10 Versuch Nr. 8

10.1 Anmerkungen zum Versuch Nr. 8

Während der letzten 4 Versuche haben Sie sich mit dem detaillierten Rechner-Entwurf beschäftigt. Im letzten Versuch konnten Sie abschließend einen kleinen Rechner entwerfen, der grundlegende Funktionen, wie Addition, Subtraktion und Multiplikation durchführen konnte.

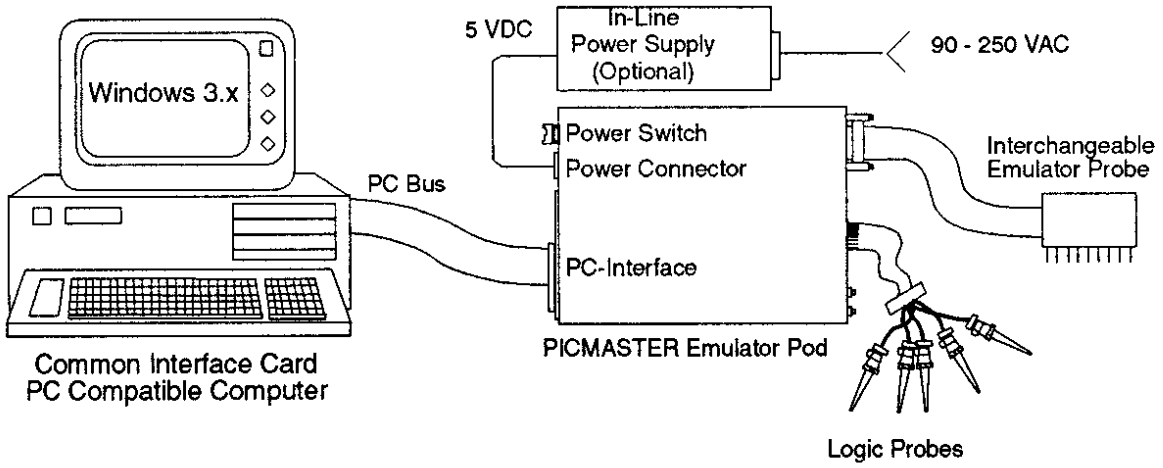
Um nun einen tieferen Einblick in den Rechneraufbau zu bekommen, werden Sie sich in den letzten drei Versuchen des Praktikums mit der Programmierung eines Mikroprozessors und dem Umgang der entsprechenden Entwicklungsumgebung beschäftigen. Der Mikroprozessor, der in diesen Versuchen behandelt wird, ist der PIC17C43. Im Versuch 8 werden Sie reine Softwareprogramme schreiben, in den Versuchen 9 und 10 müssen Sie eine Hardwareanbindung realisieren. Im Anhang von diesem Kapitel befindet sich eine Beschreibung des PIC17C43 und der Entwicklungsumgebung (Software, Emulator u.s.w.).

10.1.1 Einführung in die Entwicklung von Programmen und Hardware mit dem PIC17C4X

Die Programme, die ein Mikroprozessor ausführen soll, werden normalerweise in das interne ROM des Mikroprozessors gebrannt, welches zumeist nur einmal beschreibbar ist. Um nun in der Entwicklungsphase nicht mehrere Bausteine brennen zu müssen - die man dann ja nicht mehr verwenden könnte - steht dem Anwender Soft- und Hardware zur Entwicklung zur Verfügung, wie unten in der Abbildung zu sehen ist.

Auf dem PC entwickelt der Anwender die `Assemblerprogramme`, in Form eines ASCII-Files. Am PC ist ein sogenannter `Emulator-Pod` und `eine Probe` angeschlossen, die den eigentlichen Prozessor ersetzen, und diesen emulieren. An der Emulator-Probe ist die richtige Schaltung, in welche der Prozessor normalerweise eingesetzt wird, über ein Kabel angeschlossen. Das Kabel ist genau an der Stelle angeschlossen, wo der echte Prozessor sitzen soll.

Per Software (`MPLAB`) auf dem PC kann man nun seine Assembler-Programme übersetzen und schließlich testen !



10.2 Literaturhinweis zu Versuch Nr. 8

- Microchip Data Book, 1994
- Microchip Embedded Control Book, 1995
- Michael Rose: Mikroprozessor PIC17C42, Architektur und Applikation

10.3 Aufgabenstellung vor Versuchsdurchführung

Lesen Sie sich genau den Anhang zu diesem Versuch durch. Das Selbststudium des Anhangs ist zwingend notwendig zur Durchführung der Aufgaben.

10.3.1

Bereiten Sie ein Additionsprogramm für zwei 16-Bit-Zahlen vor. Die `Summanden` sollen sich in zwei `aufeinanderfolgenden Registern` ihrer Wahl befinden. Das Ergebnis soll an `Port C` und `D` ausgegeben werden. Gehen Sie dabei wie folgt vor:

- Skizzieren Sie ein Flussdiagramm, dem zu entnehmen ist, wie das Programm ablaufen soll.
- Schreiben Sie Ihr Additionsprogramm unter der Verwendung des Befehlssatzes aus Anhang 10.5.2. Dokumentieren Sie die Funktionalität Ihres Programms ausreichend!

10.3.2

Bereiten Sie ein Multiplikationsprogramm für zwei 8-Bit-Zahlen vor. Die `Multiplikanden` sollen sich in zwei `aufeinanderfolgenden Registern` ihrer Wahl befinden. Das Ergebnis soll an `Port C` und `D` ausgegeben werden. Gehen Sie dabei wie folgt vor:

- Skizzieren Sie ein Flussdiagramm, dem zu entnehmen ist, wie das Programm ablaufen soll.
- Schreiben Sie Ihr Multiplikationsprogramm unter der Verwendung des Befehlssatzes aus Anhang 10.5.2. Verwenden Sie jedoch nicht den vorhandenen Multiplizier-Befehl. Dokumentieren Sie die Funktionalität Ihres Programms ausreichend!

10.4 Versuchsdurchführung

ACHTUNG ! ACHTUNG ! ACHTUNG ! ACHTUNG ! ACHTUNG ! ACHTUNG !
VOR VERSUCHSDURCHFÜHRUNG BEACHTEN !!!!!!!

Bei der Entwicklung und dem Testen der Anwenderschaltungen und der Programme beachten Sie bitte folgendes zur Stromversorgung der einzelnen Komponenten:

Einschalten der Stromversorgung:

- Schalten Sie zunächste Ihren PC ein.
- Schalten Sie dann den Emualtor-Pod auf dessen Rückseite auf externe Versorgung ein.
- Schalten Sie dann die Anwenderschaltung ein.
- Starten Sie die entsprechende Software.

Ausschalten der Stromversorgung:

Das Ausschalten der einzelnen Komponenten geschieht in umgekehrter Reihenfolge:

- Verlassen der Software.
- Ausschalten der Anwenderschaltung.
- Ausschalten des Emulator-Pods.
- Ausschalten des PCs.

Gehen Sie bitte auch achtsam mit der offenliegenden Emulator-Probe um !

ACHTUNG ! ACHTUNG ! ACHTUNG ! ACHTUNG ! ACHTUNG ! ACHTUNG !
VOR VERSUCHSDURCHFÜHRUNG BEACHTEN !!!!!!!

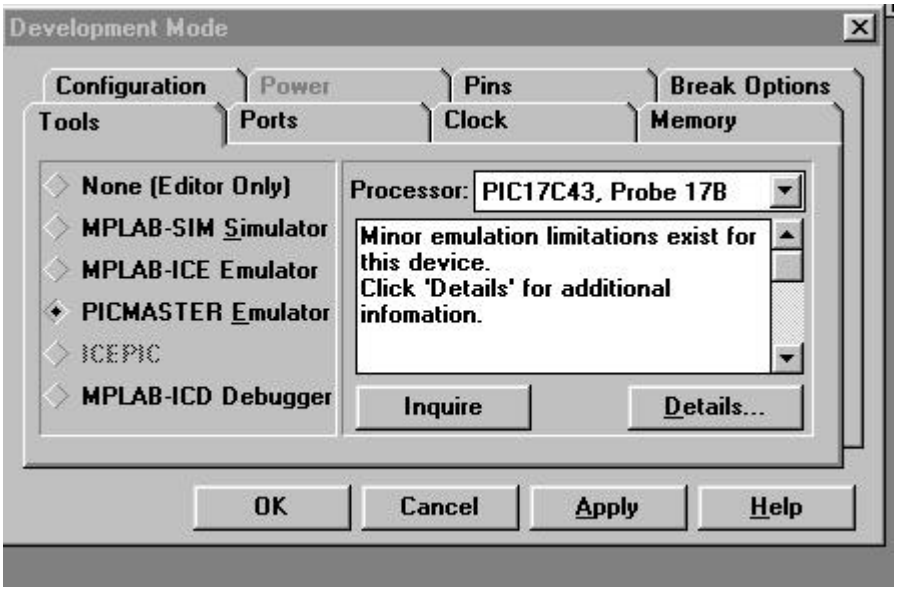
Verwendete Geräte: 1 PC inkl. Assembler

10.4.1

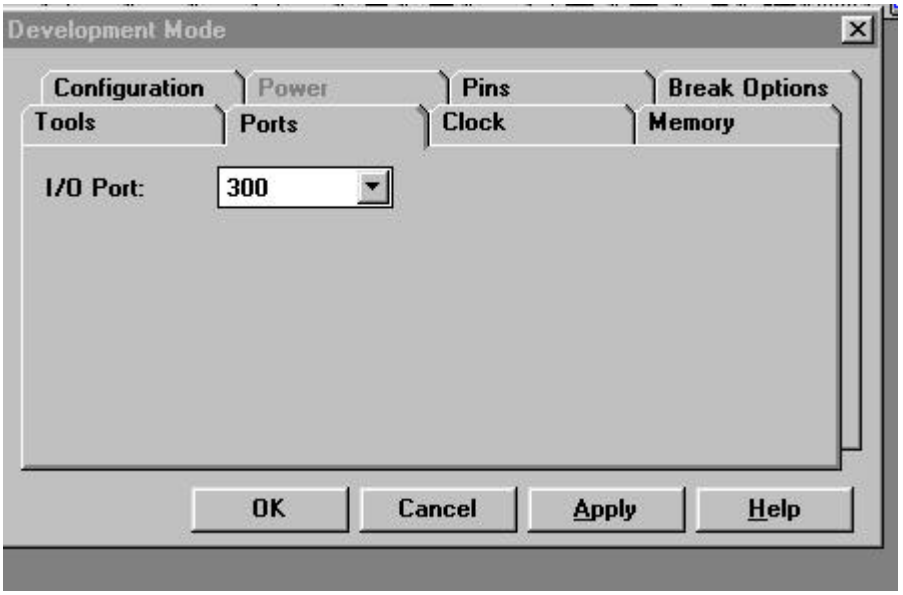
- Arbeiten Sie zunächst das Tutorial für die Software MPLAB aus Anhang 10.5.6 durch, um ein besseres Verständnis für die Entwicklungsumgebung zu bekommen.

Achten Sie dabei auf folgendes:

- Bei der Einstellung des `Development-Modes` (`Menü Options -> Development Mode`) sollte das folgende eingestellt sein:



und



- Ihre Dateien sollten in folgendem Verzeichnis abgespeichert werden: `c:\labor\versuch8`

10.4.2

- Geben Sie nun Ihr entworfenes Additionsprogramm aus 10.3.1 wie im Tutorial beschrieben mit der Software MPLAB ein. Dokumentieren Sie Ihr Assembler-File ausreichend. Speichern Sie ihre Assembler-File unter `c:\labor\versuch8\add.asm ab.`

- Lassen Sie Ihr Programm laufen (`Run`) bzw. Steppen Sie Schritt für Schritt jede Befehlszeile Ihres Programms durch, damit Sie nachvollziehen können, was im Inneren des Prozessor geschieht. Betrachten Sie sich dazu die folgenden Fenster: `Program Memory Window, Special Function Register Window` und `File Register Window.`

- Geforderte Dokumentation:

  - Ausdruck des Assembler-Files.
  - Ausdruck der folgenden Fenster nach erfolgreicher Ausführung des Programms: `Program Memory Window, Special Function Register Window` und `File Register Window.`

10.4.3

- Geben Sie nun Ihr entworfenes Multiplikationssprogramm aus 10.3.2 wie im Tutorial beschrieben mit der Software MPLAB ein. Dokumentieren Sie Ihr Assembler-File ausreichend. Speichern Sie ihre Assembler-File unter `c:\labor\versuch8\mul.asm ab.`

- Lassen Sie Ihr Programm laufen (`Run`) bzw. Steppen Sie Schritt für Schritt jede Befehlszeile Ihres Programms durch, damit Sie nachvollziehen können, was im Inneren des Prozessor geschieht. Betrachten Sie sich dazu die folgenden Fenster: `Program Memory Window, Special Function Register Window` und `File Register Window.`

- Geforderte Dokumentation:

  - Ausdruck des Assembler-Files.
  - Ausdruck der folgenden Fenster nach erfolgreicher Ausführung des Programms: `Program Memory Window, Special Function Register Window` und `File Register Window.`

10.5 Anhang zu Versuch Nr. 8

10.5.1 Datenblattauszug des PIC17C4X

MICROCHIP

PIC17C4X

High-Performance 8-Bit CMOS EPROM Microcontroller

Devices Included In This Data Sheet

- PIC17C42
- PIC17C43
- PIC17C44

High-Performance RISC-like CPU Features

- Only 58 single word instructions to learn
- All single cycle instructions (160 ns) except for program branches which are two-cycle and table reads/writes
- Operating speed:
  - DC - 25 MHz clock input
  - DC - 160 ns instruction cycle

Device	Program Memory	Data Memory
PIC17C44	8K	454
PIC17C43	4K	454
PIC17C42	2K	232

- 8 x 8 Hardware Multiplier (PIC17C43/C44 Devices only)
- Interrupt capability
- 16 levels deep hardware stack
- Direct, indirect and relative addressing modes
- Internal/External program memory execution
- 64K x 16 addressable program memory space

Peripheral Features

- 33 I/O pins with individual direction control
- High current sink/source for direct LED drive
  - RA2 and RA3 are open collector, high voltage (12V), high current (60 mA), I/O
- Two capture inputs and two PWM outputs
  - Captures are 16-bit, max resolution 160 ns
  - PWM resolution is 1- to 10-bit
- TMR0: 16-bit timer/counter with 8-bit programmable prescaler
- TMR1: 8-bit timer/counter
- TMR2: 8-bit timer/counter
- TMR3: 16-bit timer/counter
- Serial Communications Interface (SCI/USART)

Special microcontroller features

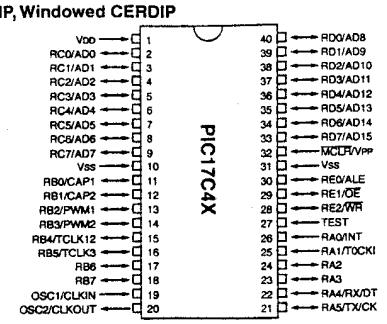
- Power-On Reset (POR), Power-Up Timer (PWRT) and Oscillator Start-Up Timer (OST)
- Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation
- Code-protection
- Power saving SLEEP mode
- Selectable oscillator options

CMOS Technology

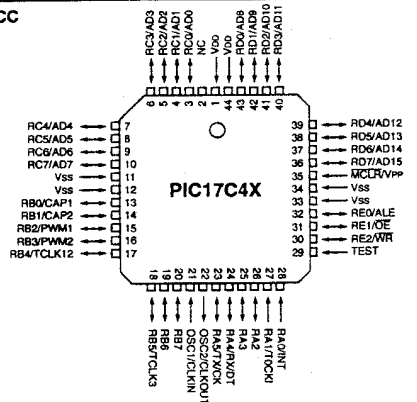
- Low-power, high-speed CMOS EPROM technology
- Fully static design
- Wide operating voltage range (2.5V to 6.0V)
- Commercial and Industrial Temperature Range
- Low-power consumption
  - < 5 mA @ 5V, 4 MHz
  - 100 µA typical @ 4.5V, 32 kHz
  - < 1 µA typical standby current @ 5V

PACKAGE TYPES

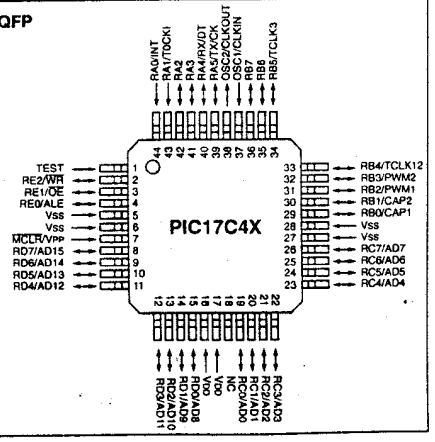
PDIP, Windowed CERDIP



PLCC



QFP



1.0 OVERVIEW

This data sheet covers the PIC17C4X group of the PIC17CXX family of microcontrollers. The following devices are discussed in this data sheet:

- PIC17C42
- PIC17C43
- PIC17C44

The PIC17C43 and PIC17C44 devices include architectural enhancements over the PIC17C42. These enhancements will be discussed throughout this data sheet.

The PIC17C4X devices are 40-Pin, EPROM-based members of the versatile PIC17CXX family of low-cost, high-performance, CMOS, fully-static, 8-bit microcontrollers.

All PIC16/17 microcontrollers employ an advanced RISC-like architecture. The PIC17CXX has enhanced core features, sixteen-level deep stack, and multiple internal and external interrupt sources. The separate instruction and data buses of the Harvard architecture allow a 16-bit wide instruction word with a separate 8-bit wide data. The two stage instruction pipeline allows all instructions to execute in a single cycle, except for program branches (which require two cycles). A total of 55 instructions (reduced instruction set) are available in the PIC17C42 and 58 instructions in the PIC17C43 and PIC17C44 devices. Additionally, a large register set gives some of the architectural innovations used to achieve a very high performance. For mathematical intensive applications the PIC17C43 and PIC17C44 devices have a single cycle 8 x 8 Hardware Multiplier.

