares the data originally sent to the part with what the part has just read from the programmed memory location. This not only ensures that the data wasn't garbled in transmission, but also verifies that the individual memory cells have been unambiguously set to the correct binary value.

Parts of this process are actually handled by the programming state machine inside the microcontroller. Most of it, however, is handled by some agency outside of the device. This is usually some combination of hardware and software that "decides" what sequence is correct for a particular device, queues up the necessary commands and data and toggles the appropriate voltages and signals on the right pins in the right sequence.

As we've already mentioned, there are a number of device programmers that will take care of everything for you – as long as the microcontroller can be placed in a programmer socket or connected to an external device. It is when you need to debug an assembly line programming process, or add in-system field programming to a Microchip-based design, that understanding the details of actual programming specifications becomes necessary.

Managing the Process: The Programming Commands

Now that we know how the information gets into the device, the next question is, “How does the device know what to do with the information?” The answer is simple: programming is controlled by a completely different set of commands that are distinct from the normal PICmicro instruction sets.

In broad terms, there are two versions of the programming command set: one for the mid-range and previous architectures and one for the enhanced PIC18 architecture. How these commands are used to carry out the programming building blocks will be covered later.

PICmicro devices, up to the mid-range level with 12-bit and 14-bit instruction words (from PIC12 through PIC16), all use a common set of 6-bit instructions. For brevity, we’ll refer to this version of the protocol as the “PIC16 method” or the “mid-range method”. In practical terms, this programming method works directly on memory locations under state machine control. Data is loaded, programmed and read back by separate commands. The program counter is incremented with its own explicit command.

Commands are issued by cycling PGC for 6 clocks; data is sent over PGD at the same time. A command is followed by an interval where PGC is held low, to delimit the command from what follows. If the command is used to move data, such as “Load” and “Read”, it is followed by 16 cycles of PGC for the data payload; during this interval, data is clocked in (for a Load), or clocked out (for a Read). As previously mentioned, data on PGD is clocked in on each falling edge of PGC; data being read out is clocked on the rising edge of PGC, starting with the second clock pulse. Since the actual data is only 12 or 14 bits long (depending on the device), the payload is padded with ‘0’s at the leading and trailing ends as needed to get a total of 16 bits.

During a mid-range method programming cycle, only 6-bit programming commands may be executed; instructions from the core instruction set are not available. The timing for a typical mid-range method command is shown in Figure 1.

In contrast, PIC18 devices do not have separate programming commands in the same sense as the mid-range method. Instead, the architecture of this family incorporates instructions (Table Read and Table Write) that allow direct access to the program memory. To enhance this feature for ICSP operation, PIC18 devices use a 4-bit command “shorthand” that actually implements the existing Table Read and Table Write instructions. An additional NOP command allows the user to execute other instructions for the core instruction set under ICSP control; data following a NOP is interpreted as an instruction rather than data and is executed accordingly.

Note:	The only instructions that can’t be executed using the NOP are Table Reads and Table Writes. Attempting this will disrupt the internal timing of the ICSP state machine. Table Reads and Table Writes can only be done in ICSP operation using their 4-bit command versions.

Commands are issued in a manner similar to the mid-range method; in this case, PGC is cycled 4 times for each command, followed by a delimiting interval. When the command has a data payload, PGC is clocked 16 times as data is being loaded, or 8 times as data is being read out. (The difference is because program memory is written as a two-byte word, but read as single bytes.) As with the mid-range method, data is clocked in on the falling edge of PGC. Data being read out, however, is also clocked out on the falling edge, starting with the first clock pulse.

By combining sequences of Table Read, Table Write and Move Register commands, all of the explicit commands in the PIC16 programming command set can be duplicated. This may seem inefficient until you consider that the Table Read and Table Write commands can also implicitly increment or decrement the Table Pointer. These instructions also eliminate the need for separate commands to load and write data, which makes the overall process faster than its PIC16 counterpart. The timing for a typical PIC18 4-bit command is shown in Figure 2.

It is also important to note that, no matter which command method is used, information is sent to and from the device during programming with the Least Significant bit first. This applies to programming commands, as well as any data being loaded or read back.

A summary of the 6-bit and 4-bit ICSP commands is presented in Table 3.

Note:	In programming specifications for PIC18 devices, the terms “SCLK” and “SDATA” are used interchangeably for “PGC” and “PGD”, respectively. The latter designations are the preferred usages. Don’t confuse “SCLK” and “SDATA” with “SCK” and “SDA”, which are names for the multiplexed clock and data pins of the Synchronous Serial Port peripheral. Attempting to program the device with these pins will result in a programming failure, at the very least.

FIGURE 1: EXAMPLE OF A 6-BIT "LOAD PROGRAM MEMORY" COMMAND (PIC16F87XA)

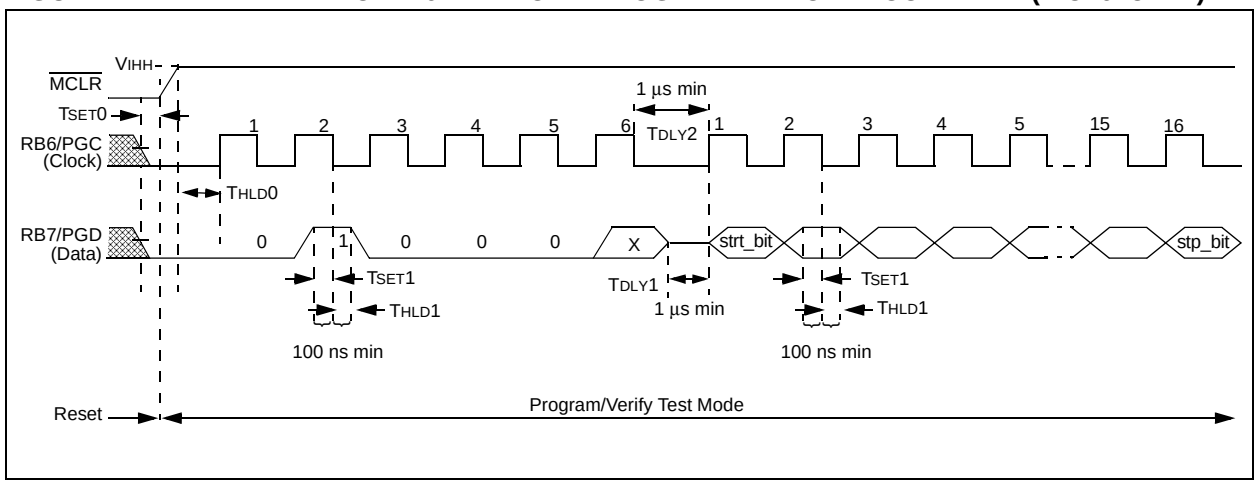


FIGURE 2: EXAMPLE OF A 4-BIT PIC18 COMMAND SEQUENCE (TABLE WRITE/POST-INCREMENT WITH DATA IN PIC18FXX20)

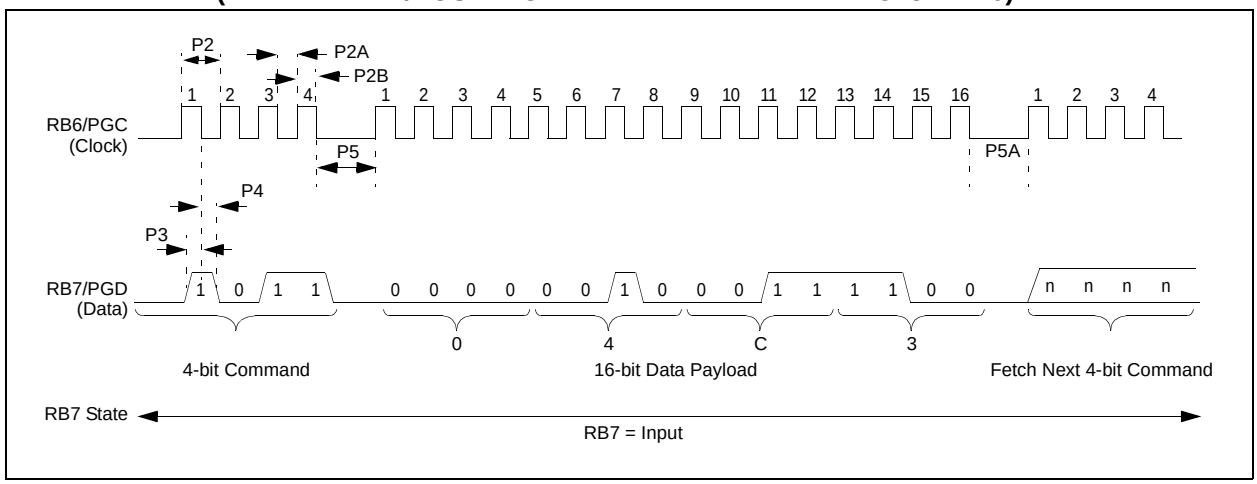


TABLE 3: OVERVIEW OF 6-BIT AND 4-BIT PROGRAMMING COMMAND SETS

6-bit Command Set (PIC12 through PIC16)							
Instruction	**Opcode (MSb...LSb)**						**Data Payload Format**
Load Configuration Word	0(1)	0(1)	0	0	0	0	0, (data (0:13)), 0
Load Program Data	0(1)	0(1)	0	0	1	0	
Read Program Data	0(1)	0(1)	0	1	0	0	
Increment Address	0(1)	0(1)	0	1	1	0	N/A
Begin Programming or Begin Erase/Programming Cycle(2)	0	0	1	0	0	0	
Begin Programming Only Cycle(3)	0	1	1	0	0	0	
Load Data Memory(4)	x(5)	x(5)	0	0	1	1	0, (data(0:7)), 0000000
Read Data Memory(4)	x(5)	x(5)	0	1	0	1	
Bulk Erase Program Memory(3)	x(5)	x(5)	1	0	0	1	N/A
Bulk Erase Data Memory(4)	x(5)	x(5)	1	0	1	1	
End Programming(7,8)	0(1)	0(1)	1	1	1	0	
Chip Erase(9)	x(5)	1	1	1	1	1	