PIC17CXX microcontrollers typically achieve a 2:1 code compression and a 4:1 speed improvement over other 8-bit microcontrollers in their class.

The PIC17C4X devices have up to 454 bytes of RAM and 33 I/O pins. In addition, the PIC17C4X adds several peripheral features useful in many high performance applications including:

- Four timer/counters
- Two capture inputs
- Two PWM outputs
- A Serial Communications Interface (SCI)

The SCI can be configured for either synchronous or asynchronous communications (USART).

These special features reduce external components, thus reducing cost, enhancing system reliability and reducing power consumption. There are four oscillator options, of which the single pin RC oscillator provides a low-cost solution, the LF oscillator is for low frequency crystals and minimizes power consumption, XT is a standard crystal, and the EC is for external clock input. The SLEEP (power-down) mode offers additional power saving. The user can wake up the chip from SLEEP through several external and internal interrupts and device resets.

There are four configuration options for the device operational modes:

- Microprocessor
- Microcontroller
- Extended microcontroller
- Protected microcontroller

The microprocessor and extended microcontroller modes allow up to 64K-words of external program memory.

A highly reliable Watchdog Timer with its own on-chip RC oscillator provides protection against software malfunction.

Table 1-1 lists the features of the PIC17C4X devices.

A UV-erasable CERDIP-packaged version is ideal for code development while the cost-effective One-Time Programmable (OTP) version is suitable for production in any volume.

A simplified block diagram of the PIC17C42 is shown in Figure 3-1 and the block diagram for the PIC17C43 and PIC17C44 devices is shown in Figure 3-2.

The PIC17C4X fits perfectly in applications ranging from precise motor control and industrial process control to automotive, instrumentation, and telecom applications. Other applications that require extremely fast execution of complex software programs or the flexibility of programming the software code as one of the last steps of the manufacturing process would also be well suited. The EPROM technology makes customization of application programs (with unique security codes, combinations, model numbers, parameter storage, etc.) fast and convenient. Small footprint package options make the PIC17C4X ideal for applications with space limitations that require high performance. High speed execution, powerful peripheral features, flexible I/O, and low power consumption all at low cost make the PIC17C4X ideal for a wide range of embedded control applications.

1.1 Family and Upward Compatibility

Those users familiar with the PIC16C5x and PIC16Cxx families of microcontrollers will see the architectural enhancements that have been implemented. These enhancements allow the device to be more efficient in software and hardware requirements. Please refer to Appendix A for a detailed list of enhancements and modifications. Code written for PIC16C5X or PIC16CXX can be easily ported to PIC17CXX family of devices (see Appendix B).

1.2 Development Support

The PIC17CXX family is supported by a full-featured macro assembler, a software simulator, an in-circuit emulator, a universal programmer, a "C" compiler, and fuzzy logic support tools.

TABLE 1-1: PIC17CXX FAMILY OF DEVICES

		PIC17C42	PIC17C43	PIC17C44
Maximum Frequency of Operation		25 MHz	25 MHz	25 MHz
Operating Voltage Range		4.5 - 5.5V	2.5 - 6.0V	2.5 - 6.0V
On-chip Program Memory (16-bits wide)		2K	4K	8K
Data Memory (bytes)		232	454	454
Hardware Multiplier (8 x 8)		No	Yes	Yes
Timer0 (16-bit + 8-bit postscaler)		Yes	Yes	Yes
Timer1 (8-bit)		Yes	Yes	Yes
Timer2 (8-bit)		Yes	Yes	Yes
Timer3 (16-bit)		Yes	Yes	Yes
Capture inputs (16-bit)		2	2	2
PWM outputs (up to 10-bit)		2	2	2
Serial Communications Interface (SCI/USART)		Yes	Yes	Yes
Power-On Reset		Yes	Yes	Yes
Watchdog Timer		Yes	Yes	Yes
External Interrupts		Yes	Yes	Yes
Interrupt Sources		11	11	11
Program Memory Code Protect		Yes	Yes [1]	Yes [1]
I/O		33	33	33
I/O High Current Capability	Source	25 mA	25 mA	25 mA
	Sink	25 mA[2]	25 mA[2]	25 mA[2]
Package Types		40-Pin DIP, 44-pin PLCC 44-pin MQFP	40-Pin DIP, 44-pin PLCC 44-pin TQFP	40-Pin DIP, 44-pin PLCC 44-pin TQFP

Note 1: The Code Protect Feature is different from the PIC17C42.
2: RA2 and RA3 can sink up to 60 mA.

3.0 ARCHITECTURAL OVERVIEW

The high performance of the PIC17C4X can be attributed to a number of architectural features commonly found in RISC microprocessors. To begin with, the PIC17C4X uses a modified Harvard architecture. This architecture has the program and data accessed from separate memories. So the device has a program memory bus and a data memory bus. This improves bandwidth over traditional von Neumann architecture, where program and data are fetched from the same memory (accesses over the same bus). Separating program and data memory further allows instructions to be sized differently than the 8-bit wide data word. PIC17C4X opcodes are 16-bits wide, enabling single word instructions. The full 16-bit wide program memory bus fetches a 16-bit instruction in a single cycle. A two-stage pipeline overlaps fetch and execution of instructions. Consequently, all instructions execute in a single cycle (160 ns @ 25 MHz), except for program branches and two special instructions that transfer data between program and data memory.

The PIC17C4X can address up to 64K x 16 of program memory space. The PIC17C42 integrates 2K x 16 EPROM program memory on-chip, while the PIC17C43 integrates 4K x 16 and the PIC17C44 integrates 8K x 16 EPROM program memory. Program execution can be internal only (microcontroller or protected mirco-controller mode), external only (microprocessor mode) or both (extended microcontroller mode).

The PIC17CXX can directly or indirectly address its register files or data memory. All special function registers, including the Program Counter (PC) and Working Register (WREG), are mapped in the data memory. The PIC17CXX has an orthogonal (symmetrical) instruction set that makes it possible to carry out any operation on any register using any addressing mode. This symmetrical nature and lack of 'special optimal situations' make programming with the PIC17CXX simple yet efficient. In addition, the learning curve is reduced significantly.

One of the PIC17CXX family architectural enhancements from the PIC16CXX family allows two file registers to be used in some two operand instructions. This allows data to be moved directly between two registers without going through the WREG register. This increases performance and decreases program memory usage.

The PIC17CXX devices contain an 8-bit ALU and working register. The ALU is a general purpose arithmetic unit. It performs arithmetic and Boolean functions between data in the working register and any register file.

The ALU is 8-bits wide and capable of addition, subtraction, shift and logical operations. Unless otherwise mentioned, arithmetic operations are two's complement in nature.

The WREG register is an 8-bit working register used for ALU operations.

The PIC17C43 and PIC17C44 devices also have an 8 x 8 hardware multiplier. This multiplier generates a 16-bit result in a single cycle.

Depending on the instruction executed, the ALU may affect the values of the Carry (C), Digit Carry (DC), and Zero (Z) bits in the STATUS register. The C and DC bits operate as a $\overline{\text{borrow}}$ and $\overline{\text{digit borrow}}$ out bit, respectively, in subtraction. See the `SUBLW` and `SUBWF` instructions for examples.

Although the ALU does not perform signed arithmetic, the Overflow bit (OV) can be used to implement signed math. Signed arithmetic is comprised of a magnitude and a sign bit. The overflow bit indicates if the magnitude overflows and causes the sign bit to change state. Signed math can have greater than 7-bit values (magnitude), if more than one byte is used. The use of the overflow bit only operates on bit6 (MSb of magnitude) and bit7 (sign bit) of the value in the ALU. That is, the overflow bit is not useful if trying to implement signed math where the magnitude, for example, is 11-bits. If the signed math values are greater than 7-bits (15-, 24- or 31-bit), the algorithm must ensure that the low order bytes ignore the overflow status bit.

Care should be taken when adding and subtracting signed numbers to ensure that the correct operation is executed. Example 3-1 shows an item that must be taken into account when doing signed arithmetic on an ALU which operates as an unsigned machine.

EXAMPLE 3-1: SIGNED MATH

```
Signed Math          Unsigned Math
  -127   (FFh)          255   (FFh)
+    1   (01h)        +   1   (01h)
= -126   (FEh)        =   0   (00h) ; C = 1
```

Signed math requires the result in REG to be FEh (-126). This would be accomplished by subtracting one as opposed to adding one.

Simplified block diagrams are shown in Figure 3-1 and Figure 3-2. The descriptions of the device pins are listed in Table 3-1.

FIGURE 3-2: PIC17C43 AND PIC17C44 BLOCK DIAGRAM

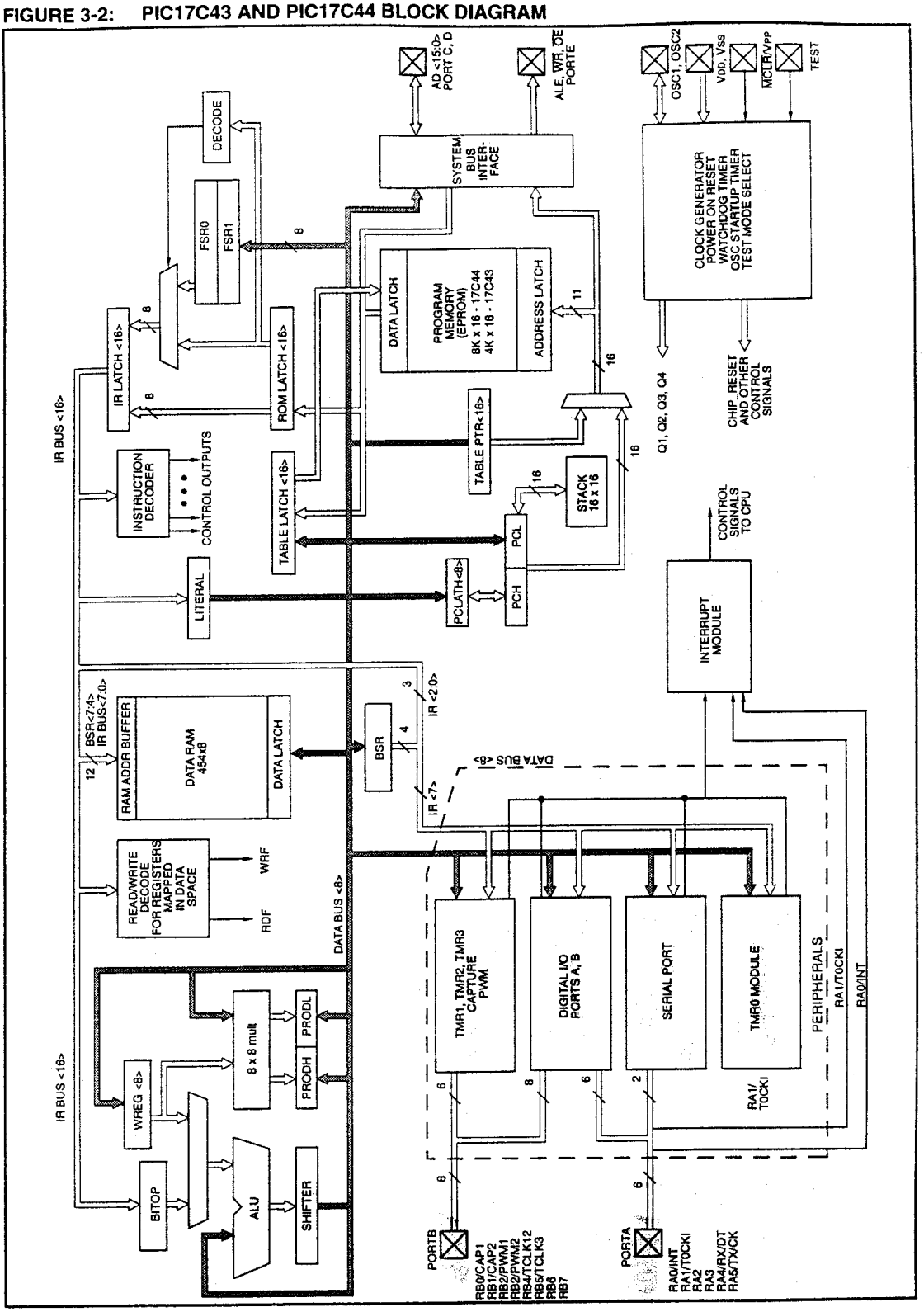


TABLE 3-1: PIC17C4X PINOUT DESCRIPTIONS

Name	DIP No.	PLCC No.	QFP No.	I/O/P Type	Buffer Type	Description
OSC1/CLKIN	19	21	37	I	ST	Oscillator input in crystal/resonator or RC oscillator mode. External clock input in external clock mode.
OSC2/CLKOUT	20	22	38	O	—	Oscillator output. Connects to crystal or resonator in crystal oscillator mode. In RC oscillator or external clock modes OSC2 pin outputs CLKOUT which has one fourth the frequency of OSC1 and denotes the instruction cycle rate.
$\overline{MCLR}$/VPP	32	35	7	I/P	ST	Master clear (reset) input/Programming Voltage (VPP) input. This is the active low reset input to the chip.
						PORTA is a bidirectional I/O Port except for RA0 and RA1 which are input only.
RA0/INT	26	28	44	I	ST	RA0/INT can also be selected as an external interrupt input. Interrupt can be configured to be on positive or negative edge.
RA1/T0CKI	25	27	43	I	ST	RA1/T0CKI can also be selected as an external interrupt input, and the interrupt can be configured to be on positive or negative edge. RA1T0CKI can also be selected to be the clock input to the TMR0 timer/counter.
RA2	24	26	42	I/O	ST	High voltage, high current open collector input/output port pins.
RA3	23	25	41	I/O	ST	High voltage, high current open collector input/output port pins.
RA4/RX/DT	22	24	40	I/O	ST	RA4/RX/DT can also be selected as the SCI Asynchronous Receive or SCI Synchronous Data.
RA5/TX/CK	21	23	39	I/O	ST	RA5/TX/CK can also be selected as the SCI Asynchronous Transmit or SCI Synchronous Clock.
						PORTB is a bidirectional I/O Port.
RB0/CAP1	11	13	29	I/O	ST	RB0/CAP1 can also be the CAP1 input pin.
RB1/CAP2	12	14	30	I/O	ST	RB1/CAP2 can also be the CAP2 input pin.
RB2/PWM1	13	15	31	I/O	ST	RB2/PWM1 can also be the PWM1 output pin.
RB3/PWM2	14	16	32	I/O	ST	RB3/PWM2 can also be the PWM2 output pin.
RB4/TCLK12	15	17	33	I/O	ST	RB4/TCLK12 can also be the external clock input to Timer1 and Timer2.
RB5/TCLK3	16	18	34	I/O	ST	RB5/TCLK3 can also be the external clock input to Timer3.
RB6	17	19	35	I/O	ST	
RB7	18	20	36	I/O	ST	
						PORTC is a bidirectional I/O Port.
RC0/AD0	2	3	19	I/O	TTL	This is also the lower half of the 16 bit wide system bus in microprocessor mode or extended microcontroller mode. In multiplexed system bus configuration, these pins are address output as well as data input or output.
RC1/AD1	3	4	20	I/O	TTL	
RC2/AD2	4	5	21	I/O	TTL	
RC3/AD3	5	6	22	I/O	TTL	
RC4/AD4	6	7	23	I/O	TTL	
RC5/AD5	7	8	24	I/O	TTL	
RC6/AD6	8	9	25	I/O	TTL	
RC7/AD7	9	10	26	I/O	TTL	

Legend: I = Input only; O = Output only; I/O = Input/Output; P = Power; — = Not Used; TTL = TTL input; ST = Schmitt Trigger input.

TABLE 3-1: PIC17C4X PINOUT DESCRIPTIONS (CONT.)

Name	DIP No.	PLCC No.	QFP No.	I/O/P Type	Buffer Type	Description
						PORTD is a bidirectional I/O Port.
RD0/AD8	40	43	15	I/O	TTL	This is also the upper byte of the 16-bit system bus in microprocessor mode or extended microprocessor mode or extended microcontroller mode. In multiplexed system bus configuration these pins are address output as well as data input or output.
RD1/AD9	39	42	14	I/O	TTL	
RD2/AD10	38	41	13	I/O	TTL	
RD3/AD11	37	40	12	I/O	TTL	
RD4/AD12	36	39	11	I/O	TTL	
RD5/AD13	35	38	10	I/O	TTL	
RD6/AD14	34	37	9	I/O	TTL	
RD7/AD15	33	36	8	I/O	TTL	
						PORTE is a bidirectional I/O Port.
RE0/ALE	30	32	4	I/O	TTL	In microprocessor mode or extended microcontroller mode, it is the Address Latch Enable (ALE) output. Address should be latched on the falling edge of ALE output.
RE1/$\overline{OE}$	29	31	3	I/O	TTL	In microprocessor or extended microcontroller mode, it is the Output Enable ($\overline{OE}$) control output (active low).
RE2/$\overline{WR}$	28	30	2	I/O	TTL	In microprocessor or extended microcontroller mode, it is the Write Enable ($\overline{WR}$) control output (active low).
TEST	27	29	1	I	ST	Test mode selection control input. Always tie to VSS for normal operation.
VSS	10, 31	11, 12, 33, 34	5, 6, 27, 28	P		Ground reference for logic and I/O pins.
VDD	1	1, 44	16, 17	P		Positive supply for logic and I/O pins.

3.1 Clocking Scheme/Instruction Cycle

The clock input (from OSC1) is internally divided by four to generate four non-overlapping quadrature clocks, namely Q1, Q2, Q3 and Q4. Internally, the program counter (PC) is incremented every Q1, and the instruction is fetched from the program memory and latched into the instruction register in Q4. The instruction is decoded and executed during the following Q1 through Q4. The clocks and instruction execution flow are shown in Figure 3-3.

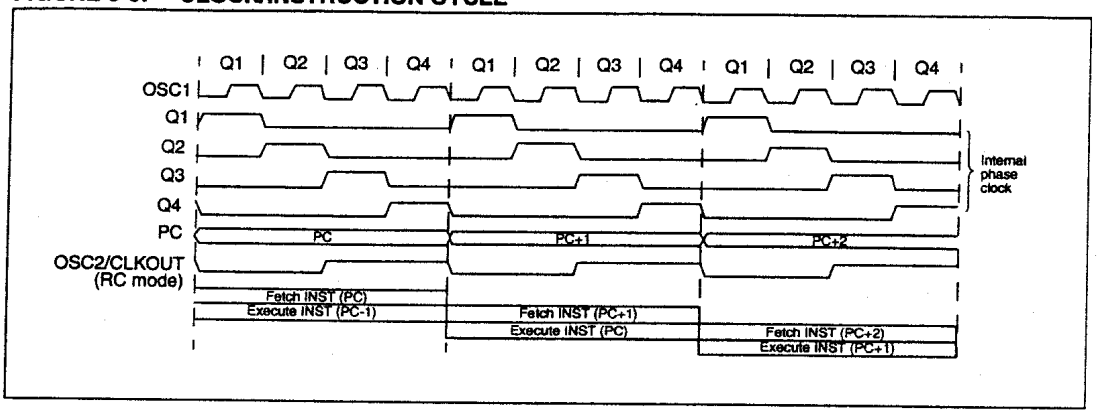
3.2 Instruction Flow/Pipelining

An "Instruction Cycle" consists of four Q cycles (Q1, Q2, Q3 and Q4). The instruction fetch and execute are pipelined such that fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to the pipelining, each instruction effectively executes in one cycle. If an instruction causes the program counter to change (e.g. GOTO) then two cycles are required to complete the instruction (see Example 3-2).

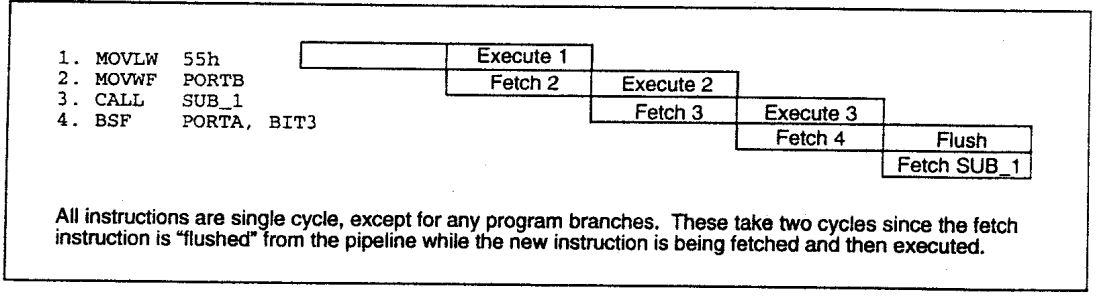
A fetch cycle begins with the program counter incrementing in Q1.

In the execution cycle, the fetched instruction is latched into the "Instruction Register (IR)" in cycle Q1. This instruction is then decoded and executed during the Q2, Q3, and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write).

FIGURE 3-3: CLOCK/INSTRUCTION CYCLE



EXAMPLE 3-2: INSTRUCTION PIPELINE FLOW



All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is "flushed" from the pipeline while the new instruction is being fetched and then executed.

4.0 RESET

The PIC17CXX differentiates between various kinds of reset:

- Power-On Reset (POR)
- $\overline{MCLR}$ reset during normal operation
- WDT time-out reset during normal operation

Some registers are not affected in any reset condition; their status is unknown on POR and unchanged in any other reset. Most other registers are reset to a "reset state" on Power-On Reset (POR), on $\overline{MCLR}$ or WDT reset during normal operation and on $\overline{MCLR}$ reset during SLEEP. They are not affected by a WDT reset during SLEEP, since this reset is viewed as the resumption of normal operation. The $\overline{TO}$ and $\overline{PD}$ bits are set or cleared differently in different reset situations as indicated in Table 4-3. These bits are used in software to determine the nature of reset. See Table 4-4 for a full description of reset states of all registers.

4.1 Power-On Reset (POR), Power-Up-Timer (PWRT) and Oscillator Start-Up Timer (OST)

4.1.1 POWER-ON RESET (POR)

The Power-On Reset circuit holds the device in reset until V_{DD} is above the trip point (in the range of 1.4V - 2.3V). PIC17C43 and PIC17C44 will produce an internal reset for both rising and falling V_{DD}. The PIC17C42 does not produce an internal reset when V_{DD} declines. To take advantage of the POR, just tie the $\overline{MCLR}/V_{PP}$ pin directly (or through a resistor) to V_{DD}. This will eliminate external RC components usually needed to create Power-On Reset. A minimum rise time for V_{DD} is required. See Electrical Specifications for details.

4.1.2 POWER-UP TIMER (PWRT)

The Power-Up Timer provides a fixed 96 ms time-out (nominal) on power-up. This occurs from rising edge of the POR signal and after the first rising edge of $\overline{MCLR}$ (detected high). The power-up timer operates on an internal RC oscillator. The chip is kept in RESET as long as the PWRT is active. In most cases the PWRT delay allows the V_{DD} to rise to an acceptable level.

The power-up time delay will vary from chip to chip and to V_{DD} and temperature. See DC parameters for details.

4.1.3 OSCILLATOR START-UP TIMER (OST)

The Oscillator Start-Up Timer (OST) provides a 1024 oscillator cycle (1024 Tosc) delay after $\overline{MCLR}$ is detected high or a wake-up from SLEEP event occurs.

The OST time-out is invoked only for XT and LF oscillator modes on a Power-On Reset or a Wake-Up from SLEEP.

The OST counts the oscillator pulses on the OSC1/CLKIN pin. The counter only starts incrementing after the amplitude of the signal reaches the oscillator input thresholds. This delay allows the crystal oscillator or resonator to stabilized before the device exits reset. The length of time-out is a function of the crystal/resonator frequency.

4.1.4 TIME-OUT SEQUENCE

On power-up the time-out sequence is as follows: First the internal POR signal goes high when the POR trip point is reached. If $\overline{MCLR}$ is high, then both the OST and PWRT timers start. In general the PWRT time-out is longer, except with low frequency crystals/resonators. The total time-out also varies based on oscillator configuration. Table 4-1 shows the times that are associated with the oscillator configuration. Figure 4-2 and Figure 4-3 display these time-out sequences.

If the device voltage is not within electrical specification at the end of a time-out, the $\overline{MCLR}/V_{PP}$ pin must be held low until the voltage is within the device specification. The use of an external RC delay is sufficient for many of these applications.

The time-out sequence begins from the first rising edge of $\overline{MCLR}$.

TABLE 4-1: TIME-OUT IN VARIOUS SITUATIONS

Oscillator Configuration	Power-up	Wake up from SLEEP	$\overline{MCLR}$ Reset
XT, LF	Greater of (96 ms, 1024 Tosc)	1024 Tosc	—
EC, RC	Greater of (96 ms, 1024 Tosc)	—	—

TABLE 4-2: STATUS BITS AND THEIR SIGNIFICANCE

$\overline{TO}$	$\overline{PD}$	Event
1	1	Power-On Reset, $\overline{MCLR}$ reset during normal operation, or CLRWDT instruction executed
1	0	$\overline{MCLR}$ reset during SLEEP or interrupt wake-up from SLEEP
0	1	WDT reset during normal operation
0	0	WDT time-out wake-up from SLEEP

In Figure 4-2, Figure 4-3 and Figure 4-4, $T_{PWRT} > T_{OST}$, as would be the case in higher frequency crystals. For lower frequency crystals, (i.e., 32 kHz) T_{OST} would be greater.

Table 4-3 shows the reset conditions for some special registers, while Table 4-4 shows the initialization conditions for all the registers. The shaded registers (in Table 4-4) only exist for PIC17C43 and PIC17C44 devices. In the PIC17C42, the PRODH and PRODL registers are general purpose RAM.

TABLE 4-3: RESET CONDITION FOR THE PROGRAM COUNTER AND THE CPUSTA REGISTER

Event		PCH:PCL	CPUSTA	OST Active
Power-On Reset		0000h	--11 11--	Yes
$\overline{MCLR}$ reset during normal operation		0000h	--11 11--	No
$\overline{MCLR}$ reset during SLEEP		0000h	--11 10--	Yes [2]
WDT reset during normal operation		0000h	--11 01--	No
WDT during SLEEP		0000h	--11 00--	Yes [2]
Interrupt wake-up from SLEEP	GLINTD is set	PC + 1	--11 10--	Yes [2]
	GLINTD is clear	PC + 1 [1]	--10 10--	Yes [2]

Legend: u = unchanged, x = unknown, - = unimplemented, reads as '0'.
Note 1: On wake-up, this instruction is executed. The instruction at the appropriate interrupt vector is fetched and then executed.
2: The OST is only active when the Oscillator is configured for XT or LF modes.

FIGURE 4-2: TIME-OUT SEQUENCE ON POWER-UP ($\overline{MCLR}$ TIED TO VDD)

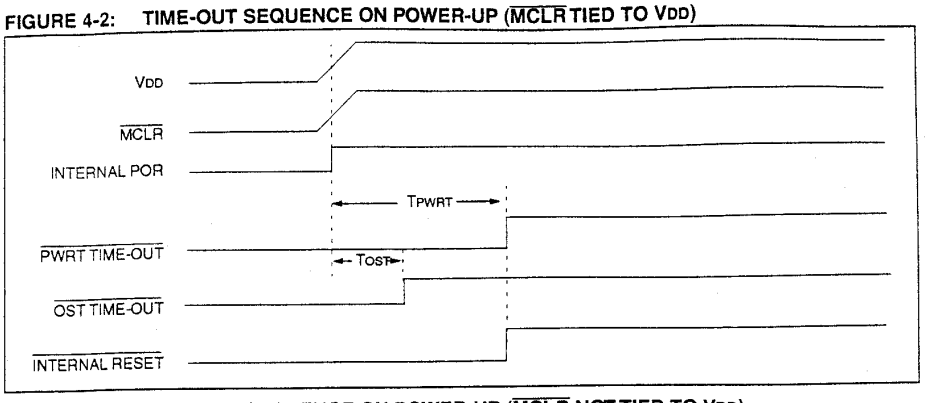


FIGURE 4-3: TIME-OUT SEQUENCE ON POWER-UP ($\overline{MCLR}$ NOT TIED TO VDD)

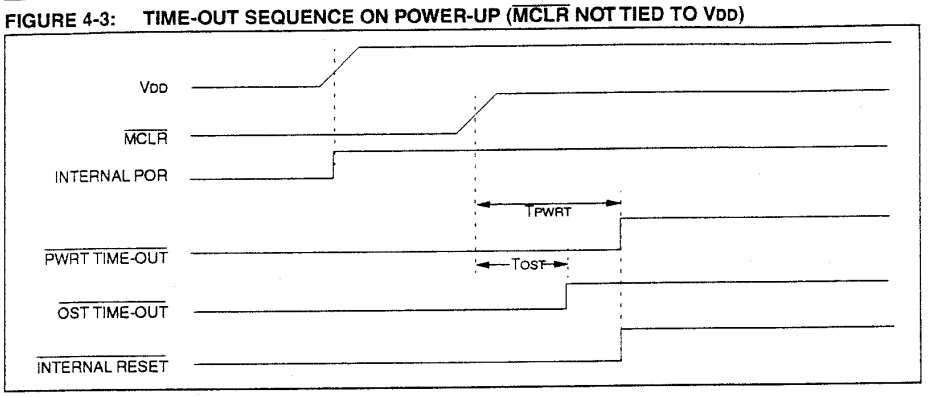


FIGURE 4-4: SLOW RISE TIME ($\overline{MCLR}$ TIED TO VDD)

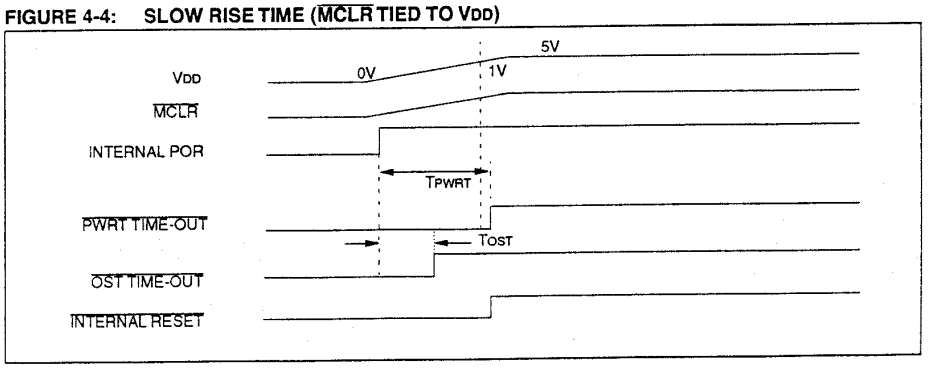


TABLE 4-4: INITIALIZATION CONDITIONS FOR SPECIAL FUNCTION REGISTERS (CONT.)

Register	Address	Power-On Reset	MCLR Reset WDT Reset	Wake up from SLEEP through interrupt
BANK 2				
TMR1	10h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TMR2	11h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TMR3L	12h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TMR3H	13h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PR1	14h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PR2	15h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PR3/CA1L	16h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PR3/CA1H	17h	xxxx xxxx	uuuu uuuu	uuuu uuuu
BANK 3				
PW1DCL	10h	xx-- ----	uu-- ----	uu-- ----
PW2DCL	11h	xx-- ----	uu-- ----	uu-- ----
PW1DCH	12h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PW2DCH	13h	xxxx xxxx	uuuu uuuu	uuuu uuuu
CA2L	14h	xxxx xxxx	uuuu uuuu	uuuu uuuu
CA2H	15h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TCON1	16h	0000 0000	0000 0000	uuuu uuuu
TCON2	17h	0000 0000	0000 0000	uuuu uuuu
UNBANKED				
PRODL[4]	18h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PRODH[4]	19h	xxxx xxxx	uuuu uuuu	uuuu uuuu

TABLE 4-4: INITIALIZATION CONDITIONS FOR SPECIAL FUNCTION REGISTERS

Register	Address	Power-On Reset	MCLR Reset WDT Reset	Wake up from SLEEP through interrupt
UNBANKED				
INDF0	00h	0000 0000	0000 0000	0000 0000
FSR0	01h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PCL	02h	0000h	0000h	PC + 1[2]
PCLATH	03h	xxxx xxxx	uuuu uuuu	uuuu uuuu
ALUSTA	04h	1111 xxxx	1111 uuuu	1111 uuuu
T0STA	05h	0000 000-	0000 000-	0000 000-
CPUSTA[3]	06h	--11 11--	--11 ??--	--uu ??--
INTSTA	07h	0000 0000	0000 0000	uuuu uuuu[1]
INDF1	08h	0000 0000	0000 0000	uuuu uuuu
FSR1	09h	xxxx xxxx	uuuu uuuu	uuuu uuuu
WREG	0Ah	xxxx xxxx	uuuu uuuu	uuuu uuuu
TMR0L	0Bh	xxxx xxxx	uuuu uuuu	uuuu uuuu
TMR0H	0Ch	xxxx xxxx	uuuu uuuu	uuuu uuuu
TBLPTRL	0Dh	xxxx xxxx	uuuu uuuu	uuuu uuuu
TBLPTRH	0Eh	xxxx xxxx	uuuu uuuu	uuuu uuuu
TBLPTRL[4]	0Dh	0000 0000	0000 0000	uuuu uuuu
TBLPTRH[4]	0Eh	0000 0000	0000 0000	uuuu uuuu
BSR	0Fh	0000 0000	0000 0000	uuuu uuuu
BANK 0				
PORTA	10h	0-xx xxxx	0-uu uuuu	uuuu uuuu
DDRB	11h	1111 1111	1111 1111	uuuu uuuu
PORTB	12h	xxxx xxxx	uuuu uuuu	uuuu uuuu
RCSTA	13h	0000 -00x	0000 -00u	uuuu -uuu
RCREG	14h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TXSTA	15h	0000 --1x	0000 --1u	uuuu --uu
TXREG	16h	xxxx xxxx	uuuu uuuu	uuuu uuuu
SPBRG	17h	xxxx xxxx	uuuu uuuu	uuuu uuuu
BANK 1				
DDRC	10h	1111 1111	1111 1111	uuuu uuuu
PORTC	11h	xxxx xxxx	uuuu uuuu	uuuu uuuu
DDRD	12h	1111 1111	1111 1111	uuuu uuuu
PORTD	13h	xxxx xxxx	uuuu uuuu	uuuu uuuu
DDRE	14h	---- -111	---- -111	---- -uuu
PORTE	15h	---- -xxx	---- -uuu	---- -uuu
PIR	16h	0000 0010	0000 0010	uuuu uuu[1]
PIE	17h	0000 0000	0000 0000	uuuu uuuu

Legend: u = unchanged, x = unknown, - = unimplemented, reads as '0', ? = value depends on condition

Note 1: One or more bits in INTSTA, PIR will be affected (to cause wake-up).
2: When the wake-up is due to an interrupt and the GLINTD bit is cleared, the PC is loaded with the interrupt vector.
3: See Table 4-3 for reset value of specific condition.
4: These values are for the PIC17C43 and PIC17C44 only.

5.0 INTERRUPTS

The PIC17C4X devices have 11 sources of interrupt:

- External interrupt from the RA0/INT pin
- Change on RB<7:0> pins
- Timer0 Overflow
- Timer1 Overflow
- Timer2 Overflow
- Timer3 Overflow
- SCI Transmit buffer empty
- SCI Receive buffer full
- Capture1
- Capture2
- T0CKI edge occurred

There are four registers used in the control and status of interrupts. These are:

- CPUSTA
- INTSTA
- PIE
- PIR

The CPUSTA register contains the GLINTD bit. This is the Global Interrupt Disable bit. When this bit is set, all interrupts are disabled. This bit is part of the controller core functionality and is described in the Memory Organization section.