4-bit Command Set (PIC18)					
Instruction	**Opcode (MSb...LSb)**				**Data Payload Format**
Forced NOP (core instruction follows)	0	0	0	0	data (0:15)
Shift Out Value of TABLAT(10)	0	0	1	0	
Table Read, don't change pointer	1	0	0	0	
Table Read, post-increment pointer	1	0	0	1	
Table Read, post-decrement pointer	1	0	1	0	
Table Read, pre-increment pointer	1	0	1	1	
Table Write, don't change pointer	1	1	0	0	
Table Write, post-increment pointer(11)	1	1	0	1	
Table Write, post-decrement pointer(11)	1	1	1	0	
Table Write, pre-increment pointer or start programming(12)	1	1	1	1	

Legend: x = Don't care.

Note 1: These values must be '0' for OTP devices, but are "don't-care" for most Flash devices. Refer to the programming specification for a particular device for exceptions.

2: Implemented as "Begin Programming" for OTP devices and "Begin Erase/Program" for Flash devices.

3: Implemented for Flash devices only. Some devices implement this as an externally timed write requiring an "End Programming" command; other devices do not implement it at all. Refer to the programming specification of a particular device for specific information.

4: Implemented only for Flash devices with data EEPROM. The trailing pad of the data payload is seven '0's.

5: These values must be '0' for some Flash devices. Refer to the programming specification for a particular device for the specific requirement.

6: Implemented for Flash devices only.

7: Implemented for OTP devices and some Flash devices. A very few devices implement this command as 'xx1010'.

8: Rare Flash devices use 'x10111' to terminate externally timed programming.

9: Implemented for rare Flash devices to erase everything on the device, including data EEPROM. Devices without data EEPROM may use this term for what other devices refer to as Bulk Program Erase.

10: Implemented on PIC18F devices only.

11: For PIC18 OTP devices, the increment/decrement is 1; for PIC18F devices, it is 2.

12: For PIC18 OTP devices, the command pre-increments the Table Pointer before performing the Table Write. For PIC18F devices, the command initiates the timed programming cycle.

READING A PROGRAMMING SPECIFICATION

When you approach PICmicro device programming for the first time, you may see a huge array of documents – lots and lots of documents, with each subfamily of devices getting its own specification. When you find the one for your device of interest, you'll find pages and pages of text, diagrams, code examples, timing and voltage specifications, all of it calling for your attention. You may ask, why are there so many specifications? And where do I start?

To begin with, Microchip makes many different types of microcontrollers. Although everything up to the PIC18 devices is based on the same basic Harvard-RISC architecture, there are still many differences. There are several versions of the microprocessor core, each one more complex than the last. Some devices use Flash-based program memory, while some use EPROM; some write to memory one word at a time, while others write in blocks of 4 words . . . in short, there are lots of variations. Each of these factors has an impact on how the device is programmed, even though the same basic hardware protocol is used. It's the unique combination of all of these that determines the programming specification for a particular device family and how it differs from other families.

Regardless of the device family, all of Microchip's programming specifications are organized in the same way. Each document contains at least these basic sections:

- **Overview**. This section lists the devices covered by the document, shows the device pinouts and describes the connection and voltage requirements. It also states the programming protocol used (usually the ICSP protocol, but occasionally something else).
- **Program Memory Programming and Verification.** This section describes the device memory architecture, the command set and specific algorithms for programming. Supporting processes (like erase for Flash memory) are also covered here.
- **Configuration Word, Device IDs and ID Locations**. This section covers the same topics as above for these normally inaccessible memory areas.
- **Code Protection and Checksum.** This topic describes the device code protection scheme and its use. It also covers the device checksum and how it is calculated.
- **AC/DC Specifications**. This section describes the voltage and signal requirements for programming and the specific timing parameters.

Depending on the specific device, there may also be a section to describe data EEPROM programming. Certain other devices may include additional sections for other programming methods (for example, parallel programming in the PIC17C7XX).

Since we've covered the basic things found in the Overview already, we'll use the next few pages covering the remaining topics in approximately the order you would find them in a real programming specification. This will highlight the factors that influence the programming algorithms for different device families.

The Device Memory Map

Although the memory maps are rolled into the program section, they actually show all of the addressable areas of the microcontroller, except for data EEPROM. For that reason, it makes more sense to discuss it separately.

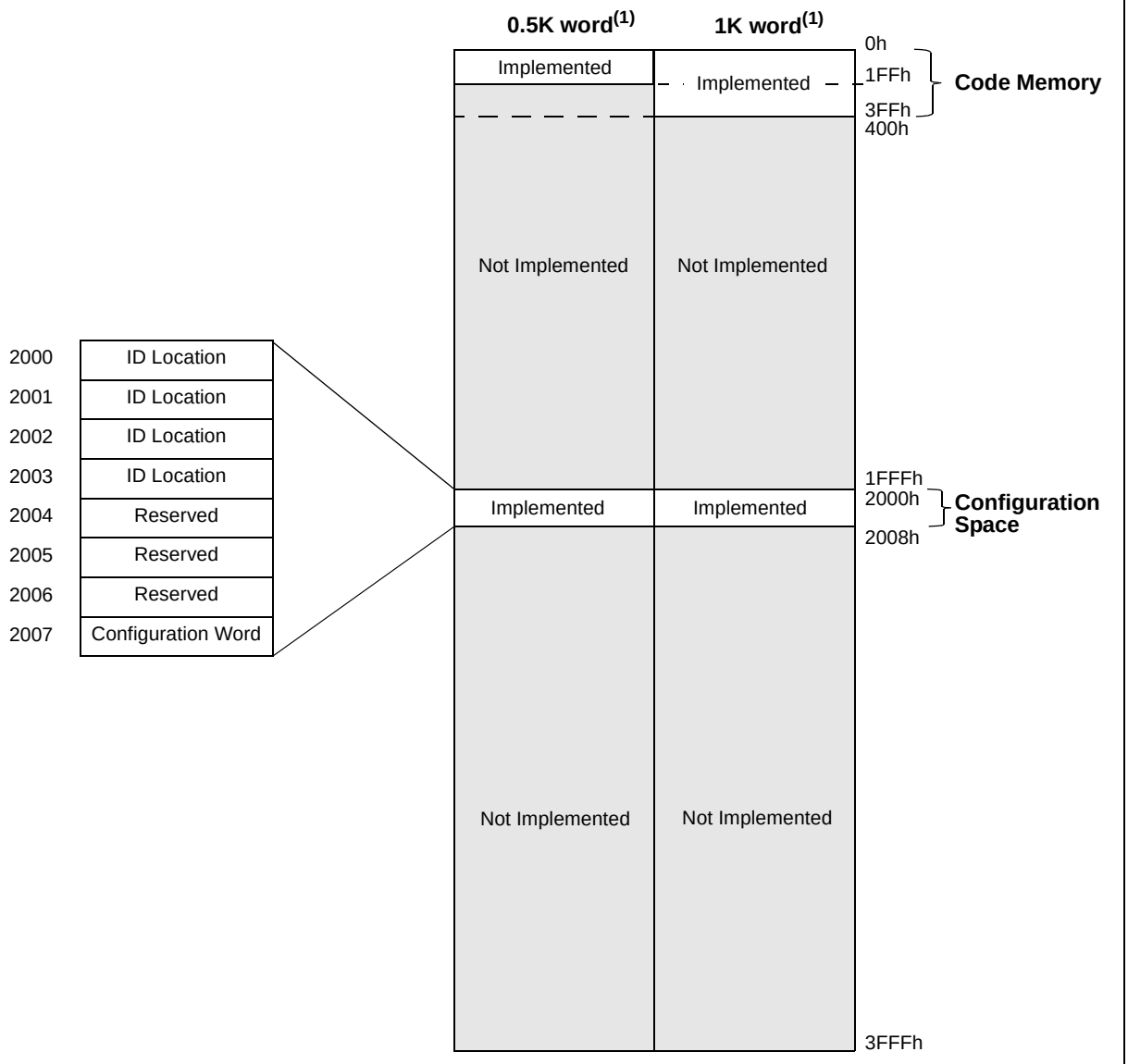
The memory map gives a visual overview of where the target memory locations are in the device's memory space. It also gives a comparison of the size of program memory for different individual devices in the same family. The programmable address space is shown in relation to the entire memory space of the device.