When an interrupt is responded to, the GLINTD bit is automatically set to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with the interrupt vector address. There are four interrupt vectors. Each vector address is for a specific interrupt source (except the peripheral interrupts which have the same vector address). These sources are:

- External interrupt from the RA0/INT pin
- Timer0 Overflow
- T0CKI edge occurred
- Any peripheral interrupt

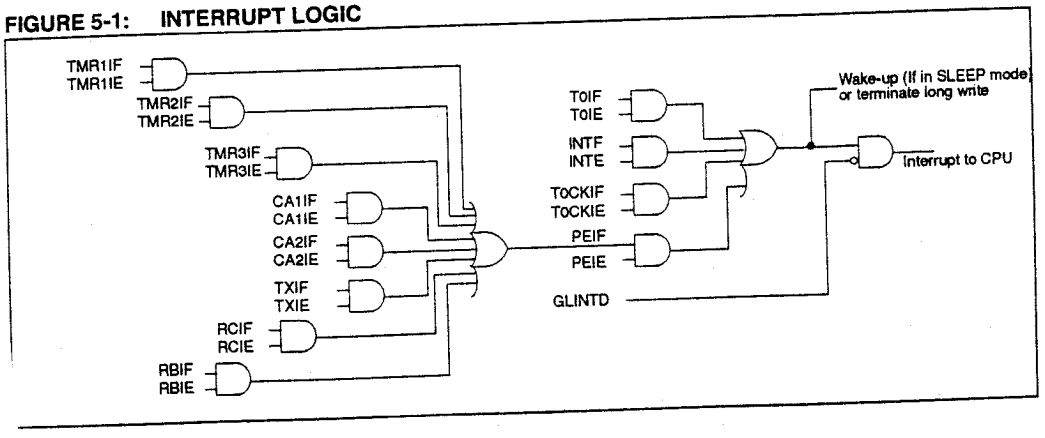
When program execution vectors to one of these interrupt vector addresses (except for the peripheral interrupt address), the interrupt flag bit is automatically cleared. Vectoring to the peripheral interrupt vector address does not automatically clear the source of the interrupt. In the peripheral interrupt service routine, the source(s) of the interrupt can be determined by testing the interrupt flag bits. The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid infinite interrupt requests.

All of the individual interrupt flag bits will be set regardless of the status of their corresponding mask bit or the GLINTD bit.

For external interrupt events, there will be an interrupt latency. For two cycle instructions, the latency could be one instruction cycle longer.

The "return from interrupt" instruction, `RETFIE`, can be used to mark the end of the interrupt service routine. When this instruction is executed, the stack is "POPed", and the GLINTD bit is cleared (to re-enable interrupts).

FIGURE 5-1: INTERRUPT LOGIC



5.1 Interrupt Status Register (INTSTA)

The Interrupt Status/Control register (INTSTA) records the individual interrupt requests in flag bits, and contains the individual enable bits (not for the peripherals).

The PEIF bit is a read only, bit wise OR of all the peripheral flag bits in the PIR register (Figure 5-4).

Note: T0IF, INTF, T0CKIF, or PEIF will be set by the specified condition, even if the corresponding interrupt enable bit is cleared (interrupt disabled) or the GLINTD bit is set (all interrupts disabled).

FIGURE 5-2: INTSTA REGISTER (ADDRESS: 07H, UNBANKED)

R - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0
PEIF	T0CKIF	T0IF	INTF	PEIE	T0CKIE	T0IE	INTE
bit7							bit0

R = Readable bit
W = Writable bit
- n = Value at POR reset

bit 7: **PEIF**: Peripheral Interrupt Flag bit
This bit is the OR of all peripheral interrupt flag bits AND'ed with their corresponding enable bits.
1 = A peripheral interrupt is pending
0 = No peripheral interrupt is pending

bit 6: **T0CKIF**: External Interrupt on T0CKI Pin Flag bit
This bit is cleared by hardware, when the interrupt logic forces program execution to vector (18h).
1 = The software specified edge occurred on the RA1/T0CKI pin
0 = The software specified edge did not occur on the RA1/T0CKI pin

bit 5: **T0IF**: Timer0 Overflow Interrupt Flag bit
This bit is cleared by hardware, when the interrupt logic forces program execution to vector (10h).
1 = Timer0 overflowed
0 = Timer0 did not overflow

bit 4: **INTF**: External Interrupt on INT Pin Flag bit
This bit is cleared by hardware, when the interrupt logic forces program execution to vector (08h).
1 = The software specified edge occurred on the RA0/INT pin
0 = The software specified edge did not occur on the RA0/INT pin

bit 3: **PEIE**: Peripheral Interrupt Enable bit
This bit enables all peripheral interrupts that have their corresponding enable bits set.
1 = Enable peripheral interrupts
0 = Disable peripheral interrupts

bit 2: **T0CKIE**: External Interrupt on T0CKI Pin Enable bit
1 = Enable software specified edge interrupt on the RA1/T0CKI pin
0 = Disable interrupt on the RA1/T0CKI pin

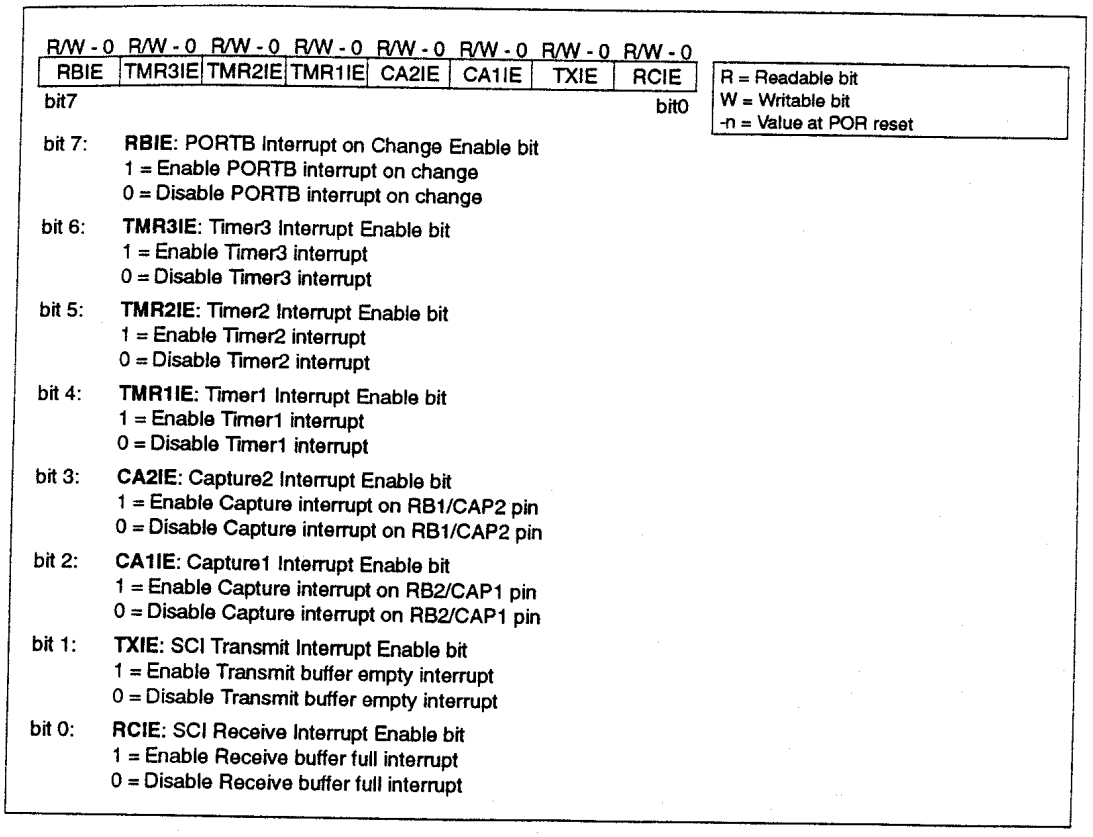
bit 1: **T0IE**: Timer0 Overflow Interrupt Enable bit
1 = Enable Timer0 overflow interrupt
0 = Disable Timer0 overflow interrupt

bit 0: **INTE**: External Interrupt On INT Pin Enable bit
1 = Enable software specified edge interrupt on the RA0/INT pin
0 = Disable software specified edge interrupt on the RA0/INT pin

5.2 Peripheral Interrupt Enable Register (PIE)

This register contains the individual flag bits for the Peripheral interrupts.

FIGURE 5-3: PIE REGISTER (ADDRESS: 17H, BANK 1)



5.3 Peripheral Interrupt Request Register (PIR)

This register contains the individual flag bits for the peripheral interrupts.

Note: These bits will be set by the specified condition, even if the corresponding interrupt enable bit is cleared (interrupt disabled), or the GLINTD bit is set (all interrupts disabled). Before enabling an interrupt, the user may wish to clear the interrupt flag to ensure that the program does not immediately branch to the peripheral interrupt service routine.

FIGURE 5-4: PIR REGISTER (ADDRESS: 16H, BANK 1)

R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R - 1	R - 0
RBIF	TMR3IF	TMR2IF	TMR1IF	CA2IF	CA1IF	TXIF	RCIF
bit7							bit0

R = Readable bit
W = Writable bit
-n = Value at POR reset

bit 7: **RBIF**: PORTB Interrupt on Change Flag bit
1 = One of the PORTB inputs changed (Software must end the mismatch condition)
0 = None of the PORTB inputs have changed

bit 6: **TMR3IF**: Timer3 Interrupt Flag bit
If Capture1 is enabled (CA1/$\overline{PR3}$ = 1)
1 = Timer3 overflowed
0 = Timer3 did not overflow

If Capture1 is disabled (CA1/$\overline{PR3}$ = 0)
1 = Timer3 value has rolled over to 0000h from equalling the period register (PR3H:PR3L) value
0 = Timer3 value has not rolled over to 0000h from equalling the period register (PR3H:PR3L) value

bit 5: **TMR2IF**: Timer2 Interrupt Flag bit
1 = Timer2 value has rolled over to 0000h from equalling the period register (PR2) value
0 = Timer2 value has not rolled over to 0000h from equalling the period register (PR2) value

bit 4: **TMR1IF**: Timer1 Interrupt Flag bit
If Timer1 is in 8-bit mode (T16 = 0)
1 = Timer1 value has rolled over to 0000h from equalling the period register (PR) value
0 = Timer1 value has not rolled over to 0000h from equalling the period register (PR2) value

If Timer1 is in 16-bit mode (T16 = 1)
1 = TMR1:TMR2 value has rolled over to 0000h from equalling the period register (PR1:PR2) value
0 = TMR1:TMR2 value has not rolled over to 0000h from equalling the period register (PR1:PR2) value

bit 3: **CA2IF**: Capture2 Interrupt Flag bit
1 = Capture event occurred on RB1/CAP2 pin
0 = Capture event did not occur on RB1/CAP2 pin

bit 2: **CA1IF**: Capture1 Interrupt Flag bit
1 = Capture event occurred on RB0/CAP1 pin
0 = Capture event did not occur on RB0/CAP1 pin

bit 1: **TXIF**: SCI Transmit Interrupt Flag bit
1 = Transmit buffer is empty
0 = Transmit buffer is full

bit 0: **RCIF**: SCI Receive Interrupt Flag bit
1 = Receive buffer is full
0 = Receive buffer is empty

5.4 Interrupt Operation

Global Interrupt Disable bit, GLINTD (CPUSTA<4>), enables all unmasked interrupts (if clear) or disables all interrupts (if set). Individual interrupts can be disabled through their corresponding enable bits in the INTSTA register. Peripheral interrupts need either the global peripheral enable PEIE bit disabled, or the specific peripheral enable bit disabled. Disabling the peripherals via the global peripheral enable bit, disables all peripheral interrupts. GLINTD is set on reset (interrupts disabled).

The RETFIE instruction allows returning from interrupt and re-enable interrupts at the same time.

When an interrupt is responded to, the GLINTD bit is automatically set to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with interrupt vector. There are four interrupt vectors to reduce interrupt latency.

The peripheral interrupt vector has multiple interrupt sources. Once in the peripheral interrupt service routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits. The peripheral interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid continuous interrupts.

The PIC17C4X devices have 4 interrupt vectors. These vectors and their hardware priority are shown in Table 5-1. If two enabled interrupts occur "at the same time", the interrupt of the highest priority will be serviced first. This means that the vector address of that interrupt will be loaded into the program counter (PC).

TABLE 5-1: INTERRUPT VECTORS/ PRIORITIES

Address	Vector	Priority
0008h	External Interrupt on RA0/ INT pin (INTF)	1 (Highest)
0010h	TMR0 overflow interrupt (T0IF)	2
0018h	External Interrupt on T0CKI (T0CKIF)	3
0020h	Peripherals (PEIF)	4 (Lowest)

Note 1: Individual interrupt flag bits are set regardless of the status of their corresponding mask bit or the GLINTD bit.

Note 2: For the PIC17C42 only:
If an interrupt occurs while the Global Interrupt Disable (GLINTD) bit is being set, the GLINTD bit may unintentionally be re-enabled by the user's Interrupt Service Routine (the RETFIE instruction). The events that would cause this to occur are:

1. An interrupt occurs simultaneously with an instruction that sets the GLINTD bit.
2. The program branches to the interrupt vector and executes the Interrupt Service Routine.
3. The Interrupt Service Routine completes with the execution of the RETFIE instruction. This causes the GLINTD bit to be cleared (enables interrupts), and the program returns to the instruction after the one which was meant to disable interrupts.

The method to ensure that interrupts are globally disabled is:

1. Ensure that the GLINTD bit was set by the instruction, as shown in the following code:

```
LOOP   BCF    CPUSTA, GLINTD  ; Disable Global
                               ; Interrupts
       BTFSC  CPUSTA, GLINTD  ; Global Interrupts
                               ; Disabled?
       GOTO   LOOP            ; NO, try again
                               ; YES, continue
                               ; with program
                               ; flow
```

5.5 INT Interrupt

The external interrupt on the RA0/INT pin is edge triggered. Either the rising edge, if INTEDG bit (T0STA<7>) is set, or the falling edge, if INTEDG bit is clear. When a valid edge appears on the RA0/INT pin, the INTF bit (INTSTA<4>) is set. This interrupt can be disabled by clearing the INTE control bit (INTSTA<0>). The INT interrupt can wake the processor from SLEEP. See Section 14.4 for details on SLEEP operation.

5.6 TMR0 Interrupt

An overflow (FFFFh → 0000h) in TMR0 will set the T0IF (INTSTA<5>) bit. The interrupt can be enabled/disabled by setting/clearing the T0IE control bit (INTSTA<1>). For operation of the TMR0 module, see Section 11.0.

5.7 T0CKI Interrupt

The external interrupt on the RA1/T0CKI pin is edge triggered. Either the rising edge, if the T0SE bit (T0STA<6>) is set, or the falling edge, if the T0SE bit is clear. When a valid edge appears on the RA1/T0CKI pin, the T0CKIF bit (INTSTA<6>) is set. This interrupt can be disabled by clearing the T0CKIE control bit (INTSTA<2>). The T0CKI interrupt can wake up the processor from SLEEP. See Section 14.4 for details on SLEEP operation.

5.8 Peripheral Interrupt

The peripheral interrupt flag indicates that at least one of the peripheral interrupts occurred (PEIF is set). The PEIF bit is a read only bit, and is a bit wise OR of all the flag bits in the PIR register AND'ed with the corresponding enable bits in the PIE register. Some of the peripheral interrupts can wake the processor from SLEEP. See Section 14.4 for details on SLEEP operation.

5.9 Context Saving During Interrupts

During an interrupt, only the returned PC value is saved on the stack. Typically, users may wish to save key registers during an interrupt, e.g. WREG, ALUSTA and the BSR registers. This requires implementation in software.

Example 5-1 shows the saving and restoring of information for an interrupt service routine. The PUSH and POP routines could either be in each interrupt service routine or could be subroutines that were called. Depending on the application, other registers may also need to be saved, such as PCLATH.

EXAMPLE 5-1: SAVING STATUS AND WREG IN RAM

```
;
; The addresses that are used to store the CPUSTA and WREG values
; must be in the data memory address range of 18h - 1Fh. Up to
; 8 locations can be saved and restored using
; the MOVFP instruction. This instruction neither affects the status
; bits, nor corrupts the WREG register.
;
;
PUSH    MOVFP   WREG, TEMP_W          ; Save WREG
        MOVFP   ALUSTA, TEMP_STATUS   ; Save ALUSTA
        MOVFP   BSR, TEMP_BSR         ; Save BSR

ISR     :                             ; This is the interrupt service routine
        :
POP     MOVFP   TEMP_W, WREG          ; Restore WREG
        MOVFP   TEMP_STATUS, ALUSTA   ; Restore ALUSTA
        MOVFP   TEMP_BSR, BSR         ; Restore BSR
        RETFIE                        ; Return from Interrupts enabled
```

6.0 MEMORY ORGANIZATION

There are two memory blocks in the PIC17C4X; program memory and data memory. Each block has its own bus, so that access to each block can occur during the same oscillator cycle.

The data memory can further be broken down into General Purpose RAM and the Special Function Registers (SFRs). The operation of the SFRs that control the "core" are described here. The SFRs used to control the peripheral modules are described in the section discussing each individual peripheral module.

6.1 Program Memory Organization

PIC17C4X devices have a 16-bit program counter capable of addressing a 64K x 16 program memory space. The reset vector is at 0000h and the interrupt vectors are at 0008h, 0010h, 0018h, and 0020h (Figure 6-1).

6.1.1 PROGRAM MEMORY OPERATION

The PIC17C4X can operate in one of four possible program memory configurations. The configuration is selected by two configuration bits. The possible modes are:

- Microprocessor
- Microcontroller
- Extended Microcontroller
- Protected Microcontroller

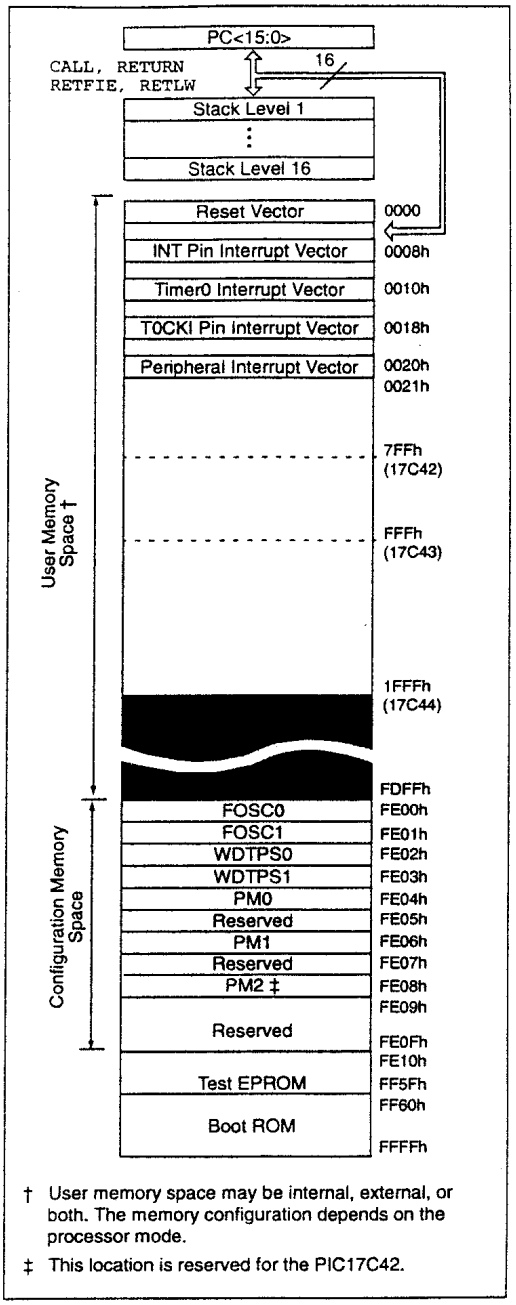
The microcontroller and protected microcontroller modes only allow internal execution. Any access beyond the program memory reads unknown data. The protected microcontroller mode also enables the code protection feature.

The extended microcontroller mode accesses both the internal program memory as well as external program memory. Execution automatically switches between internal and external memory. The 16-bits of address allow a program memory range of 64K-words.

The microprocessor mode only accesses the external program memory. The on-chip program memory is ignored. The 16-bits of address allow a program memory range of 64K-words. Microprocessor mode is the default mode of an unprogrammed device.

The different modes allow different access to the configuration bits, test memory, and boot ROM. Table 6-1 lists which modes can access which areas in memory. Test Memory and Boot Memory are not required for normal operation of the device. Care should be taken to ensure that no unintended branches occur to these areas.

FIGURE 6-1: PROGRAM MEMORY MAP AND STACK



† User memory space may be internal, external, or both. The memory configuration depends on the processor mode.

‡ This location is reserved for the PIC17C42.

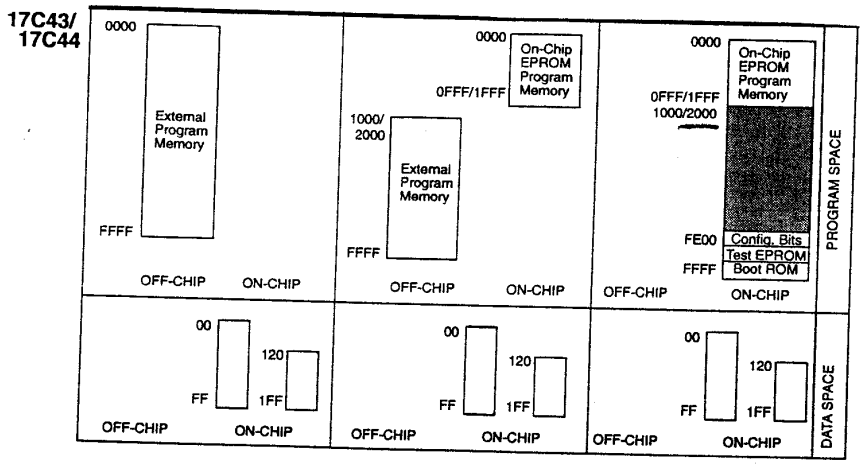
TABLE 6-1: MODE MEMORY ACCESS

Operating Mode	Internal Program Memory	Configuration Bits, Test Memory, Boot ROM
Microprocessor	No Access	No Access
Microcontroller	Access	Access
Extended Microcontroller	Access	No Access
Protected Microcontroller	Access	Access

The PIC17C4X can operate in modes where the program memory is off-chip. They are the microprocessor and extended microcontroller modes. The microprocessor mode is the default for an unprogrammed device.

Regardless of the processor mode, data memory is always on-chip.

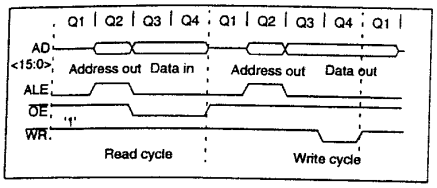
FIGURE 6-2: MEMORY MAP IN DIFFERENT MODES



6.1.2 EXTERNAL MEMORY INTERFACE

When either microprocessor or extended microcontroller mode is selected, PORTC, PORTD and PORTE are configured as the system bus. PORTC and PORTD are the multiplexed address/data bus and PORTE is for the control signals. External components are needed to demultiplex the address and data. This can be done as shown in Figure 6-4. The waveforms of address and data are shown in Figure 6-3. For complete timings, please refer to the electrical specification section.

FIGURE 6-3: EXTERNAL PROGRAM MEMORY ACCESS WAVEFORMS



As the speed of the processor increases, external EPROM memory with faster access time must be used. Table 6-2 lists external memory speed requirements for a given PIC17C4X device frequency.

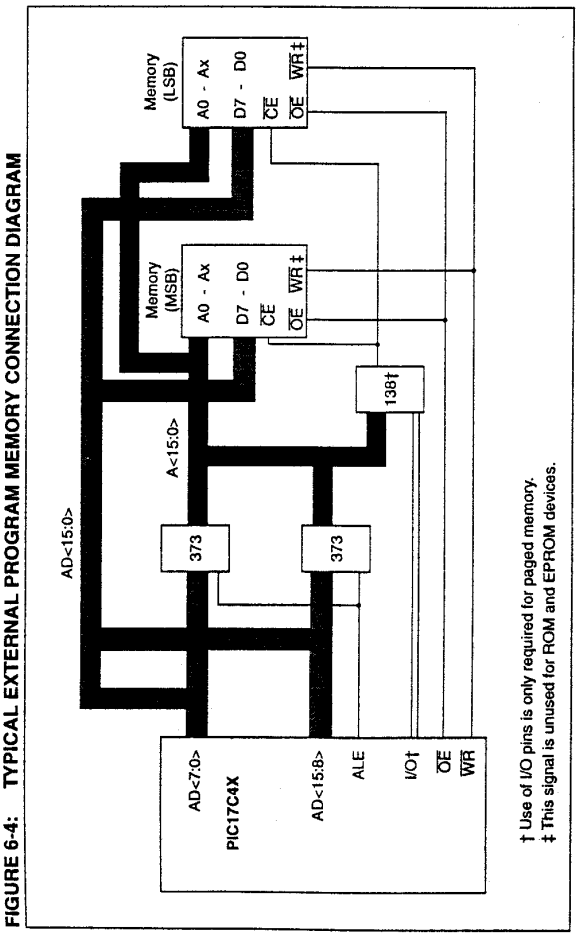
In extended microcontroller mode, when the device is executing out of internal memory, the control signals will continue to be active. That is, they indicate the action that is occurring in the internal memory. The external memory access is ignored.

TABLE 6-2: EPROM MEMORY ACCESS TIME ORDERING SUFFIX †

PIC17C4X Oscillator Frequency	Instruction Cycle Time (Tcy)	EPROM Suffix	
		PIC17C42	PIC17C43 PIC17C44
8 MHz	500 ns	-25	-25
16 MHz	250 ns	-12	-15
20 MHz	200 ns	-90	-10
25 MHz	160 ns	N.A.	-70

† This selection is for use with Microchip EPROMs. For interfacing to other manufacturers memory, please refer to the electrical specifications of the desired PIC17C4X device, as well as the desired memory device to ensure compatibility.

FIGURE 6-4: TYPICAL EXTERNAL PROGRAM MEMORY CONNECTION DIAGRAM



6.2 Data Memory Organization

Data memory is partitioned into two areas. The first is the General Purpose Registers (GPR) area, while the second is the Special Function Registers (SFR) area. The SFRs control the operation of the device.

Portions of data memory are banked, this is for both areas. The GPR area is banked to allow greater than 232 bytes of general purpose RAM. SFRs are for the registers that control the peripheral functions. Banking requires the use of control bits for bank selection. These control bits are located in the Bank Select Register (BSR). If an access is made to a location outside this banked region, the BSR bits are ignored. Figure 6-5 shows the data memory map organization for the PIC17C42 and Figure 6-6 for the PIC17C43 and PIC17C44 devices.

Instructions `MOVPF` and `MOVFP` provide the means to move values from the peripheral area ("P") to any location in the register file ("F"), and vice-versa. The definition of the "P" range is from 0h to 1Fh, while the "F" range is 0h to FFh. The "P" range has 8 more locations than peripheral registers (6 locations for the PIC17C43 and PIC17C44 devices)which can be used as General Purpose Registers. This can be useful in some applications where variables need to be copied to other locations in the general purpose RAM (such as saving status information during an interrupt).

The entire data memory can be accessed either directly or indirectly through file select registers FSR0 and FSR1 (see Section 6.4). Indirect addressing uses the appropriate control bits of the BSR for accesses into the banked areas of data memory. The BSR is explained in greater detail in Section 6.8.

6.2.1 GENERAL PURPOSE REGISTER (GPR)

All devices have some amount of GPR area. The GPRs are 8-bits wide. When the GPR area is greater than 232, it must be banked to allow access to the additional memory space.

Only the PIC17C43 and PIC17C44 devices have banked memory in the GPR area. To facilitate switching between these banks, the `MOVLR bank` instruction has been added to the instruction set. GPRs are not initialized by a POR reset and are unchanged on all other resets.

6.2.2 SPECIAL FUNCTION REGISTERS (SFR)

The SFRs are used by the CPU and peripheral functions to control the operation of the device (see Figure 6-5 and Figure 6-6). These registers are static RAM.

The SFRs can be classified into two sets, those associated with the "core" function and those related to the peripheral functions. Those registers related to the "core" are described here, while those related to a peripheral feature are described in the section for each peripheral feature.

The peripheral registers are in the banked portion of memory, while the core registers are in the unbanked region. To facilitate switching between the peripheral banks, the `MOVLB bank` instruction has been provided.

FIGURE 6-6: PIC17C43 AND PIC17C44 REGISTER FILE MAP

Addr	Unbanked			
00h	INDF0			
01h	FSR0			
02h	PCL			
03h	PCLATH			
04h	ALUSTA			
05h	T0STA			
06h	CPUSTA			
07h	INTSTA			
08h	INDF1			
09h	FSR1			
0Ah	WREG			
0Bh	TMR0L			
0Ch	TMR0H			
0Dh	TBLPTRL			
0Eh	TBLPTRH			
0Fh	BSR			
	Bank 0	**Bank 1** [1]	**Bank 2** [1]	**Bank 3** [1]
10h	PORTA	DDRC	TMR1	PW1DCL
11h	DDRB	PORTC	TMR2	PW2DCL
12h	PORTB	DDRD	TMR3L	PW1DCH
13h	RCSTA	PORTD	TMR3H	PW2DCH
14h	RCREG	DDRE	PR1	CA2L
15h	TXSTA	PORTE	PR2	CA2H
16h	TXREG	PIR	PR3L/CA1L	TCON1
17h	SPBRG	PIE	PR3H/CA1H	TCON2
18h	PRODL			
19h	PRODH			
1Ah 1Fh				
20h FFh	General Purpose RAM [2]	General Purpose RAM [2]		

Note 1: SFR file locations 10h - 17h are banked. All other SFRs ignore the Bank Select Register (BSR) bits.

2: General Purpose Registers (GPR) locations 20h - FFh and 120h - 1FFh are banked. All other GPRs ignore the Bank Select Register (BSR) bits.

TABLE 6-3: SPECIAL FUNCTION REGISTERS

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-On Reset	Value on all other resets [3]
Unbanked											
00h	INDF0	Uses contents of FSR0 to address data memory (not a physical register)								---- ----	---- ----
01h	FSR0	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
02h	PCL	Low order 8-bits of PC								0000 0000	0000 0000
03h	PCLATH [1,4]	Holding register for upper 8-bits of PC								xxxx xxxx	uuuu uuuu
04h	ALUSTA	FS3	FS2	FS1	FS0	OV	Z	DC	C	1111 xxxx	1111 uuuu
05h	T0STA	INTEDG	T0SE	T0CS	PS3	PS2	PS1	PS0	—	0000 000-	0000 000-
06h	CPUSTA [2]	—	—	STKAV	GLINTD	$\overline{TO}$	$\overline{PD}$	—	—	--11 11--	--11 ??--
07h	INTSTA	PEIF	T0CKIF	T0IF	INTF	PEIE	T0CKIE	T0IE	INTE	0000 0000	0000 0000
08h	INDF1	Uses contents of FSR1 to address data memory (not a physical register)								---- ----	---- ----
09h	FSR1	Indirect data memory address pointer 1								xxxx xxxx	uuuu uuuu
0Ah	WREG	Working register								xxxx xxxx	uuuu uuuu
0Bh	TMR0L	Timer0 low byte								xxxx xxxx	uuuu uuuu
0Ch	TMR0H	Timer0 high byte								xxxx xxxx	uuuu uuuu
0Dh	TBLPTRL	Low byte of program memory table pointer								Note 4	Note 4
0Eh	TBLPTRH	High byte of program memory table pointer								Note 4	Note 4
0Fh	BSR	Bank select register								0000 0000	0000 0000
Bank 0											
10h	PORTA	$\overline{RBPU}$	—	RA5	RA4	RA3	RA2	RA1/T0CKI	RA0/INT	0-xx xxxx	0-uu uuuu
11h	DDRB	Data direction register for PORTB								1111 1111	1111 1111
12h	PORTB	PORTB data latch								xxxx xxxx	uuuu uuuu
13h	RCSTA	SPEN	RC8/9	SREN	CREN	—	FERR	OERR	RCD8	0000 -00x	0000 -00u
14h	RCREG	Serial port receive register								xxxx xxxx	uuuu uuuu
15h	TXSTA	CSRC	TX8/9	TXEN	SYNC	—	—	TRMT	TXD8	0000 --1x	0000 --1u
16h	TXREG	Serial port transmit register								xxxx xxxx	uuuu uuuu
17h	SPBRG	Baud rate generator register								xxxx xxxx	uuuu uuuu
Bank 1											
10h	DDRC	Data direction register for PORTC								1111 1111	1111 1111
11h	PORTC	RC7/ AD7	RC6/ AD6	RC5/ AD5	RC4/ AD4	RC3/ AD3	RC2/ AD2	RC1/ AD1	RC0/ AD0	xxxx xxxx	uuuu uuuu
12h	DDRD	Data direction register for PORTD								1111 1111	1111 1111
13h	PORTD	RD7/ AD15	RD6/ AD14	RD5/ AD13	RD4/ AD12	RD3/ AD11	RD2/ AD10	RD1/ AD9	RD0/ AD8	xxxx xxxx	uuuu uuuu
14h	DDRE	Data direction register for PORTE								---- -111	---- -111
15h	PORTE	—	—	—	—	—	RE2/$\overline{WR}$	RE1/$\overline{OE}$	RE0/ALE	---- -xxx	---- -uuu
16h	PIR	RBIF	TMR3IF	TMR2IF	TMR1IF	CA2IF	CA1IF	TXIF	RCIF	0000 0010	0000 0010
17h	PIE	RBIE	TMR3IE	TMR2IE	TMR1IE	CA2IE	CA1IE	TXIE	RCIE	0000 0000	0000 0000

Legend: x = Unknown, u = Unchanged, - = Unimplemented, reads as '0', ? - Value depends on condition.

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a holding register for PC<15:8> whose contents are updated from or transferred to the upper byte of the program counter.

2: The $\overline{TO}$ and $\overline{PD}$ status bits in CPUSTA are not affected by a $\overline{MCLR}$ reset.

3: Other (non power-up) resets include: external reset through $\overline{MCLR}$ and the Watchdog Timer time-out reset.

4: The following values are for both TBLPTRL and TBLPTRH:
PIC17C42 (Power-On Reset xxxx xxxx) and (All other resets uuuu uuuu)
PIC17C43/PIC17C44 (Power-On Reset 0000 0000) and (All other resets 0000 0000)

5: These registers are only implemented in the PIC17C43 and PIC17C44.