All maps are represented as reaching to the boundaries set by the size of the device's program counter. For most devices up to mid-range with 14-bit instruction sets, the program counter is 13 bits wide, which yields an upper boundary of 1FFF (8K word). For PIC18 devices, the 20-bit program counter yields an upper boundary of 1FFFFFh (2 Mbytes or 1M word). It is important to remember that just because a memory map extends so far, it doesn't mean that all of those addresses are physically implemented. Addresses that are available are indicated with the their boundaries marked. Devices that subdivide their program memories into smaller panels will also show the boundaries for these areas. Unimplemented memory spaces are usually indicated in grey.

In addition to the program memory, it is a convention to show a memory space of equal size extending from the bottom of the program memory space to a lower boundary of twice the program memory (3FFFh for devices up to mid-range, 3FFFFFh for PIC18 devices). Addresses in this range are part of the device's "configuration space", a virtual memory area that is largely unimplemented and not available under normal operating modes. The configuration words, Device IDs and identification locations are mapped to this area and are available in programming modes. This area of the map is usually marked as "Unimplemented" or "Configuration Space". Except for those items already mentioned, it is not available to the user.

Figures 3 and 4 show typical memory maps for PIC16 and PIC18 devices. The latter is usually presented as two diagrams, but are combined into one here to save space.

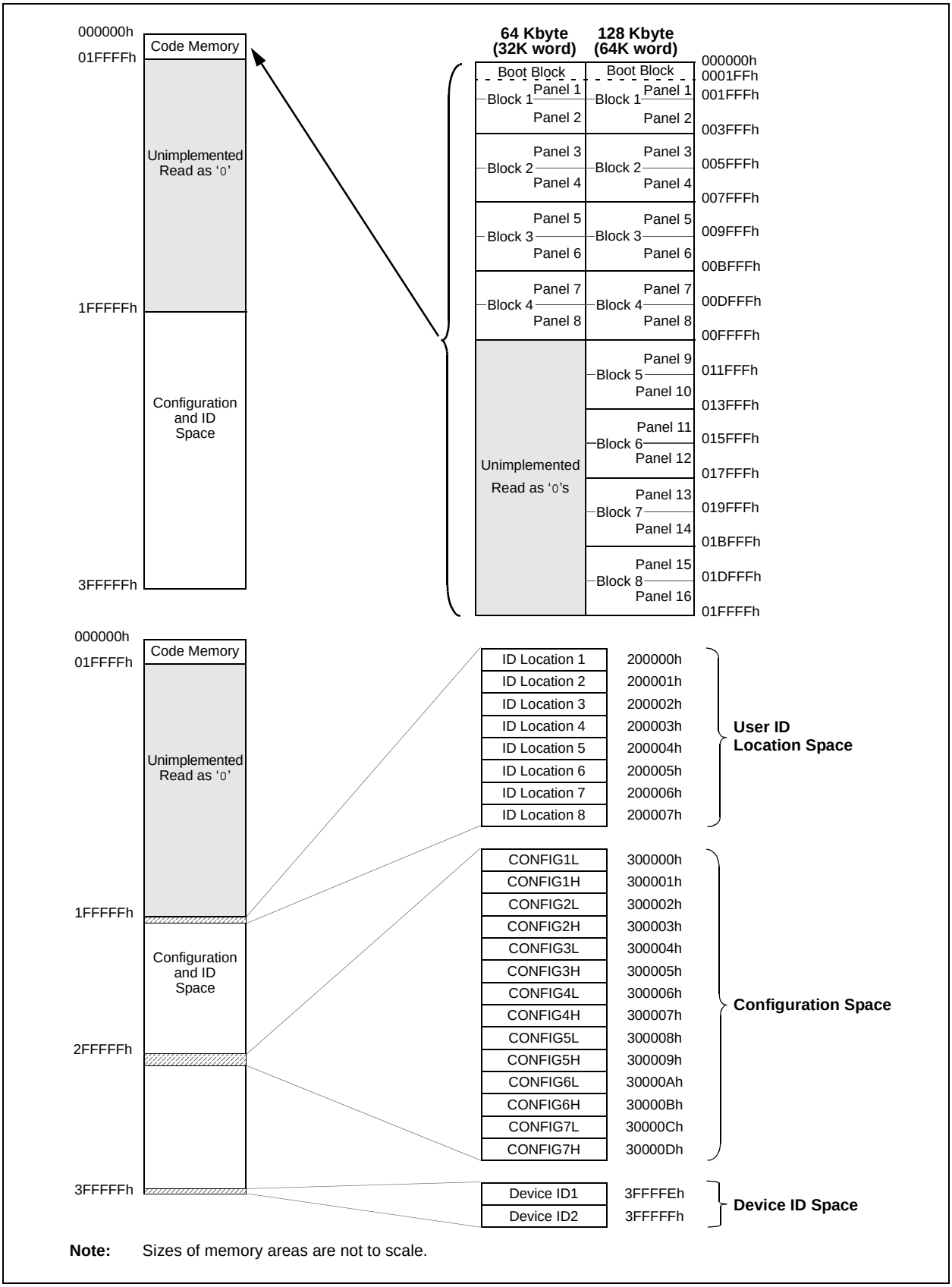
FIGURE 3: TYPICAL MEMORY MAPS FOR PIC16 DEVICES



Note: Sizes of memory areas not to scale.

* Although 0.5 and 1K word program memory sizes are shown here, PIC16 devices are available with program memory sizes up to 8K words.

FIGURE 4: COMBINED MEMORY MAP FOR PIC18F DEVICES, SHOWING PROGRAM MEMORY AND CONFIGURATION LOCATIONS



Note: Sizes of memory areas are not to scale.

Programming, Part I: Program Memory

Although the core architecture is the determining factor in the programming method, the major determinant in designing the actual algorithm is the memory's architecture. As PICmicro microcontrollers have developed over the past decade, the operation of program memory has also changed.

In the architecture of program memory, there are two important considerations: the type of memory arrays and the write/erase block size. We'll examine each of these separately, then consider how they fit together.

MEMORY ARRAYS

The type of basic memory unit used in the program memory array determines the length of time for programming; this is because some technologies just take more effort to acquire and hold the necessary charge to be read consistently as the desired value. Just as important is the support "peripherals" (sense amplifiers, charge pumps, etc.) built around the cells to support their operation. These give flexibility to the operating range and programming conditions and may even allow for other convenient features.

PICmicro devices use two types of user programmable memory arrays: One-Time-Programmable EPROM (usually referred to as "OTP") and rewritable Flash technology. OTP arrays are implemented in one fashion for all devices. Flash arrays are implemented in two ways; the cell technology remains the same, but the array structure varies depending on the particular family's feature set.

OTP Arrays

OTP program memories use the typical cycle of "load data-write data-read back data". The write process is controlled entirely by the programming system and depends on the device architecture. For devices other than PIC18 family members, a write is controlled by separate "Begin Programming" and "End Programming" commands. Between these commands, PGC must be held low, typically for 100 µs. This interval is sometimes referred to as the *programming pulse*. For PIC18 devices, the write is initiated by a Table Write command and ends when the instruction is finished executing. A separate pair of commands, or a Table Write instruction, must be issued for each and every programming "pulse" for every address.

Generally, OTP arrays also require "over-programming", where each location is written to more than once to ensure the proper margin on each programmed cell. Many algorithms will attempt to write a location up to 25 times, verifying the location after each attempt. When verification occurs, the same number of writes is re-applied to the location several more times. For an algorithm that calls for 3x overprogramming, for example, a location that requires 8 pulses to get a verified write will end up receiving 32 pulses in total (8 pulses on the first pass, plus three more passes of 8 pulses). The level of overprogramming varies widely and depends on several factors. When needed, the requirement is clearly spelled out in the device's programming algorithm.

As a side note, the default state of an OTP EPROM cell is a '1'; programming it makes it a '0'. "Clearing" a cell back to its original state of '1' is not possible unless the microcontroller is a windowed device. The entire program memory can be erased in these devices with exposure to UV light.

The flow for a typical PIC16 OTP programming algorithm is shown in Figure 5.

Flash Arrays with Internally Timed Writing

In this version of the array, we also use the cycle of load-write-read back. The difference here is that the write cycle is controlled by a hardware timer in the array itself, which has been calibrated to ensure a proper programming margin on each cell. While each programming pulse may take longer, the elimination of repetitive passes shortens the overall programming time per address.

As noted before, PIC18 devices use the Table Write command to program a location; the data is loaded and written in a single Table Write command. All other devices use an explicit "Begin Programming" command, but do not need a separate "End Programming" command to conclude the write. This saves total programming time. It is still necessary, however, to hold PGC low during the programming interval, which can last up to 10 ms (depending on the device). The actual timing is controlled by the array hardware; the pause simply prevents the state machine from becoming "confused" and terminating the Programming mode early.

A significant difference from OTP arrays is that a Flash array must include specific erase functions. This is only logical, since Flash cells can be written and erased multiple times. (With OTP arrays, you can program a cell once; you can't unprogram it after that.) If you consider the possibility that writing to a location may not unprogram individual bits that were previously programmed, the need for erasure becomes clear.

For all devices except the PIC18 family, erase is generally handled as part of the programming command. All devices use an "Erase Program" command, which automatically blanks the target location before writing new data to it. Many also include a "Program Only" command, which performs a timed write without doing the erase first. This is generally used when the target memory is known to be blank. Separate commands are usually implemented to "Bulk Erase" a particular memory space or clear all memory on the device ("Chip Erase"). The particular erase commands implemented varies from family to family and are discussed in that family's programming specification.

The programming algorithm flow for a typical PIC16 Flash device is shown in Figure 6.

FIGURE 5: TYPICAL PROGRAMMING ALGORITHM FOR AN OTP DEVICE (PIC16C7XX)

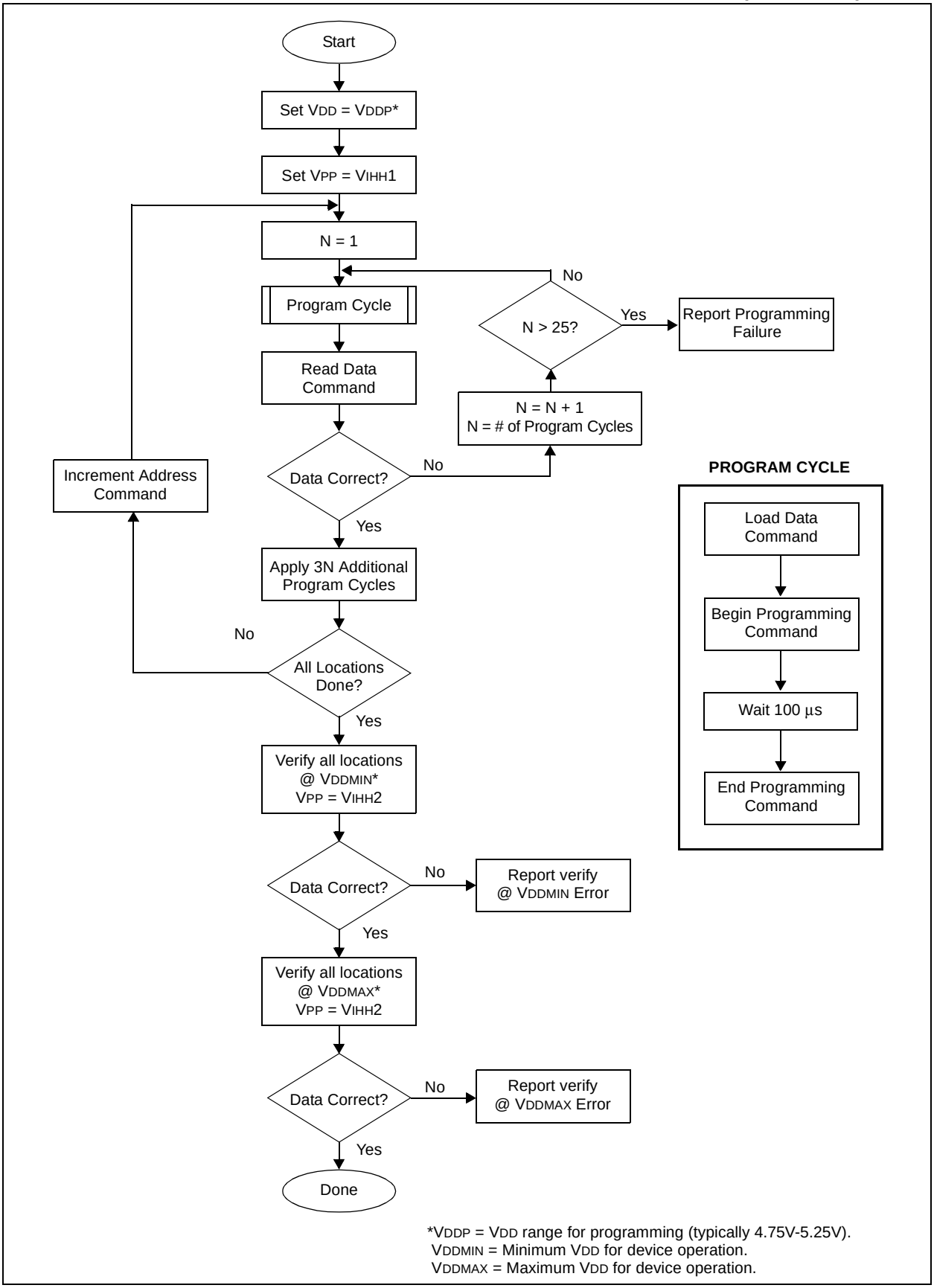
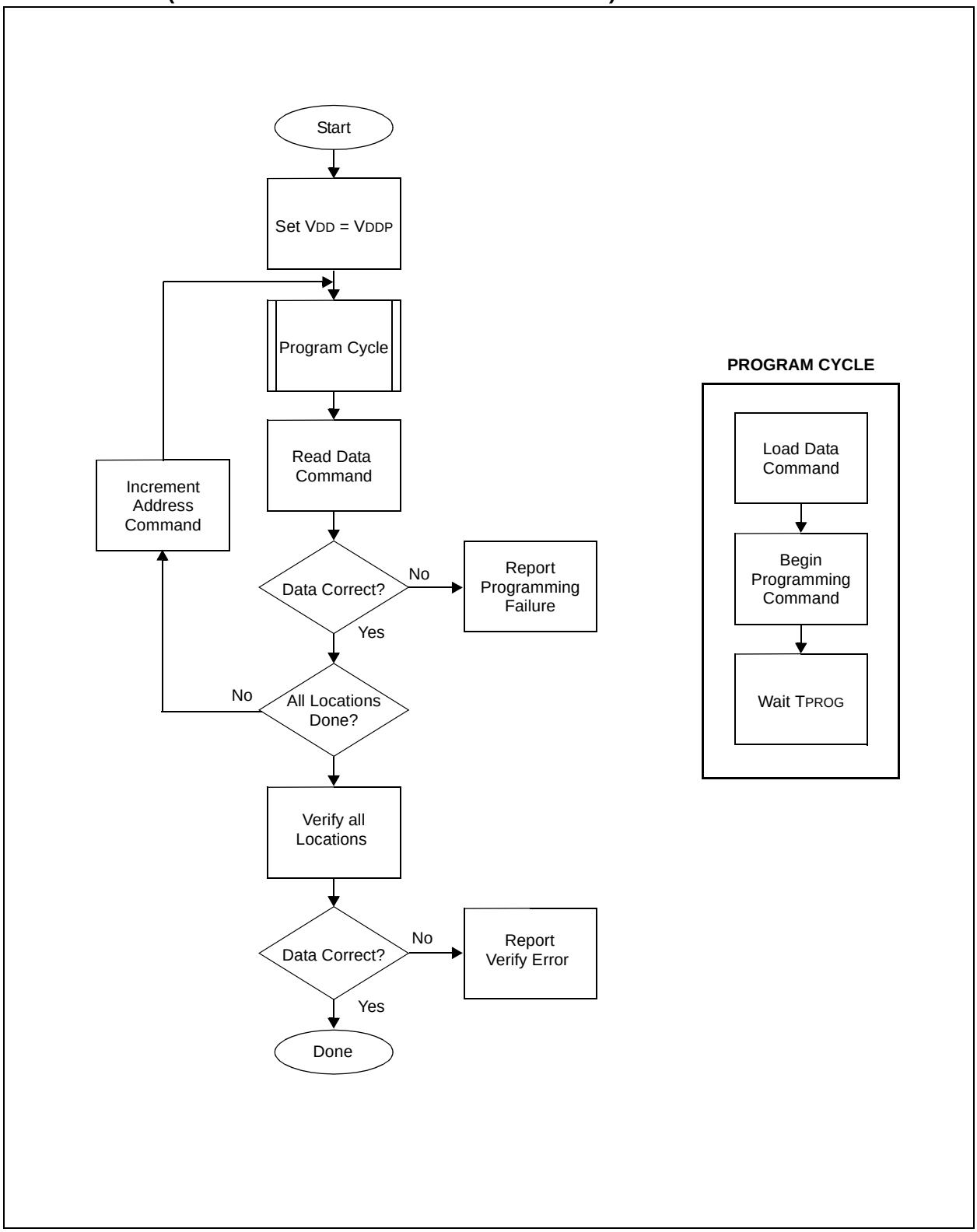


FIGURE 6: TYPICAL PROGRAMMING ALGORITHM FOR A PIC16 FLASH DEVICE (ARRAY WITH INTERNALLY TIMED WRITING)



For PIC18 devices, an erase must be done separately before a write. Special commands that are triggered by writing to the configuration test area are used to clear blocks of program memory, the entire program memory, the configuration words, the data EEPROM or the entire device.