TABLE 6-3: SPECIAL FUNCTION REGISTERS (CONT.)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-On Reset	Value on all other resets [3]
Bank 2											
10h	TMR1	Timer1								xxxx xxxx	uuuu uuuu
11h	TMR2	Timer2								xxxx xxxx	uuuu uuuu
12h	TMR3L	Timer3 low byte								xxxx xxxx	uuuu uuuu
13h	TMR3H	Timer3 high byte								xxxx xxxx	uuuu uuuu
14h	PR1	Timer1 period register								xxxx xxxx	uuuu uuuu
15h	PR2	Timer2 period register								xxxx xxxx	uuuu uuuu
16h	PR3L/CA1L	Timer3 period register, low byte/capture1 register, low byte								xxxx xxxx	uuuu uuuu
17h	PR3H/CA1H	Timer3 period register, high byte/capture1 register, high byte								xxxx xxxx	uuuu uuuu
Bank 3											
10h	PW1DCL	DC1	DC0	—	—	—	—	—	—	xx-- ----	uu-- ----
11h	PW2DCL	DC1	DC0	TM2PW2	—	—	—	—	—	xx0- ----	uu0- ----
12h	PW1DCH	DC9	DC8	DC7	DC6	DC5	DC4	DC3	DC2	xxxx xxxx	uuuu uuuu
13h	PW2DCH	DC9	DC8	DC7	DC6	DC5	DC4	DC3	DC2	xxxx xxxx	uuuu uuuu
14h	CA2L	Capture2 low byte								xxxx xxxx	uuuu uuuu
15h	CA2H	Capture2 high byte								xxxx xxxx	uuuu uuuu
16h	TCON1	CA2ED1	CA2ED0	CA1ED1	CA1ED0	T16	TMR3CS	TMR2CS	TMR1CS	0000 0000	0000 0000
17h	TCON2	CA2OVF	CA1OVF	PWM2ON	PWM1ON	CA1/$\overline{PR3}$	TMR3ON	TMR2ON	TMR1ON	0000 0000	0000 0000
Unbanked											
18h [5]	PRODL	Low Byte of 16-bit Product (8 x 8 Hardware Multiply)								xxxx xxxx	uuuu uuuu
19h [5]	PRODH	High Byte of 16-bit Product (8 x 8 Hardware Multiply)								xxxx xxxx	uuuu uuuu

6.2.2.1 ALU STATUS REGISTER (ALUSTA)

The ALUSTA register contains the status bits of the Arithmetic and Logic Unit and the mode control bits for the indirect addressing register.

As with all the other registers, the ALUSTA register can be the destination for any instruction. If the ALUSTA register is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to the device logic. Therefore, the result of an instruction with the ALUSTA register as destination may be different than intended.

For example, `CLRF ALUSTA` will clear the upper four bits and set the Z bit. This leaves the status register as `0000u1uu` (where u = unchanged).

It is recommended, therefore, that only `BCF`, `BSF`, `SWAPF` and `MOVWF` instructions be used to alter the status registers because these instructions do not affect any status bit. To see how other instructions affect the status bits, see the "Instruction Set Summary".

Note: The C and DC bits operate as a borrow out bit in subtraction. See the `SUBLW` and `SUBWF` instructions for examples.

Arithmetic and Logic Unit (ALU) is capable of carrying out arithmetic or logical operations on two operands or a single operand. All single operand instructions operate either on the WREG register or a file register. For two operand instructions, one of the operands is the WREG register and the other one is either a file register or an 8-bit immediate constant.

FIGURE 6-7: ALUSTA REGISTER (ADDRESS: 04H, UNBANKED)

R/W - 1	R/W - 1	R/W - 1	R/W - 1	R/W - x	R/W - x	R/W - x	R/W - x
FS3	FS2	FS1	FS0	OV	Z	DC	C
bit7							bit0

R = Readable bit
W = Writable bit
-n = Value at POR reset
(x = unknown)

bit 7-6: **FS3:FS2**: FSR1 Mode Select bits
00 = Post auto-decrement FSR1 value
01 = Post auto-increment FSR1 value
1x = FSR1 value does not change

bit 5-4: **FS1:FS0**: FSR0 Mode Select bits
00 = Post auto-decrement FSR0 value
01 = Post auto-increment FSR0 value
1x = FSR0 value does not change

bit 3: **OV**: Overflow bit
This bit is used for signed arithmetic. It indicates an overflow of the 7-bit magnitude, which causes the sign bit (bit7) to change state.
1 = Overflow occurred for signed arithmetic, (in this arithmetic operation)
0 = No overflow occurred

bit 2: **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The results of an arithmetic or logic operation is not zero

bit 1: **DC**: Digit carry/borrow bit
For `ADDWF` and `ADDLW` instructions.
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result
Note: For borrow the polarity is reversed.

bit 0: **C**: carry/borrow bit
For `ADDWF` and `ADDLW` instructions.
1 = A carry-out from the most significant bit of the result occurred
Note that a subtraction is executed by adding the two's complement of the second operand. For rotate (`RRCF`, `RLCF`) instructions, this bit is loaded with either the high or low order bit of the source register.
0 = No carry-out from the most significant bit of the result occurred
Note: For borrow the polarity is reversed.

2.2 CPU STATUS REGISTER (CPUSTA)

CPUSTA register contains the status and control for the CPU. This register is used to globally ble/disable interrupts. If only a specific interrupt is red to be enabled/disabled, please refer to the rrupt STAtus (INTSTA) register and the Peripheral rupt Enable (PIE) register. This register also indis if the stack is available and contains the Power n ($\overline{PD}$) and Time-Out ($\overline{TO}$) bits. The $\overline{TO}$, $\overline{PD}$, and AV bits are not writable. These bits are set or red according to device logic. Therefore, the result n instruction with the CPUSTA register as destinamay be different than intended.

FIGURE 6-8: CPUSTA REGISTER (ADDRESS: 06H, UNBANKED)

U - 0	U - 0	R - 1	R/W - 1	R - 1	R - 1	U - 0	U - 0
—	—	STKAV	GLINTD	$\overline{TO}$	$\overline{PD}$	—	—
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, Read as '0'
- n = Value at POR reset

bit 7-6: **Unimplemented**, reads as '0'

bit 5: **STKAV**: Stack Available bit
This bit indicates that the 4-bit stack pointer value is Fh, or has rolled over from Fh→ 0h (stack overflow).
1 = Stack is available
0 = Stack is full, or a stack-overflow may have occurred (Once this bit has been cleared by a stack overflow, only a device reset will set this bit)

bit 4: **GLINTD**: Global Interrupt Disable bit
This bit disables all interrupts. When enabling interrupts, only the sources with their enable bits set can cause an interrupt.
1 = Disable all interrupts
0 = Enables all un-masked interrupts

bit 3: **$\overline{TO}$**: WDT Time-Out status bit
1 = After power-up or by a `CLRWDT` instruction
0 = A Watchdog Timer time-out occurred

bit 2: **$\overline{PD}$**: Power Down status bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit 1-0: **Unimplemented**, reads as '0'

6.2.2.3 TMR0 STATUS/CONTROL REGISTER (T0STA)

This register contains various control bits. One bit is used to control the edge upon which a signal on the RA0/INT pin will set the INT interrupt flag. The other bits (shaded) configure the Timer0 prescaler and clock source. These shaded bits are described in Figure 11-1.

FIGURE 6-9: T0STA REGISTER (ADDRESS: 05H, UNBANKED)

R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	U - 0
INTEDG	T0SE	T0CS	PS3	PS2	PS1	PS0	—
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented, reads as '0'
-n = Value at POR reset

bit 7: **INTEDG:** INT Pin Interrupt Edge Select bit
This bit selects the edge upon which the interrupt is detected.
1 = Rising edge of RA0/INT pin generates interrupt
0 = Falling edge of RA0/INT pin generates interrupt

bit 6: **T0SE:** Timer0 Clock Input Edge Select bit
This bit selects the edge upon which TMR0 will increment.
When T0CS = 0
1 = Rising edge of RA1/T0CKI pin increments Timer0 and/or generates a T0CKIF interrupt
0 = Falling edge of RA1/T0CKI pin increments Timer0 and/or generates a T0CKIF interrupt
When T0CS = 1
Don't care

bit 5: **T0CS:** Timer0 Clock Source Select bit
This bit selects the clock source for TMR0.
1 = Internal instruction clock cycle (TCY)
0 = T0CKI pin

bit 4-1: **PS3:PS0:** Timer0 Prescale Selection bits
These bits select the prescale value for TMR0.

PS3:PS0	Prescale Value
0000	1:1
0001	1:2
0010	1:4
0011	1:8
0100	1:16
0101	1:32
0110	1:64
0111	1:128
1xxx	1:256

bit 0: **Unimplemented:** reads as '0'

6.3 Stack Operation

The PIC17C4X devices have a 16 x 16-bit wide hardware stack (see Figure 6-1). The stack is not part of either the program or data memory space, and the stack pointer is neither readable nor writable. The PC is "PUSHed" onto the stack when a CALL instruction is executed or an interrupt is acknowledged. The stack is "POPed" in the event of a RETURN, RETLW, or a RETFIE instruction execution. PCLATH is not affected by a "PUSH" or a "POP" operation.

The stack operates as a circular buffer, with the stack pointer initialized to '0' after all resets. There is a stack available bit (STKAV) to allow software to ensure that the stack has not overflowed. The STKAV bit is set after a device reset. When the stack pointer equals Fh, STKAV is cleared. When the stack pointer rolls over from Fh to 0h, the STKAV bit will be held clear until a device reset.

Note 1: There is not a status bit for stack underflow. The STKAV bit can be used to detect the underflow which results in the stack pointer being at the top of stack.

Note 2: There are no instruction mnemonics called PUSH or POP. These are actions that occur from the execution of the CALL, RETURN, RETLW, and RETFIE instructions, or the vectoring to an interrupt vector.

Note 3: After a reset, if a "POP" operation occurs before "PUSH" operation, the STKAV bit will be cleared. This will appear as if the stack is full (underflow has occured). If a "PUSH" operation occurs next, the STKAV bit will be locked clear. Only a device reset will cause this bit to set.

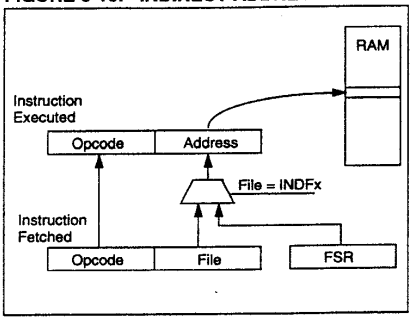
After the device is "PUSHed" sixteen times (without a "POP"), the seventeenth push overwrites the value from the first push. The eighteenth push overwrites the second push (and so on).

6.4 Indirect Addressing

Indirect addressing is a mode of addressing data memory where the data memory address in the instruction is not fixed. That is, the register that is to be read or written can be modified by the program. This can be useful for data tables in the data memory. Figure 6-10 shows the operation of indirect addressing. This shows the moving of the value to the data memory address specified by the value of the FSR register.

Example 6-1 shows the use of indirect addressing to clear RAM in a minimum number of instructions. A similar concept could be used to move a defined number of bytes (block) of data to the SCI transmit register (TXREG). The starting address of the block of data to be transmitted could easily be modified by the program.

FIGURE 6-10: INDIRECT ADDRESSING



6.4.1 INDIRECT ADDRESSING REGISTERS

The PIC17C4X has four registers for indirect addressing. These registers are:

- INDF0 and FSR0
- INDF1 and FSR1

Registers INDF0 and INDF1 are not physically implemented. Reading or writing to these registers activates indirect addressing, with the value in the corresponding FSR register being the address of the data. The FSR is an 8-bit register and allows addressing anywhere in the 256-byte data memory address range. For banked memory, the bank of memory accessed is specified by the value in the BSR.

If file INDF0 (or INDF1) itself is read indirectly via an FSR, all '0's are read (Zero bit is set). Similarly, if INDF0 (or INDF1) is written to indirectly, the operation will be equivalent to a NOP, and the status bits are not affected.

6.4.2 INDIRECT ADDRESSING OPERATION

The indirect addressing capability has been enhanced over that of the PIC16CXX family. There are two control bits associated with each FSR register. These two bits configure the FSR register to:

- Auto-decrement the value (address) in the FSR after an indirect access
- Auto-increment the value (address) in the FSR after an indirect access
- No change to the value (address) in the FSR after an indirect access

These control bits are located in the ALUSTA register. The FSR1 register is controlled by the FS3:FS2 bits and FSR0 is controlled by the FS1:FS0 bits.

When using the auto-increment or auto-decrement features, the effect on the FSR is not reflected in the ALUSTA register. For example, if the indirect address causes the FSR to equal '0', the Z bit will not be set.

If the FSR register contains a value of 0h, an indirect read will read 0h (Zero bit is set) while an indirect write will be equivalent to a NOP (status bits are not affected).

Indirect addressing allows single cycle data transfers within the entire data space. This is possible with the use of the MOVPF and MOVFP instructions, where either 'p' or 'f' is specified as INDF0 (or INDF1).

If the source or destination of the indirect address is in banked memory, the location accessed will be determined by the value in the BSR.

A simple program to clear RAM from 20h - FFh is shown in Example 6-1.

EXAMPLE 6-1: INDIRECT ADDRESSING

```
    MOVLW   0x20          ;
    MOVWF   FSR0          ; FSR0 = 20h
    BCF     ALUSTA, FS1   ; Increment FSR
    BSF     ALUSTA, FS0   ; after access
    BCF     ALUSTA, C     ; C = 0
    MOVLW   END_RAM + 1   ;
LP  CLRF    INDF0         ; Addr(FSR) = 0
    CPFSEQ  FSR0          ; FSR0 = END_RAM+1?
    GOTO    LP            ; NO, clear next
    :                     ; YES, All RAM is
    :                     ; cleared
```

6.5 Table Pointer (TBLPTRL and TBLPTRH)

File registers TBLPTRL and TBLPTRH form a 16-bit pointer to address the 64K program memory space. The table pointer is used by instructions TABLWT and TABLRD.

The TABLRD and the TABLWT instructions allow transfer of data between program and data space. The table pointer serves as the 16-bit address of the data word within the program memory. For a more complete description of these registers and the operation of Table Reads and Table Writes, see Section 7.0.

6.6 Table Latch (TBLATH, TBLATL)

The table latch (TBLAT) is a 16-bit register, with TBLATH and TBLATL referring to the high and low bytes of the register. It is not mapped into data or program memory. The table latch is used as a temporary holding latch during data transfer between program and data memory (see descriptions of instructions TABLRD, TABLWT, TLRD and TLWT). For a more complete description of these registers and the operation of Table Reads and Table Writes, see Section 7.0.

6.7 Program Counter Module

The Program Counter (PC) is a 16-bit register. PCL, the low byte of the PC, is mapped in the data memory. PCL is readable and writable just as is any other register. PCH is the high byte of the PC and is not directly addressable. Since PCH is not mapped in data or program memory, an 8-bit register PCLATH (PC high latch) is used as a holding latch for the high byte of the PC. PCLATH is mapped into data memory. The user can read or write PCH through PCLATH.

The 16-bit wide PC is incremented after each instruction fetch during Q1 unless:

- Modified by GOTO, CALL, LCALL, RETURN, RETLW, or RETFIE instruction
- Modified by an interrupt response
- Due to destination write to PCL by an instruction

"Skips" are equivalent to a forced NOP cycle at the skipped address.

Note: On POR, the contents of the PCLATH register are unknown. The PCLATH should be initialized before a CALL, GOTO, or any instruction that modifies the PCL register is executed.

The operations of the PC and PCLATH for different instructions are as follows:

a) LCALL instruction:
   An 8-bit destination address is provided in the instruction (opcode). PCLATH is unchanged.
   PCLATH → PCH
   Opcode<7:0> → PCL

b) CALL, GOTO instructions:
   A 13-bit destination address is provided in the instruction (opcode).
   Opcode<12:0> → PC <12:0>
   PC<15:13> → PCLATH<7:5>
   Opcode<12:8> → PCLATH <4:0>

c) Read instructions on PCL:
   Any instruction that reads PCL.
   PCL → data bus → ALU or destination
   PCH → PCLATH

d) Write instructions on PCL:
   Any instruction that writes to PCL.
   8-bit data → data bus → PCL
   PCLATH → PCH

e) Read-Modify-Write instructions on PCL:
   Any instruction that does a read-write-modify operation on PCL, such as ADDWF PCL.
   Read: PCL → data bus → ALU
   Write: 8-bit result → data bus → PCL
   PCLATH → PCH

f) RETURN instruction:
   PCH → PCLATH

The read-modify-write only affects the PCL with the result. PCH is loaded with the value in the PCLATH. For example, ADDWF PCL will result in a jump within the current page. If PC = 03F0h, WREG = 30h and PCLATH = 03h before instruction, PC = 0320h after the instruction. To accomplish a true 16-bit computed jump, the user needs to compute the 16-bit destination address, write the high byte to PCLATH and then write the low value to PCL.

The following PC related operations do not change PCLATH:

a) LCALL, RETLW, and RETFIE instructions.
b) Interrupt vector is forced onto the PC.
c) Read-modify-write instructions on PCL (e.g. BSF PCL).

6.8 Bank Select Register (BSR)

The BSR is used to switch between banks in the data memory area (see Figure 6-11). In the PIC17C42, only the lower nibble is implemented. While in the PIC17C43 and PIC17C44 devices, the entire byte is implemented. The lower nibble is used to select the peripheral register bank. The upper nibble is used to select the general purpose memory bank.

All the Special Function Registers (SFRs) are mapped into the data memory space. In order to accommodate the large number of registers, a banking scheme has been used. A segment of the SFRs, from address 10h to address 17h, is banked. The lower nibble of the bank select register (BSR) selects the currently active "peripheral bank". Effort has been made to group the peripheral registers of related functionality in one bank. However, it will still be necessary to switch from bank to bank in order to address all peripherals related to a single task. To assist this, a MOVLB bank instruction is in the instruction set.

For the PIC17C43 and PIC17C44 devices, the need for a large general purpose memory space dictated a general purpose RAM banking scheme. The upper nibble of the BSR selects the currently active general purpose RAM bank. To assist this, a MOVLR bank instruction has been provided in the instruction set.

If the currently selected bank is not implemented (such as Bank 13), any read will read all '0's. Any write is completed to the bit bucket and the ALU status bits will be set/cleared as appropriate.

Note: Registers in Bank 15 in the Special Function Register area, are reserved for Microchip use. Reading of registers in this bank may cause random values to be read.