It is also possible to erase smaller program memory locations, down to a level as small as 32 words. The instruction sequence is considerably longer for these small erasures, but the process is convenient when you only need to erase and reprogram a small section of memory. Additionally, this process can be performed throughout the entire V_{DD} range of the device, while erase operations affecting panels or the whole device must be done at the higher end of V_{DD} (typically 4.5V and above).

As an architectural side note, many devices that include internal write timing also add other useful features. These include the ability of the device to write to its own program memory while in normal operating mode – better known as self-programming – and the ability to use single-supply ICSP programming.

Because of their ability to self-program, an additional "safety lock" feature has been added to the core logic; writes to memory under normal code execution require that a specific data sequence be written to a control register prior to each write. This unlock sequence is also required to program certain memory spaces during ICSP operation for PIC18F devices. The PIC16 command set bypasses the requirement for the lock entirely during ICSP operation.

Flash Arrays with Externally Timed Writing

This version of Flash memory has characteristics of both OTP and internally timed Flash arrays. The array design is similar to that used for OTP program memories, in that there is no hardware timer to control the write process. Because of this, the programming algorithms look a great deal like those for OTP arrays. Both "Begin Programming" and "End Programming" commands are required to control the write cycle; PGC must also be held low during the interval between instructions. On the other hand, the memory cells are Flash, so erase-before-write is required; this is generally accomplished with "Bulk Erase" or "Chip Erase" commands. Although it is not controlled by an internal timer, the array hardware still ensures that each cell is properly charged in a single pass.

This array architecture does not have self-programming capability and also cannot be programmed with the single-supply ICSP protocol. It is mainly found in several mid-range device families, such as the PIC16F7X. It is also implemented in a few PIC18F families, such as the PIC18F4610 and PIC18F8490.

Other Variations on Flash Arrays

A small number of PIC16F devices, such as the PIC16F87XA family, incorporate Flash program memory arrays with internally timed write capabilities. They vary from other similar parts in their low-voltage programming algorithms, which require an externally controlled write process. The ICSP command set for these devices includes an "End Programming" command, which is always used after the "Begin Program Only Cycle" (no word erase) command.

Finally, many of the latest members of the PIC18 family (such as the PIC18F4620) take another approach to Flash program memory arrays. These devices incorporate enhancements, such as self-programmability and single-supply ICSP programming, but require external timing to control their write cycles. Unlike other externally timed devices, however, they take their cue from the state of PGC and not an explicit "End Programming" command; specifically, forcing and holding PGC low for a specified interval at the end of a write cycle serves as the signal to end programming.

WRITE AND ERASE BLOCKS

Another difference in programming algorithms is based on how much data is written to the program memory at one time. In the original implementation, PICmicro devices performed write operations one word at a time; the serial data was written to a buffer and written as a whole word when the programming command was given. Many devices in the mid-range PIC16 family still perform writes in this fashion.

Newer Flash arrays perform the write in a different manner. In these implementations, program data is written to the array in increments of up to 8 bytes (4 words) at one time. The algorithm may look the same to the user, with a "Begin Program" or "Table Write" command (depending on the core architecture) being issued for each word to be programmed. In fact, the data is written to hardware buffers inside the memory array. Only when all the buffers are full is the actual write command invoked; all data is then written as a single parallel operation. Because the programming time interval is only needed every 2 or 4 cycles, significant time in programming is saved.

For PIC18F devices, the concept of parallel writing has been extended with a version known as "multi-panel" writing. This involves using a set number of physical buffers (usually 8) for each 8-Kbyte division of program memory. Multi-panel writes work by filling the buffers for all panels, then writing the data to all of the panels at once. The physical arrangement of the memory array makes this parallel write process even faster than the regular (sequential physical locations) process. Devices with multi-panel write arrays operate in this manner by default. This feature may be disabled by modifying the panel control configuration word, usually located at 3C0006h (an address that can only be accessed when the device is being programmed).

An important consideration with write block size is that changing a single location may mean rewriting more than that location. For many devices up to mid-range, the write block is the same size as the data word, so there is no issue with rewriting a single word/memory location; the physical writing is done one word at a time. For PIC18 devices, the program memory is one byte wide, while each instruction is a multiple of two bytes. (Most PIC18 instructions are two bytes, or one word; a few are two words.) This means, for practical purposes, the smallest write block must be two adjacent addresses (two bytes/one word), with the actual write to the memory occurring on every second instruction. A good example of this is the PIC18C452 device. As we've already noted, some devices write up to four words at one time, with the physical write being performed when all the hardware buffers are filled.

A final consideration is the size of the write block in relation to that of the erase block. You might think that write blocks and erase blocks are the same size, since the underlying processes are essentially the same. In fact, the two processes are often implemented with different circuits and affect different sizes of address blocks. This means that algorithms must be adjusted to account for the differences. Part of this includes the alignment of write and erase blocks, which both start at the top of memory and run in block sized multiples.

For a practical example, consider the PIC18F452 and its relatives; the write block is 8 bytes, while the erase block is 64 bytes. If you want to reprogram just 8 bytes of code starting at address 1FC0h, you will have to erase everything from 1FC0h to 1FFFh and store the existing data from 1FC9h to 1FFFh for reprogramming.

Erase block size is not a consideration for devices with OTP memory, since their memories can't be erased. In fact, most OTP devices have a write block of one word (12 or 14 bits for most devices). The exception is the PIC18C452 family, which has a program word size and block size of 2 bytes. Like the PIC18F452 family, data is written in single-byte operations, with the physical write process occurring every second byte write.

VERIFICATION

Reading back a program memory location and verifying its contents is a highly recommended practice; it makes certain that each memory cell has been properly written or erased. Regardless of the memory technology, all programming algorithms include a read-back step immediately following the write. This is done with a "Read Data" command for devices using the mid-range method, or a Table Read for PIC18 devices.

For OTP devices, a major issue in verification has been establishing the parameters for a properly programmed location. Minute differences in cells may mean that a cell properly written to at the operating V_{DD} might not show as properly erased at V_{DDMIN}, or not be consistently read as programmed at V_{DDMAX}.

With these possibilities in mind, Microchip has developed programming algorithms that take these possibilities in account. When the memory has been completely programmed, verification is performed at both the V_{DDMIN} and V_{DDMAX} of the part. This method, known as *intelligent verification*, ensures that each cell has a good "program margin" (it will read correctly across the entire operating range) as well as a good "erase margin" (it will consistently read as '1' after being erased). The need for a range of supply voltages is one reason why ICSP specifications have called for a well-regulated supply voltage with a resolution of ±0.25V.

The majority of Flash devices do not require the intelligent verification algorithm. The same internal write timing that makes overprogramming unnecessary also provides an adequate programming margin for each cell across the entire operating range. All that is required for verification is a successful read-back at the device's operating voltage. The reliability of Flash technology has eliminated the need for intelligent verification in those algorithms and the ±0.25V resolution requirement for V_{DD}. (A well regulated power supply, of course, is still a good idea.)

Programming, Part II: Other than Program Memory and Code Protection

When you're programming a PICmicro device using the ICSP protocol, your main concern is moving the application firmware into the device. But there are other things that should be programmed while you're there, such as the device's configuration settings. Also, it would be nice to preload non-firmware data (program constants, calibration values, etc.). The ICSP protocol makes all of that possible.

CONFIGURATION WORDS AND BITS

Most of the hardware options for PICmicro devices are enabled or disabled using configuration bits. These are individual bits located in one or more program memory words, which are in turn, located just beyond the end of the program memory space. By setting these bits, users can select hardware features that are more or less permanent to their application. Depending on the particular device, this includes things like oscillator configuration, Watchdog Timer prescaler, code protection, low-power features, pin multiplexing for modules and (potentially) many others.

For the mid-range method, configuration words are essentially programmed like any other program memory location. Although the locations are beyond the lower boundary of the program memory space and are not normally accessible, a special “Load Configuration Word” command alerts the device that the incoming data is for the configuration word and forces the program counter to point to configuration memory. (Users cannot force the program counter in this way in normal operation.) Since these devices usually have a block of ID or reserved locations around the configuration word, it may be necessary to increment the address pointer to get to the right address.

For PIC18 devices, the configuration words are more accessible; although these addresses are also outside of the program memory space, they can be written to by using a 22-bit address for the Table Pointer. They are written with Table Write commands, like any other program memory space address; the only difference is they are written one byte at a time. Their values can also be read back with Table Read instructions.

PIC18F devices also use Table Read and Table Write commands for programming their configuration words; they can also access their configuration words under normal operation. For reasons we will discuss shortly, this should be done with considerable caution.

CODE PROTECTION, CONFIGURATION BITS AND ICSP OPERATION

Offered with the other hardware options in the configuration word is the ability to protect all or part of the program memory’s contents. The actual protection system varies enormously from device family to device family. There are two generalizations we can make, however. For most devices other than the PIC18F family, code protection affects either the entire program memory or some fraction of memory. In the latter cases, this is implemented as contiguous blocks of increasing size, going from one end to the other. For example, we can protect the bottom quarter, then the bottom half, then all but the top quarter; we cannot protect the bottom and top quarters, and leave the middle unprotected. The particular scheme depends on the individual device. Data EERPOM, when implemented, is protected separately from program memory.