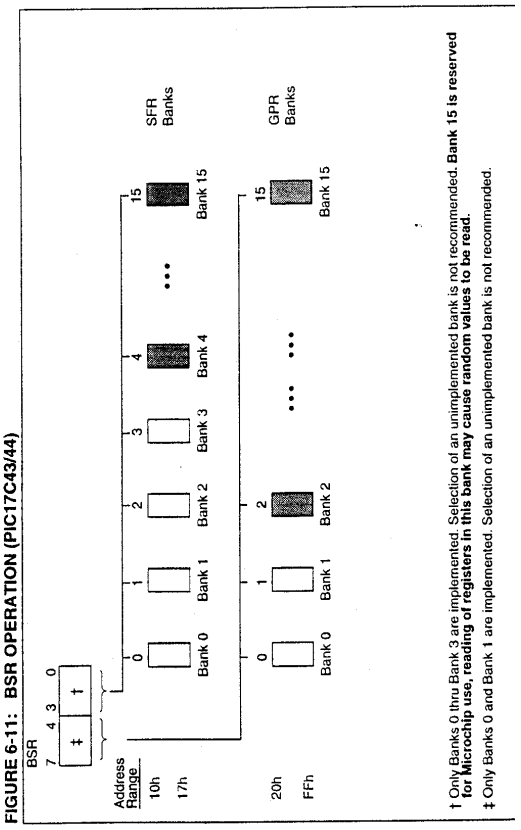


FIGURE 6-11: BSR OPERATION (PIC17C43/44)

7.0 TABLE READS AND TABLE WRITES

The PIC17C4X has four instructions that allow the processor to move data from the data memory space to the program memory space, and vice versa. Since the program memory space is 16-bits wide and the data memory space is 8-bits wide, two operations are required to move 16-bit values to/from the data memory.

The `TLWT t,f` and `TABLWT t,i,f` instructions are used to write data from the data memory space to the program memory space. The `TLRD t,f` and `TABLRD t,i,f` instructions are used to write data from the program memory space to the data memory space.

The program memory can be internal or external. For the program memory access to be external, the device needs to be operating in extended microcontroller or microprocessor mode.

8.0 HARDWARE MULTIPLIER

The PIC17C43 and PIC17C44 devices have an 8 x 8 hardware multiplier included in the ALU of the device. By making the multiply a hardware operation, it completes in a single instruction cycle. This is an unsigned multiply that gives a 16-bit result. The result is stored into the 16-bit PRODuct register (PRODH:PRODL). The multiplier does not affect any flags in the ALUSTA register.

Making the 8 x 8 multiplier execute in a single cycle gives the following advantages:

- Higher computational throughput
- Reduces code size requirements for multiply algorithms

The performance increase allows the PIC17C43 and PIC17C44 devices to be used in applications previously reserved for Digital Signal Processors.

9.0 I/O PORTS

The PIC17C4X devices have five I/O ports, PORTA through PORTE. PORTB through PORTE have a corresponding Data Direction Register (DDR), which is used to configure the port pins as inputs or outputs. These five ports are made up of 33 I/O pins. Some of these ports pins are multiplexed with alternate functions.

PORTC, PORTD and PORTE are multiplexed with the system bus. These pins are configured as the system bus when the device's fuses are selected to Microprocessor or Extended Microcontroller modes. In the two other microcontroller modes, these pins are general purpose I/O.

PORTA and PORTB are multiplexed with the peripheral features of the device. These peripheral features are:

- Timer modules
- Capture module
- PWM module
- SCI module (USART)
- External Interrupt pin

When some of these peripheral modules are turned on, the port pin will automatically configure to the alternate function. The modules that do this are:

- PWM module
- SCI module

When a pin is automatically configured as an output by a peripheral module, its data direction bit may be left in an unknown state. After disabling the peripheral module, the user should re-initialize the DDR bit to the desired configuration.

The other peripheral modules (which require an input) must have their data direction bit configured appropriately.

Note:	A pin that is a peripheral input, can be configured as an output (DDRx<y> is cleared). The peripheral events will be determined by the action output on the port pin.

9.1 PORTA Register

PORTA is a 6-bit wide latch. PORTA does not have a corresponding Data Direction Register (DDR).

Reading PORTA reads the status of the pins.

The RA1 pin is multiplexed with TMR0 clock input, and RA4 and RA5 are multiplexed with the SCI functions. The control of RA4 and RA5 as outputs is automatically configured by the SCI module.

9.1.1 USING RA2, RA3 AS OUTPUTS

The RA2 and RA3 pins are open collector outputs. To use the RA2 or the RA3 pin(s) as output(s), simply write to the PORTA register the desired value. A '0' will cause the pin to drive low, while a '1' will cause the pin to float (hi-impedance). An external pull-up resistor should be used to pull the pin high. Writes to PORTA will not affect the other pins.

Note:	When using the RA2 or RA3 pin(s) as output(s), read-modify-write instructions (such as BCF, BSF, BTG) on PORTA are not recommended. Such operations read the port pins, do the desired operation, and then write this value to the data latch. This may inadvertently cause the RA2 or RA3 pins to switch from input to output (or vice-versa).

.2 PORTB and DDRB Registers

ORTB is an 8-bit wide bi-directional port. The corre-
ponding data direction register is DDRB. A '1' in DDRB
onfigures the corresponding port pin as an input. A '0'
the DDRB register configures the corresponding port
n as an output. Reading PORTB reads the status of
e pins, whereas writing to it will write to the port latch.

ach of the PORTB pins has a weak internal pull-up. A
ngle control bit can turn on all the pull-ups. This is
ne by clearing the $\overline{\text{RBPU}}$ (PORTA<7>) bit. The weak
ll-up is automatically turned off when the port pin is
nfigured as an output. The pull-ups are enabled on
y reset.

)RTB also has an interrupt on change feature. Only
is configured as inputs can cause this interrupt to
cur (i.e. any RB<7:0> pin configured as an output is
cluded from the interrupt on change comparison).
e input pins (of RB<7:0>) are compared with the
ue in the PORTB data latch. The "mismatch" outputs
RB<7:0> are OR'ed together to generate the RBIF
errupt (flag latched in PIR<7>).

his interrupt can wake the device from SLEEP. The
er, in the interrupt service routine, can clear the inter-
pt in one of two ways:

Disable the interrupt by clearing the RBIE (PIE<7>) bit.

Read-Write PORTB (`MOVPF PORTB, PORTB`). This will end mismatch condition. Then, clear the RBIF bit.

mismatch condition will continue to set the RBIF bit.
eading then writing PORTB will end the mismatch
ndition, and allow the RBIF bit to be cleared.

his interrupt on mismatch feature, together with soft-
are configurable pull-ups on this port, allows easy
terface to a key pad and make it possible for wake-up
n key-depression. (See AN552 in the *Embedded Con-
ol Handbook*).

he interrupt on change feature is recommended for
ake-up on operations where PORTB is only used for
e interrupt on change feature and key depression
peration.

mple 9-1 shows the instruction sequence to initial-
PORTB. The Bank Select Register (BSR) must be
cted to Bank 0 for the port to be initialized.

XAMPLE 9-1: INITIALIZING PORTB

```
VLB 0          ; Select Bank 0
RF  PORTB      ;Initialize PORTB by clearing
               ; output data latches
VLW 0xCF       ;Value used to initialize
               ;data direction
VWF DDRB       ;Set RB<3:0> as inputs
               ;   RB<5:4> as outputs
               ;   RB<7:6> as inputs
```

9.3 PORTC and DDRC Registers

PORTC is an 8-bit bi-directional port. The corresponding data direction register is DDRC. A '1' in DDRC configures the corresponding port pin as an input. A '0' in the DDRC register configures the corresponding port pin as an output. Reading PORTC reads the status of the pins, whereas writing to it will write to the port latch. PORTC is multiplexed with the system bus. When operating as the system bus, PORTC is the low order byte of the address/data bus (AD<7:0>). The timing for the system bus is shown in the Electrical Characteristics section.

Note: This port is configured as the system bus when the device's configuration bits are selected to Microprocessor or Extended Microcontroller modes. In the two other microcontroller modes, this port is a general purpose I/O.

Example 9-2 shows the instruction sequence to initialize PORTC. The Bank Select Register (BSR) must be selected to Bank 1 for the port to be initialized.

EXAMPLE 9-2: INITIALIZING PORTC

```
MOVLB  1          ;  Select Bank 1
CLRF   PORTC      ;  Initialize PORTC data
                  ;    latches before setting
                  ;    the data direction
                  ;    register
MOVLW  0xCF       ;  Value used to initialize
                  ;    data direction
MOVWF  DDRC       ;  Set RC<3:0> as inputs
                  ;    RC<5:4> as outputs
                  ;    RC<7:6> as inputs
```

9.4 PORTD and DDRD Registers

PORTD is an 8-bit bi-directional port. The corresponding data direction register is DDRD. A '1' in DDRD configures the corresponding port pin as an input. A '0' in the DDRC register configures the corresponding port pin as an output. Reading PORTD reads the status of the pins, whereas writing to it will write to the port latch. PORTD is multiplexed with the system bus. When operating as the system bus, PORTD is the high order byte of the address/data bus (AD<15:8>). The timing for the system bus is shown in the Electrical Characteristics section.

Note: This port is configured as the system bus when the device's configuration bits are selected to Microprocessor or Extended Microcontroller modes. In the two other microcontroller modes, this port is a general purpose I/O.

Example 9-3 shows the instruction sequence to initialize PORTD. The Bank Select Register (BSR) must be selected to Bank 1 for the port to be initialized.

EXAMPLE 9-3: INITIALIZING PORTD

```
MOVLB  1          ;  Select Bank 1
CLRF   PORTD      ;  Initialize PORTD data
                  ;    latches before setting
                  ;    the data direction
                  ;    register
MOVLW  0xCF       ;  Value used to initialize
                  ;    data direction
MOVWF  DDRD       ;  Set RD<3:0> as inputs
                  ;    RD<5:4> as outputs
                  ;    RD<7:6> as inputs
```

EXAMPLE 9-4: INITIALIZING PORTE

```
MOVLB  1          ;  Select Bank 1
CLRF   PORTE      ;  Initialize PORTE data
                  ;    latches before setting
                  ;    the data direction
                  ;    register
MOVLW  0x03       ;  Value used to initialize
                  ;    data direction
MOVWF  DDRE       ;  Set RE<1:0> as inputs
                  ;    RE<2> as outputs
                  ;    RE<7:3> are always
                  ;    read as '0'
```

Note: This port is configured as the system bus when the device's configuration bits are selected to Microprocessor or Extended Microcontroller modes. In the two other microcontroller modes, this port is a general purpose I/O.

Example 9-4 shows the instruction sequence to initialize PORTE. The Bank Select Register (BSR) must be selected to Bank 1 for the port to be initialized.

9.4.1 PORTE AND DDRE REGISTER

PORTE is a 3-bit bi-directional port. The corresponding data direction register is DDRE. A '1' in DDRE configures the corresponding port pin as an input. A '0' in the DDRE register configures the corresponding port pin as an output. Reading PORTE reads the status of the pins, whereas writing to it will write to the port latch. PORTE is multiplexed with the system bus. When operating as the system bus, PORTE contains the control signals for the address/data bus (AD<15:0>). These control signals are Address Latch Enable (ALE), Output Enable ($\overline{\text{OE}}$), and Write ($\overline{\text{WR}}$). The control signals $\overline{\text{OE}}$ and $\overline{\text{WR}}$ are active low signals. The timing for the system bus is shown in the Electrical Characteristics section.

0.0 OVERVIEW OF TIMER RESOURCES

he PIC17C4X has four timer modules. Each module an generate an interrupt to indicate that an event has ccurred. These timers are called:

TMR0 - Timer0 (16-bit timer with programmable 8-bit prescaler)
TMR1 - Timer1 (8-bit timer)
TMR2 - Timer2 (8-bit timer)
TMR3 - Timer3 (16-bit timer)

or enhanced time-base functionality, two input Capires and two Pulse Width Modulation (PWM) outputs e possible. The PWMs use the TMR1 and TMR2 sources and the input Captures use the TMR3 source.

0.1 TMR0 Overview

he TMR0 module is a simple 16-bit overflow counter. he clock source can be either the internal system ock (Fosc/4) or an external clock.

he TMR0 module also has a programmable prescaler ption. The PS3:PS0 bits (T0STA<4:1>) determine the escaler value. TMR0 can increment at the following ates: 1:1, 1:2, 1:4, 1:8, 1:16, 1:32, 1:64, 1:128, 1:256.

/hen TMR0's clock source is an external clock, the MR0 module can be selected to increment on either e rising or falling edge.

ynchronization of the external clock occurs after the escaler. When the prescaler is used, the external ock frequency may be higher then the device's frequency. The maximum frequency is 50 MHz, given the igh and low time requirements of the clock.

0.2 TMR1 Overview

he TMR1 module is an 8-bit timer/counter with an 8-it period register (PR1). When the TMR1 value rolls ver from the period match value to 0h, the TMR1IF ag is set, and an interrupt will be generated when nabled. In counter mode, the clock comes from the B4/TCLK12 pin, which can also be selected to be the ock for the TMR2 module.

MR1 can be concatenated to TMR2 to form a 16-bit mer. The TMR1 register is the LSB and TMR2 is the ISB. When in the 16-bit timer mode, there is a corresponding 16-bit period register (PR2:PR1). When the MR2:TMR1 value rolls over from the period match alue to 0h, the TMR1IF flag is set, and an interrupt ill be generated when enabled.

10.3 TMR2 Overview

The TMR2 module is an 8-bit timer/counter with an 8-bit period register (PR2). When the TMR2 value rolls over from the period match value to 0h, the TMR2IF flag is set, and an interrupt will be generated when enabled. In counter mode, the clock comes from the RB4/TCLK12 pin, which can also be selected to be the clock for the TMR1 module.

TMR1 can be concatenated to TMR2 to form a 16-bit timer. The TMR2 register is the MSB and TMR1 is the LSB. When in the 16-bit timer mode, there is a corresponding 16-bit period register (PR2:PR1). When the TMR2:TMR1 value rolls over from the period match value to 0h, the TMR1IF flag is set, and an interrupt will be generated when enabled.

10.4 TMR3 Overview

The TMR3 module is a 16-bit timer/counter with a 16-bit period register. When the TMR3H:TMR3L value rolls over to 0h, the TMR3IF bit is set and an interrupt will be generated when enabled. In counter mode, the clock comes from the RB5/TCLK3 pin.

When operating in the dual capture mode, the period registers become the second 16-bit capture register.

10.5 Role of the Timer/Counters

The timer modules are general purpose, but have dedicated resources associated with them. TMR1 and TMR2 are the time-bases for the two Pulse Width Modulation (PWM) outputs, while TMR3 is the time-base for the two input captures.

FIGURE 11-1: T0STA REGISTER (ADDRESS: 05H, UNBANKED)

R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	U - 0
INTEDG	T0SE	T0CS	PS3	PS2	PS1	PS0	—
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented, Read as '0'
-n = Value at POR reset

bit 7: **INTEDG**: INT Pin Interrupt Edge Select bit
This bit selects the edge upon which the interrupt is detected
1 = Rising edge of RA0/INT pin generates interrupt
0 = Falling edge of RA0/INT pin generates interrupt

bit 6: **T0SE**: Timer0 Clock Input Edge Select bit
This bit selects the edge upon which TMR0 will increment
When T0CS = 0
1 = Rising edge of RA1/T0CKI pin increments Timer0 and/or generates a T0CKIF interrupt
0 = Falling edge of RA1/T0CKI pin increments Timer0 and/or generates a T0CKIF interrupt
When T0CS = 1
Don't care

bit 5: **T0CS**: Timer0 Clock Source Select bit
This bit selects the clock source for TMR0.
1 = Internal instruction clock cycle (TCY)
0 = T0CKI pin

bit 4-1: **PS3:PS0**: Timer0 Prescale Selection bits
These bits select the prescale value for TMR0.

PS3:PS0	Prescale Value
0000	1:1
0001	1:2
0010	1:4
0011	1:8
0100	1:16
0101	1:32
0110	1:64
0111	1:128
1xxx	1:256

bit 0: **Unimplemented**: read as '0'

11.0 TIMER0

The Timer0 (TMR0) module consists of a 16-bit timer/counter, TMR0. The high byte is TMR0H and the low byte is TMR0L. A software programmable 8-bit prescaler makes an effective 24-bit overflow timer. The clock source is also software programmable as either the internal instruction clock or the RA1/T0CKI pin. The control bits for this module are in register T0STA (Figure 11-1).

11.1 TMR0 Operation

When T0CS is set, TMR0 increments on the internal clock. When T0CS is clear, TMR0 increments on the external clock (RA1/T0CKI pin). The external clock edge can be configured in software. When T0SE is set, the timer will increment on the rising edge of the RA1/T0CKI pin. When T0SE is clear, the timer will increment on the falling edge of the RA1/T0CKI pin. The prescaler can be programmed to introduce a prescale of 1:1 to 1:256. The timer increments from 0000h to FFFFh and rolls over to 0000h. On overflow, the TMR0 Interrupt Flag bit (T0IF) is set. The TMR0 interrupt can be masked off by clearing the corresponding TMR0 Interrupt Enable bit (T0IE). The TMR0 Interrupt Flag bit (T0IF) is automatically cleared when vectoring to the TMR0 interrupt vector.

11.2 Using TMR0 with External Clock

When the external clock input is used for TMR0, it is synchronized with the internal phase clocks. Figure 11-3 shows the synchronization of the external clock. This synchronization is done after the prescaler. The output of the prescaler (PSOUT) is sampled twice in every instruction cycle to detect a rising or a falling edge. The timing requirements for the external clock are detailed in the electrical specification section for the desired device.

11.2.1 DELAY FROM EXTERNAL CLOCK EDGE

Since the prescaler output is synchronized with the internal clocks, there is a small delay from the time the external clock edge occurs to the time TMR0 is actually incremented. Figure 11-3 shows that this delay is between 3 TOSC and 7 TOSC. Thus, for example, measuring the interval between two edges (e.g. period) will be accurate within ±4 TOSC (±160 ns @ 25 MHz).

11.3 Read/Write Consideration for TMR0

Although TMR0 is a 16-bit timer/counter, only 8-bits at a time can be read or written during a single instruction cycle. Care must be taken during any read or write.

11 3.1 READING 16-BIT VALUE

The problem in reading the entire 16-bit value is that after reading the low (or high) byte, its value may change from FFh to 00h.