For PIC18F devices, program memory is divided into several blocks. Each block can be protected separately from external reads and writes and can also be protected from operations requested from within the block. Data EEPROM and the top segment of program memory are protected separately from the rest of code memory.

Code protection is accomplished by setting the appropriate bit(s) in the device’s configuration word(s). This is done as part of device programming and is usually saved as the last step. This should be self-explanatory: if a device is read-protected, it becomes impossible to verify by normal means. This is equally applicable if reading the program memory yields all ‘0’s (as with most PICmicro devices), or yields scrambled data (with devices such as those in the PIC16C5X family). By the same token, write-protecting a device makes attempts at programming rather pointless.

CONSIDERATIONS IN PROGRAMMING CONFIGURATION WORDS

For the same reasons as setting code protection, you must be careful when programming the other configuration bits. Just as code protection can’t be undone once it’s been set, the bits for other features cannot be changed on OTP devices. If you accidentally set the configuration for a crystal oscillator when your application is really using an RC oscillator, your design will probably fail and you’ll be stuck. The problem isn’t nearly as grave with Flash-based devices, but it still means than you will have to erase everything and start from scratch.

A good practice to avoid these programming irritations is to include the configuration word setting directly into your code project; this makes your intended device configuration much less ambiguous. To make this easier, Microchip’s assembly language tools even include a set of compiler directives (the `_CONFIG` directives) which allows programmers to spell out clearly what features they want to enable. These will place the proper bit settings in the actual binary code to be programmed and program those bits at the right time (the end of the cycle).

ID LOCATIONS AND DEVICE IDs

Most PICmicro devices have set aside several words in the configuration memory space for customer use. These words, referred to as ID locations, can be used to store unique identifying information about the device, such as a serial number; their location in the configuration space make them less likely to be altered by accident. These locations are programmed in the same manner as configuration words.

ID locations are different from the Device ID. This is a single word that contains a device-specific identifier programmed at the factory. The most significant bits of the ID (nine in PIC16 devices, 11 in PIC18) specifically identifies the device by part number; the five Least Significant bits indicate the revision level.

Device IDs are always located at the same addresses for particular device families: at 2006h (adjacent to the configuration word) in PIC16 devices, or at 3FFFFFh (the very bottom of the configuration memory space) in PIC18. In either event, the values are read-only.

Device IDs can be used to identify the device for programming and other related purposes. The specific values are listed in the programming specification for a device family and are also called out in most data sheets. Future silicon errata of PICmicro devices will also use this information to differentiate in which part revisions issues occur.

DATA EEPROM

Many PICmicro devices include a block of memory that can be specifically used for storing application data that is not either the operating program or the temporary “scratchpad” data RAM. This particular block of memory is implemented with high-endurance EEPROM cells and is designed to keep static data intact while the device is powered down.

In the majority of PICmicro devices, data EEPROM is not mapped into the program memory space. Instead, it is treated as a peripheral, with its locations being addressed through one or more pointer registers. A few mid-range devices, such as the PIC16F87X family, map the EEPROM into the configuration memory space starting at 2100h.

Data EEPROM can be easily programmed during normal device operation, or through ICSP operation. In the latter, the algorithm is simpler to remember than for program memory, since there are only two variations. How the EEPROM is implemented (mapped or unmapped) has no effect on the programming method.

For devices using the mid-range method, the process is similar to the typical “load data-program-read data-increment” cycle for program memory. Reading and writing to the data EEPROM involves commands that are distinct from those used for program memory. The data payload is still 16 bits following the command; however, since the data EEPROM is only one byte wide, only the first eight bits of data clocked in are read. The word is padded with six ‘0’s following the data, plus the ‘0’s used as Start and Stop bits at either end. (Just as with programming data, information for the data EEPROM is sent LSb first.) The write process is always self-timed, so the “End Programming” command is not used.

During ICSP operation, PIC18 devices use the same general algorithms to read and write to data EEPROM as they do during normal operation. The individual instructions involving the data latch and control registers (EEDATA and EECON1) are shifted into the device as the data payloads of ICSP `NOP` instructions. As with mid-range devices, the write process is self-timed.

The data EEPROM always uses the same type of combination lock used with the internally timed Flash arrays; a specific sequence of writes to the control register must always precede the actual write to the EEPROM. This feature is implemented even for devices in which the Flash program memory can’t self-program. The separate PIC16 ICSP commands for mid-range devices bypass the requirement for the combination lock; PIC18 devices must still use the lock, even during ICSP programming.

The combination lock sequence is discussed in detail in both the programming specifications and the individual device data sheets.

Checksums

As an added protection for the integrity of programming data, Microchip programmers also use a checksum system at the end of the process. In its basic form, the checksum is a bitwise addition of all program memory locations that are not code-protected, the configuration word locations and any masked device ID locations. The actual result used is the Least Significant 16 bits of the sum (all carry is discarded). This is independently calculated by the programming environment for the data to be programmed, as well as the information that has actually been written to the device. A successful match of checksums verifies successful programming.

The checksum scheme also takes code protection into account. The programming specification for each device provides different checksum algorithms for each possible setting of code protection. Each algorithm produces its checksum based only on the unprotected locations. This allows Microchip to publish the checksum algorithms with the confidence that it does not reveal anything that would jeopardize the safety of its customers’ code.

Along with the actual algorithms for calculation, the checksum tables in the programming specifications provide two other useful pieces of information. The first is the “blank value”, which gives the expected checksum when all locations (not counting device ID) are blank or in their unprogrammed state.

The second value, always listed in the far right column, gives the checksum if the part is blank, but the first and last program memory addresses are programmed with a reference value. That value is stated in the heading of the column; it is usually 25E6h for all devices except for members of the PIC18 family and AAh for PIC18 devices. This second version provides an alternate method of validating the checksum algorithm for a given device and level of code protection.

Just as with other programming features, checksums operate behind the scenes when you use a device programming system provided by Microchip or one of its third party partners. Knowing how it works becomes useful if you ever need to design or troubleshoot a programming system.

ICSP PROTOCOL AND ICD

The hardware interface of the ICSP protocol is not limited to just in-system programming. With some additional adaptations, it can also be used for simple monitoring and debugging of applications. This added functionality is known as *In-Circuit Debugging*, or ICD.

The heart of ICD is additional hardware in the microcontroller's core architecture, which serves as a background monitor to the microcontroller's operation. When activated, the monitor uses the ICSP interface pins and some additional controller resources to give an internal view of the device's state at any given time. It also permits users to set breakpoints and single-step the controller through its firmware.

This is, of course, not a complete emulator system; adding those capabilities to every microcontroller would be cost prohibitive and extremely impractical. What ICD does is give users the ability to perform simple debugging with the microcontroller in its actual target application and without the added expense or overhead of emulators, adapter sockets and in-circuit probes.

ICD is implemented only in the most recent versions of Flash microcontrollers. These include members of the PIC16F family with advanced feature sets (such as the PIC16F87X/87XA) and the PIC18F enhanced family.

To use ICD, users must first set the DEBUG bit in the device's configuration word. (DEBUG is occasionally referred to as DEBUG and BKBUG in earlier Microchip documentation.) Setting this bit turns control of the ICSP interface over to ICD operations and enables the debugger's executive monitor.

Users also need an appropriate hardware interface and software to control the debugger's operation. The current choice is Microchip's MPLAB® ICD 2 module, which accommodates all ICD enabled microcontrollers. (The previous MPLAB® ICD module was designed only for PIC16F87X devices and has been replaced by the MPLAB ICD 2 version.) Communication between the external hardware module and the application is accomplished through a standard 6-wire interface.

Finally, the user must have an appropriate software interface for controlling the module. This is a version of the MPLAB development environment suited to the hardware interface. It is important to note that the ICD executive code doesn't come preloaded on each device; it must be downloaded onto the microcontroller, along with the application code, for ICD to become completely functional. Because there is a firmware component, the executive does consume some memory resources. These are detailed in the MPLAB ICD User's Guides and the specific device data sheets.

ICD is not limited to use in end applications. Many of Microchip's development and evaluation kits for PICmicro products are designed to accept the 6-wire ICD interface. This is usually accomplished with an on-board, standard 6-wire RJ-11 receptacle. All MPLAB ICD 2 kits include 6-wire cables and RJ-11 male jacks at either end; these provide the connections between the development board and the MPLAB ICD or ICD 2 hardware module. The whole system provides a quick method for developers to monitor and perform simple debugging of their prototypes using Microchip tools. Although the RJ-11 configuration isn't a requirement, it does provide a convenient standard interface for anyone who may want to add the debugger interface to their applications.

Enabling ICD means that the ICSP interface can't be used for programming – at least not directly. ICD can operate in several different modes, including a programming mode which emulates the regular ICSP interface. This, in turn, allows users to reset the configuration word and disable ICD; of course, this means a complete erasure of the device, which will require reprogramming. It also means that you are not permanently locked into ICD should you choose to enable it.

On the other hand, the MPLAB ICD 2 hardware itself can be used as a device programmer. Additional hardware is required in the form of a Universal Programmer Adapter. Once this has been added, the MPLAB ICD 2 module can be configured by the user to operate as either a debugger/emulator or a programmer (but not both at once).

Users interested in learning more about ICD should refer to the *"MPLAB ICD 2 In-Circuit Debugger User's Guide"* (DS51331). This gives a more complete discussion of ICD in general, as well as requirements for implementing ICD in target applications.

CONSIDERATIONS FOR IN-SYSTEM PROGRAMMING

We can all agree that the ability to program (or reprogram) a microcontroller in its application is a good thing. Without detailing all the advantages, it can allow for big savings in inventory and manufacturing cost, while extending a product's life. Adding this desirable feature does mean making some design adaptations. Some basic circuit considerations are noted here and are shown in Figure 7.

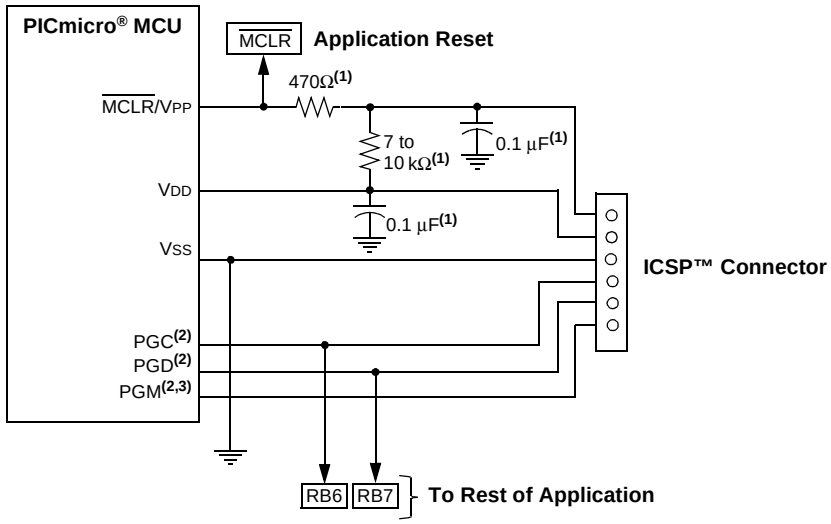
Let's start by looking at the programming voltage, V_{IHH}. For standard (high-voltage) ICSP protocol, it becomes necessary to think about the 13V source. Obviously, the microcontroller can handle it; but what about other components in the circuit? Small circuit boards with tiny surface-mounted devices may not appreciate even brief applications of this voltage level. The question becomes one of isolating the $\overline{MCLR}$ pin from the rest of the system at the programming voltage level, without interfering with the pin's real job as a device Reset.

In addition to the level of V_{IHH}, we must also consider its rise time. As we've already mentioned, a rise time that is too slow may cause the device to prematurely increment the program counter. Large filter capacitors on $\overline{MCLR}$ and V_{DD} can have a significant adverse effect on rise time. So, besides making certain that nothing attached to the $\overline{MCLR}$ pin will block the programming voltage or be damaged by it, we must also make sure that nothing will slow the rise time. If capacitors are necessary for the application to work, the programming device must have sufficient drive to overcome the capacitive load on these lines.

Another consideration relates to V_{DD} as well as V_{IHH}. Regardless of the device, a basic requirement is that the supply and programming voltages stay between the minimum and maximum levels. A sag in either voltage during programming will result in incomplete programming – a potential disaster for OTP devices. The question here becomes one of ensuring that voltage levels are properly regulated to stay within the requirements during the programming cycle.

There's also the issue of the programming pins. If in-system programming is to be built into an application, it must be done so that the PGC/PGD lines can handle a signal of 1 to 10 MHz (the approximate clock rate associated with the ICSP interface), while the rest of the application doesn't load down those lines. Needless to say, the use of series diodes in these lines will probably block programming signals entirely. A simple solution is to make certain that these two pins, typically bits 6 and 7 of PORTB, are always used as outputs in the application. Additional buffering or driving of signals on these lines may be required. It is impossible to make blanket recommendations here, as the range of possible application designs is so broad. Of course, the ideal solution would be to not use these pins for anything but programming, but this may not fit well with the application being designed.

FIGURE 7: TYPICAL CIRCUIT INCORPORATING THE ICSP PROTOCOL INTO AN APPLICATION



Note 1: Component values are recommended starting points for design. Final values must be validated in the actual application. Some applications may not require the use of the $\overline{MCLR}$ series resistor.

2: Because of multiple considerations, no other application components are shown connected to these lines. Actual designs may require passive components. See text for recommendations.

3: PGM is only used in devices capable of single-supply ICSP operation. OTP and some Flash devices use a 5-wire connection.

Some general circuit design guidelines are also provided in the *MPLAB ICD 2 User's Guide* (DS51331). While these are tailored to the requirements of the ICD interface, they also encompass the signalling requirements for the ICSP protocol. As a rule, any design that is signal compatible with ICD will also be compatible with the ICSP protocol, but not the other way around.

All of this relates to both the design of the application, as well as the programmer itself. Device programmers supplied by Microchip and its approved third party partners, whether single socket, gang programmers or in-circuit manufacturing systems, will supply the proper supply and programming voltages at the proper level and with sufficient drive to satisfy level and timing specifications. The load capacity for Microchip's ICSP socket module is indicated in the product literature. Other programming systems must be carefully evaluated to verify that they can meet these requirements. Needless to say, they should also be verified against the actual end application.

This also assumes that programming will be done from an external source, with a 5-wire or 6-wire connector supplying all the appropriate voltages and signals. Building a system that is fully field programmable is another story entirely; that means including a regulated voltage supply for both V_{DD} and V_{IHH} (as needed) into the design, adding parts and complexity. Even if the single-supply ICSP interface is implemented, this means that the I/O pin associated with PGM cannot be used for anything but programming.

These matters are all engineering issues that must be carefully balanced as the system is designed. As the responsible engineer, you must always validate your design against the system's actual programmability and adjust the application accordingly. As often as it is repeated, it still holds true: nothing beats live verification.

BOOTLOADERS AND IN-SYSTEM PROGRAMMING

Even with all the advantages of in-system programming, there may still be cases where constraints on a system's design may prevent including the necessary features. For example, the filter capacitors required to block EMI on port pins in the application may make it impossible to achieve the necessary rise-and-hold values for programming waveforms; or a required application diode on $\overline{MCLR}$ may also block programming voltages. Does this mean that in-system programming has to be ruled out entirely?

Not necessarily. A combination of features in the latest PICmicro devices makes it possible for these devices to program themselves in the end application and under normal operation. This concept, commonly referred to as *bootloading*, can eliminate all of the traditional ICSP requirements for in-system programming. As long as the application has some data channel to the rest of the world, the microcontroller's firmware can be updated. In addition, most devices that are self-programmable can use a bootloader to reprogram themselves, even if code protection has been enabled. Field upgradability does not have to take a secondary role to security.

Three features make bootloading possible. The first, of course, is Flash program memory that can tolerate multiple erase/write cycles. The second is a memory architecture that allows the program memory space to be manipulated directly through the instruction set in normal operating mode. Finally, there is the implementation of a protected block at the top of program memory, residing at the place pointed to by the program counter on device Reset. This area, the *boot block*, can hold special firmware that can write to the rest of the program memory.

Although the concept might seem a little mind-twisting at first, the basic principle of a bootloader is simple. When the microcontroller is reset, it begins to execute the code in the protected block. If it does not detect a predetermined hardware event within a certain time (a pull-up or pull-down on a given pin, a certain bit sequence from the USART that matches a value in data EEPROM, etc.), it does a one-way jump to the main area of program memory and begins to execute the normal application code. If the event is detected, however, it goes to its central bootloading routine. This involves receiving data from a serial channel like the USART, verifying its integrity and writing it to the appropriate memory space using the appropriate commands. Once the routine is done, the device is reset and can now execute the new application code. The only thing that doesn't change is the bootloader itself, which resides in a write-protected memory space.

Bootloaders are best implemented in the PIC18F family of devices, which offer all three of the necessary features discussed above, as well as a wide selection of serial communication options. They can also be implemented in PIC16F mid-range controllers with self-programmable memory arrays; the best example is the PIC16F87XA family, which implements a boot block at the top of program memory. Bootloaders can also be used in the PIC17C7XX family and the PIC18C601/801 ROMless microcontrollers.

Where the device has a programmable boot block, the programming specification will describe how to program and code-protect the space. If you're not sure about a particular device or device family, refer to its data sheet for self-programmability, or its programming specification.

EXCEPTIONS TO ICSP PROTOCOL

It would be nice if we could generalize all of PICmicro programming to the versions that we've just covered. But like most things, there are exceptions to the rules. Because these relate to parts that are in production and enjoy some popularity, we'll touch on them briefly here. Since they are so different, however, we will not go into great detail. If you are interested in learning more about programming these devices, you are encouraged to read the reference specifications for more details.

PIC16C5X Family

This group represents some of the earliest members of the PICmicro family. Even so, it remains one of Microchip's most popular product families.

All of the members of this family use a parallel programming method, requiring the 12-bit program data to be supplied all at once on PORTA and PORTB. Programming mode is invoked in a method similar to the ICSP method, by applying V_{IHH} to the $\overline{MCLR}$ pin. The process is controlled by applying signals to the T0CKI and OSC1 pins.

The programming method is described in the family programming specification (DS30190).

PIC17CXX and PIC17C7XX Families

This family serves as a stepping stone between the mid-range PIC16 devices and the enhanced PIC18 family. It incorporates many of the high-end peripheral features available in PIC18 devices with an instruction set that represents an expansion of the PIC16 set. Many features in this family were implemented in a fashion different from either the PIC16 or PIC18 families; programming is one of the most notable differences.

Devices in the PIC17CXX and PIC17C7XX families are usually programmed using a Parallel mode. This operates very much as it sounds: the 16-bit program data word is presented all at once as 16 parallel bits across two entire ports (PORTB and PORTC). Like PIC18 devices, programming is accomplished through Table Read and Table Write commands. Unlike other PICmicro devices, however, programming is actually a form of bootloading: the PIC17 devices are actually executing code from a built-in protected ROM.

In addition to the data on PORTB and PORTC, programming is controlled using the five lines of PORTA, plus control voltages on the $\overline{MCLR}$ and TEST pins. Since the device is executing code during the process, it also requires a clock source (either on-chip or external). Obviously, this is not a convenient method for in-system programming.

The PIC17C7XX devices can also use a variant of the ICSP interface. This method is very different from what we've discussed so far, however. To begin with, this method is also a bootloader method in disguise. Instead of triggering normal ICSP operation (a passive state machine under external control), the device is actually executing code from the protected ROM. All told, PIC17C7XX devices require 7 connections for ICSP operation, as opposed to 5 for other PICmicro devices.

The serial programming commands for these devices are 8 bits and represent a superset of the basic PIC16 commands. Generally speaking, they are same commands padded with '0's in the two Most Significant bits.

For a more detailed discussion, refer to the programming specification for the PIC17C4X family (DS30412) or the PIC17C7XX family (DS30274).

PIC18C601/801 ROMless Devices

At first, this may seem counter intuitive: how do you program a device without a program memory?

Actually, this isn't nearly as odd as it sounds. The PIC18C601/801 devices incorporate a data memory implemented with RAM that can be programmed through the ICSP interface. The bottom 512 bytes of this memory, in turn, can be configured to act as program memory by setting the PGRM bit in the MEMCON register. While 512 bytes doesn't sound like much, it is enough space to store a bootloader program and this lets the user write to any connected external memory device in a flat memory space of up to 2 Mbytes. Placing the bootloader in volatile RAM means that it will need to be loaded every time it's required, but the vast majority of applications will not require it on every device Reset.

In addition to the bootloader, the user must also load a memory specific configuration file; this lets the bootloader know how to read and write to the type of external memory being used (if the memory is writable). The latest versions of MPLAB IDE provide a collection of configuration files that represent the most common Flash, ROM and PROM chips available. Selecting the right file is done from a special dialog menu, enabled only when the PIC18C601/801 is selected as the target device.

It is not possible to directly program the external memory connected to these devices using the ICSP operation. However, the bootloader operation offers many advantages over the ICSP operation: the device can program itself, during normal operation, by using a convenient serial channel without the need of a programming voltage source or other special conditions. The user can also use a wide range of memory sizes and architectures, giving a great deal of flexibility to the application's design.

Of course, ICSP operation is also used to program the configuration words, just as it is with any other device in the PIC18CXXX family. The details for programming the configuration words, as well as the data memory, are covered in the PIC18CXXX family programming specification (DS39028).

An in-depth discussion of the PIC18C601/801 bootloader can be found in *AN819, "Implementing Bootloader Firmware for the PIC18C601/801 ROMless Microcontroller"* (DS00819).

‘CR’ Devices

It is possible that someone may ask about programming a PICmicro device with a ‘CR’ designation in its name. A good example would be the PIC16CR83. The best response is to first ask in detail about the type of programming they had in mind.

Devices with the ‘CR’ designation refer to devices with a masked ROM program memory; the actual device program is designed into the memory during manufacture and is a permanent architectural feature. It can’t be changed, ever.

PIC16CR devices that are ROM versions of PIC16F parts (like the aforementioned PIC16F83) may have data EEPROM. This is still accessible under the ICSP protocol and can be programmed like any other device. If you want to change the firmware on one of these, however, you’re out of luck.

PRODUCTION PROGRAMMING: MOVING BEYOND ICSP PROTOCOL

No discussion of programming is really complete without considering the next big step – what happens to an application when it moves into mass production.

The methods that we’ve described so far made an assumption of development level programming: that is, programming one or several devices at a time, using an out-of-application socket programmer or a plug-in external system. Everything we’ve talked about can be scaled up with some ease to an assembly line level, doing hundreds to thousands of units a day. We’ve also assumed that we want or need our firmware to be field upgradable under all circumstances.

But this may not be the only case. For example, what of an application with stable and mature code that ships in the millions of units? When something becomes that popular, it may become necessary to think of other bulk programming options.

For those applications that have reached this level, Microchip has factory-based options available. Programs such as Quick Turn Programming (QTP) provide customers with bulk programming for their devices; the act of reordering parts also takes care of the programming step and its associated time and labor. Its close relative, Serialized Quick Turn Programming^{SM} (SQTP^{SM}), also permits individuation of each part with a serial number or other descriptor in a customer specified format.

Make no mistake, in-system programmability and upgradability are good things for your applications. But they are not the end of the story.

WRAPPING IT UP

With its many permutations, the process of device programming may look daunting. But think back to when you first learned how to write code. Once you learned the basic instructions, it became comprehensible. As you progressed, the tangle of instructions became logical, then easy to understand. And finally, it became second nature.

In the same way, PICmicro device programming is no more complicated than designing that elegant code for a sophisticated application. Reduced to its basic terms, it’s a few simple instructions and a hardware protocol, tailored to each device it works with. In fact, the best way to think of it is the next logical step beyond the code.

APPENDIX A: ADDITIONAL READING

As already mentioned, this paper is only an introduction to the realm of programming PICmicro micro-controllers. For in-depth information on the details, as well as specific information on particular devices, the reader is encouraged to review Microchip's most current literature on device programming and specific programming specifications.

These Microchip application notes are excellent resources for information on Flash-based bootloaders:

AN819, "Implementing Bootloader Firmware for the PIC18C601/801 ROMless Microcontrollers" (DS00819), G. Kavaiya and N. Rajbharti, 2001.

AN851, "A Flash Bootloader for PIC16 and PIC18 Devices" (DS00851), R. Fosler and R. Richey, 2002.

AN247, "A CAN Bootloader for PIC18F CAN Microcontrollers" (DS00247), R. Fosler, 2003.

All documents are available in Adobe Acrobat format on the Microchip corporate web site:

www.microchip.com

Note:	Programming specifications for device families may be revised from time to time to reflect new information or an occasional typographic correction. If you have a copy of a particular specification on hand, be sure to check the Microchip web site to see if it has been updated. Revision level is indicated by the letter following the 5-digit document number; higher letters indicate later revisions.

Note the following details of the code protection feature on Microchip devices:

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

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona and Mountain View, California in October 2003. The Company's quality system processes and procedures are for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

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

Atlanta
3780 Mansell Road, Suite 130
Alpharetta, GA 30022
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

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, IN 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

San Jose
1300 Terra Bella Avenue
Mountain View, CA 94043
Tel: 650-215-1444
Fax: 650-961-0286

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

ASIA/PACIFIC

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

China - Beijing
Unit 706B
Wan Tai Bei Hai Bldg.
No. 6 Chaoyangmen Bei Str.
Beijing, 100027, China
Tel: 86-10-85282100
Fax: 86-10-85282104

China - Chengdu
Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200
Fax: 86-28-86766599

China - Fuzhou
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506
Fax: 86-591-7503521

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

China - Shanghai
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
Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel: 86-755-82901380
Fax: 86-755-8295-1393

China - Shunde
Room 401, Hongjian Building, No. 2
Fengxiangnan Road, Ronggui Town, Shunde
District, Foshan City, Guangdong 528303, China
Tel: 86-757-28395507 Fax: 86-757-28395571

China - Qingdao
Rm. B505A, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel: 86-532-5027355 Fax: 86-532-5027205

India
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-22290061 Fax: 91-80-22290062

Japan
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
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5932 or
82-2-558-5934

Singapore
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan
Kaohsiung Branch
30F - 1 No. 8
Min Chuan 2nd Road
Kaohsiung 806, Taiwan
Tel: 886-7-536-4818
Fax: 886-7-536-4803

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

EUROPE

Austria
Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

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

France
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
Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy
Via Quasimodo, 12
20025 Legnano (MI)
Milan, Italy
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands
P. A. De Biesbosch 14
NL-5152 SC Drunen, Netherlands
Tel: 31-416-690399
Fax: 31-416-690340

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

02/17/04