EXAMPLE 11-1: 16-BIT READ

```
MOVPF   TMR0L, TMPLO     ;read low tmr0
MOVPF   TMR0H, TMPHI     ;read high tmr0
MOVFP   TMPLO, WREG      ;tmplo -> wreg
CPFSLT  TMR0L, WREG      ;tmr0l < wreg?
RETFIE                   ;no then return
MOVPF   TMR0L, TMPLO     ;read low tmr0
MOVPF   TMR0H, TMPHI     ;read high tmr0
RETFIE                   ;return
```

Interrupts must be disabled during this subroutine.

11.3.2 WRITING A 16-BIT VALUE TO TMR0

Since writing to either TMR0L or TMR0H will effectively inhibit increment of that half of the TMR0 in the next cycle (following write), but not inhibit increment of the other half, the user must write to TMR0L first and TMR0H next in two consecutive instructions, as shown in Example 11-2. The interrupt must be disabled. Any write to either TMR0L or TMR0H clears the prescaler.

EXAMPLE 11-2: 16-BIT WRITE

```
BSF    CPUSTA, GLINTD ; Disable interrupt
MOVFP  RAM_L, TMR0L   ;
MOVFP  RAM_H, TMR0H   ;
BCF    CPUSTA, GLINTD ; Done, enable interrupt
```

11.4 Prescaler Assignments

Timer0 has an 8-bit prescaler. The prescaler assignment is fully under software control; i.e., it can be changed "on the fly" during program execution. When changing the prescaler assignment, clearing the prescaler is recommended before changing assignment. The value of the prescaler is "unknown", and assigning a value that is less then the present value makes it difficult to take this unknown time into account.

FIGURE 12-1: TCON1 REGISTER (ADDRESS: 16H, BANK 3)

R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0
CA2ED1	CA2ED0	CA1ED1	CA1ED0	T16	TMR3CS	TMR2CS	TMR1CS
bit7							bit0

R = Readable bit
W = Writable bit
-n = Value at POR reset

bit 7-6: **CA2ED1:CA2ED0**: Capture2 Mode Select bits
00 = Capture on every falling edge
01 = Capture on every rising edge
10 = Capture on every 4th rising edge
11 = Capture on every 16th rising edge

bit 5-4: **CA1ED1:CA1ED0**: Capture1 Mode Select bits
00 = Capture on every falling edge
01 = Capture on every rising edge
10 = Capture on every 4th rising edge
11 = Capture on every 16th rising edge

bit 3: **T16**: Timer1:Timer2 Mode Select bit
1 = Timer1 and Timer2 form a 16-bit timer
0 = Timer1 and Timer2 are two 8-bit timers

bit 2: **TMR3CS**: Timer3 Clock Source Select bit
1 = Timer3 increments off the falling edge of the RB5/TCLK3 pin
0 = Timer3 increments off the internal clock

bit 1: **TMR2CS**: Timer2 Clock Source Select bit
1 = Timer2 increments off the falling edge of the RB4/TCLK12 pin
0 = Timer2 increments off the internal clock

bit 0: **TMR1CS**: Timer1 Clock Source Select bit
1 = Timer1 increments off the falling edge of the RB4/TCLK12 pin
0 = Timer1 increments off the internal clock

TMR3 is a 16-bit timer/counter consisting of the TMR3H and TMR3L registers. This timer has four other associated registers. Two registers are used as a 16-bit period register or a 16-bit Capture1 register (PR3H/CA1H:PR3L/CA1L). The other two registers are strictly the Capture2 registers (CA2H:CA2L). Timer3 is the time-base for the two 16-bit captures.

Timer3 can be software configured to increment from the internal system clock or from an external signal on the RB5/TCLK3 pin.

Figure 12-1 and Figure 12-2 are the control registers for the operation of Timer1, Timer2, and Timer3, as well as PWM1, PWM2, Capture1, and Capture2.

12.0 TIMER1, TIMER2, TIMER3, PWMS AND CAPTURES

The PIC17C4X has a wealth of timers and time-based functions to ease the implementation of control applications. These time-base functions include two PWM outputs and two Capture inputs.

Timer1 (TMR1) and Timer2 (TMR2) are two 8-bit incrementing timers, each with a period register (PR1 and PR2 respectively) and separate overflow interrupt flags. TMR1 and TMR2 can operate either as timers (increment on internal OSC/4 clock) or as counters (increment on falling edge of external clock on pin RB4/TCLK12). They are also software configurable to operate as a single 16-bit timer. These timers are also used as the time-base for the PWM (pulse width modulation) module.

FIGURE 12-2: TCON2 REGISTER (ADDRESS: 17H, BANK 3)

R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0
CA2OVF	CA1OVF	PWM2ON	PWM1ON	CA1/PR3	TMR3ON	TMR2ON	TMR1ON
bit7							bit0

R = Readable bit
W = Writable bit
-n = Value at POR reset

bit 7: **CA2OVF**: Capture2 Overflow Status bit
This bit indicates that the capture value had not been read from the capture register pair (CA2H:CA2L before the next capture event occured. The capture register retains the oldest unread capture value (las capture before overflow). Subsequent capture events will not update the capture register with the Timer value until the capture register has been read (both bytes) and this bit is cleared in software.
1 = Overflow occurred on Capture2 register
0 = No overflow occurred on Capture2 register

bit 6: **CA1OVF**: Capture1 Overflow Status bit
This bit indicates that the capture value had not been read from the capture register pai (PR3H/CA2H:PR3L/CA2L) before the next capture event occured. The capture register retains the olde unread capture value (last capture before overflow). Subsequent capture events will not update the cap ture register with the Timer3 value until the capture register has been read (both bytes) and this bit is cleared in software.
1 = Overflow occurred on Capture1 register
0 = No overflow occurred on Capture1 register

bit 5: **PWM2ON**: PWM2 On bit
1 = PWM2 is enabled (The RB3/PWM2 pin ignores the state of the DDRB<3> bit)
0 = PWM2 is disabled (The RB3/PWM2 pin uses the state of the DDRB<3> bit for data direction)

bit 4: **PWM1ON**: PWM1 On bit
1 = PWM1 is enabled (The RB2/PWM1 pin ignores the state of the DDRB<2> bit)
0 = PWM1 is disabled (The RB2/PWM1 pin uses the state of the DDRB<2> bit for data direction)

bit 3: **CA1/PR3**: CA1/PR3 Register Mode Select bit
1 = Enables Capture1 (PR3H/CA1H:PR3L/CA1L is the Capture1 register. Timer3 runs without a period register)
0 = Enables the Period register (PR3H/CA1H:PR3L/CA1L is the Period register for Timer3)

bit 2: **TMR3ON**: Timer3 On bit
1 = Starts Timer3
0 = Stops Timer3

bit 1: **TMR2ON**: Timer2 On bit
This bit controls the incrementing of the Timer2 register. When Timer2:Timer1 form the 16-bit timer (T16 is set), TMR2ON must be set. This allows the MSB of the timer to increment.
1 = Starts Timer2 (Must be enabled if the T16 bit (TCON1<3>) is set)
0 = Stops Timer2

bit 0: **TMR1ON**: Timer1 On bit
When T16 is set (in 16-bit Timer Mode)
1 = Starts 16-bit Timer2:Timer1
0 = Stops 16-bit Timer2:Timer1

When T16 is clear (in 8-bit Timer Mode)
1 = Starts 8-bit Timer1
0 = Stops 8-bit Timer1

12.1 Timer1 and Timer2

12.1.1 TIMER1, TIMER2 IN 8-BIT MODE

Both Timer1 and Timer2 will operate in 8-bit mode when the T16 bit is clear. These two timers can be independently configured to increment from the internal instruction cycle clock or from an external clock source on the RB4/TCLK12 pin. The timer clock source is configured by the TMRxCS bit (x = 1 for Timer1 or = 2 for Timer2). When TMRxCS is clear, the clock source is internal and increments once every instruction cycle (OSC/4). When TMRxCS is set, the clock source is the RB4/TCLK12 pin, and the timer will increment on every falling edge of the RB4/TCLK12 pin.

The timer increments from 00h until it equals the Period register (PRx). It then resets to 00h at the next increment cycle. The timer interrupt flag is set when the timer is reset. Timer1 and Timer2 have individual interrupt flag bits. The Timer1 interrupt flag bit is latched into TMR1IF, and the Timer2 interrupt flag bit is latched into TMR2IF.

Each timer also has a corresponding interrupt enable bit (TMRxIE). The timer interrupt can be enabled by setting this bit and disabled by clearing this bit. For peripheral interrupts to be enabled, the Peripheral Interrupt Enable bit must be enabled (PEIE is set) and global interrupts must be enabled (GLINTD is cleared).

The timers can be turned on and off under software control. When the Timerx on control bit (TMRxON) is set, the timer increments from the clock source. When TMRxON is cleared, the timer is turned off and cannot cause the timer interrupt flag to be set.

12.1.1.1 EXTERNAL CLOCK INPUT FOR TMR1 OR TMR2

When TMRxCS is set, the clock source is the RB4/TCLK12 pin, and the timer will increment on every falling edge on the RB4/TCLK12 pin. The TCLK12 input is synchronized with internal phase clocks. This causes a delay from the time a falling edge appears on TCLK12 to the time TMR1 or TMR2 is actually incremented. For the external clock input timing requirements, see the Electrical Specification section.

12.1.2.1 EXTERNAL CLOCK INPUT FOR TMR1:TMR2

When TMR1CS is set, the 16-bit TMR2:TMR1 increments on the falling edge of clock input TCLK12. The input on the RB4/TCLK12 pin is sampled and synchronized by the internal phase clocks twice every instruction cycle. This causes a delay from the time a falling edge appears on RB4/TCLK12 to the time TMR2:TMR1 is actually incremented. For the external clock input timing requirements, see the Electrical Specification section.

TABLE 12-1: TURNING ON 16-BIT TIMER

TMR2ON	TMR1ON	Result
1	1	16-bit timer (TMR2:TMR1) ON
0	1	Only TMR1 increments
x	0	16-bit timer OFF

12.1.2 TIMER1 & TIMER2 IN 16-BIT MODE

To select 16-bit mode, the T16 bit must be set. In this mode TMR1 and TMR2 are concatenated to form a 16-bit timer (TMR2:TMR1). The 16-bit timer increments until it matches the 16-bit period register (PR2:PR1). On the following timer clock, the timer value is reset to 0h, and the TMR1IF bit is set.

When selecting the clock source for the16-bit timer, the TMR1CS bit controls the entire 16-bit timer and TMR2CS is a "don't care". When TMR1CS is clear, the timer increments once every instruction cycle (OSC/4). When TMR1CS is set, the timer increments on every falling edge of the RB4/TCLK12 pin. For the 16-bit timer to increment, both TMR1ON and TMR2ON bits must be set (see Table 12-1).

12.1.3 USING PULSE WIDTH MODULATION (PWM) OUTPUTS WITH TMR1 AND TMR2

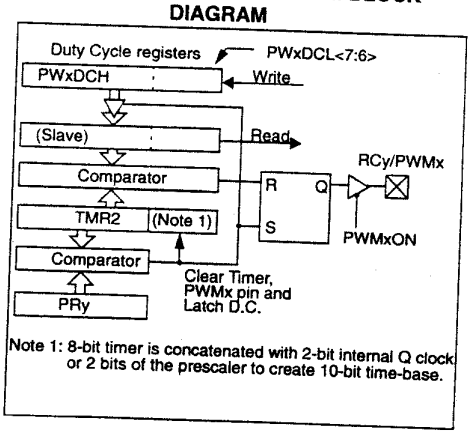
Two high speed pulse width modulation (PWM) outputs are provided. The PWM1 output uses Timer1 as its time-base, while PWM2 may be software configured to use either Timer1 or Timer 2 as the time-base. The PWM outputs are on the RB2/PWM1 and RB3/PWM2 pins.

Each PWM output has a maximum resolution of 10-bits. At 10-bit resolution, the PWM output frequency is 24.4 kHz (@ 25 MHz clock) and at 8-bit resolution the PWM output frequency is 97.7 kHz. The duty cycle of the output can vary from 0% to 100%.

Figure 12-5 shows a simplified block diagram of the PWM module. The duty cycle register is double buffered for glitch free operation. Figure 12-6 shows how a glitch could occur if the duty cycle registers were not double buffered.

The user needs to set the PWM1ON bit (TCON2<4>) to enable the PWM1 output. When the PWM1ON bit is set, the RB2/PWM1 pin is configured as PWM1 output and forced as an output irrespective of the data direction bit (DDRB<2>). When the PWM1ON bit is clear, the pin behaves as a port pin and its direction is controlled by its data direction bit (DDRB<2>). Similarly, the PWM2ON (TCON2<5>) bit controls the configuration of the RB3/PWM2 pin.

FIGURE 12-5: SIMPLIFIED PWM BLOCK DIAGRAM



Note 1: 8-bit timer is concatenated with 2-bit internal Q clock or 2 bits of the prescaler to create 10-bit time-base.

12.2 Timer3

TMR3 is a 16-bit timer consisting of the TMR3H and TMR3L registers. TMR3H is the high byte of the timer and TMR3L is the low byte. This timer has an associated 16-bit period register (PR3H/CA1H:PR3L/CA1L). This period register can be software configured to be a second 16-bit capture register.

When the TMR3CS bit (TCON1<2>) is clear, the timer increments every instruction cycle (OSC/4). When TMR3CS is set, the timer increments on every falling edge of the RB5/TCLK3 pin. In either mode, the TMR3ON bit must be set for the timer to increment. When TMR3ON is clear, the timer will not increment or set the TMR3IF bit.

TMR3 has two modes of operation, depending on the CA1/$\overline{PR3}$ bit (TCON2<3>). These modes are:

- One capture and one period register mode
- Dual capture register mode

The PIC17C4X has up to two 16-bit capture registers that capture the 16-bit value of TMR3 when events are detected on capture pins. There are two capture pins (RB0/CAP1 and RB1/CAP2), one for each capture register. The capture pins are multiplexed with PORTB pins. An event can be:

- a rising edge
- a falling edge
- 4 rising edges
- 16 rising edges

Each 16-bit capture register has an interrupt flag associated with it. The flag is set when a capture is made. The capture module is truly part of the Timer3 block. Figure 12-7 and Figure 12-8 show the block diagrams for the two modes of operation.

12.2.1 ONE CAPTURE AND ONE PERIOD REGISTER MODE

In this mode registers PR3H/CA1H and PR3L/CA1L constitute a 16-bit period register. A block diagram is shown in Figure 12-7. The timer increments until it equals the period register and then resets to 0000h. TMR3 Interrupt Flag bit (TMR3IF) is set at this point. This interrupt can be disabled by clearing the TMR3 Interrupt Enable bit (TMR3IE). TMR3IF must be cleared in software.

This mode is selected if control bit CA1/PR3 is clear. In this mode, the Capture1 register, consisting of high byte (PR3H/CA1H) and low byte (PR3L/CA1L), is configured as the period control register for TMR3. Capture1 is disabled in this mode, and the corresponding Interrupt bit CA1IF is never set. Timer3 increments until it equals the value in the period register and then resets to 0000h.

Capture2 is active in this mode. The CA2ED1 and CA2ED0 bits determine the event on which capture will occur. The possible events are:

- Capture on every falling edge
- Capture on every rising edge
- Capture every 4th rising edge
- Capture every 16th rising edge

When a capture takes place, an interrupt flag is latched into the CA2IF bit. This interrupt can be enabled by setting the corresponding mask bit CA2IE. The Peripheral Interrupt Enable bit (PEIE) must be set and the Global Interrupt Disable bit (GLINTD) must be cleared for the interrupt to be acknowledged. The CA2IF interrupt flag bit must be cleared in software.

When the capture prescale select is changed, the prescaler is not reset and an event may be generated. Therefore, the first capture after such a change will be ambiguous. However, it sets the time-base for the next capture. The prescaler is reset upon chip reset.

Capture pin RB1/CAP2 is a multiplexed pin. When used as a port pin, Capture2 is not disabled. However, the user can simply disable the Capture2 interrupt by clearing CA2IE. If RB1/CAP2 is used as an output pin, the user can activate a capture by writing to the port pin. This may be useful during development phase to emulate a capture interrupt.

The input on capture pin RB1/CAP2 is synchronized internally to internal phase clocks. This imposes certain restrictions on the input waveform (see the Electrical Specification section for timing).

The Capture2 overflow status flag bit is double buffered. The master bit is set if one captured word is already residing in the Capture2 register and another "event" has occurred on the RB1/CA2 pin. The new event will not transfer the Timer3 value to the capture register, protecting the previous unread capture value. When the user reads both the high and the low bytes (in any order) of the Capture2 register, the master overflow bit is transferred to the slave overflow bit (CA2OVF) and then the master bit is reset. The user can then read TCON2 to determine the value of CA2OVF.

The recommended sequence to read capture registers and capture overflow flag bits is shown in Example 12-1.

EXAMPLE 12-1: SEQUENCE TO READ CAPTURE REGISTERS

```
MOVLB  3                ; Select Bank 3
MOVPF  CA2L, LO_BYTE    ; Read Capture2 low
                        ; byte, store in LO_BYTE
MOVPF  CA2H, HI_BYTE    ; Read Capture2 high
                        ; byte, store in HI_BYTE
MOVPF  TCON2, STAT_VAL  ; Read TCON2 into file
                        ; STAT_VAL
```

12.2.2 DUAL CAPTURE1 REGISTER MODE

This mode is selected by setting CA1/$\overline{PR3}$. A block diagram is shown in Figure 12-8. In this mode, TMR3 runs without a period register and increments from 0000h to FFFFh and rolls over to 0000h. The Timer3 interrupt Flag (TMR3IF) is set on this roll over. The TMR3IF bit must be cleared in software.

Registers PR3H/CA1H and PR3L/CA1L make a 16-bit capture register (Capture1). It captures events on pin RB0/CAP1. Capture mode is configured by the CA1ED1 and CA1ED0 bits. Capture1 Interrupt Flag bit (CA1IF) is set on the capture event. The corresponding interrupt mask bit is CA1IE. The Capture1 overflow status bit is CA1OVF.

The Capture2 overflow status flag bit is double buffered. The master bit is set if one captured word is already residing in the Capture2 register and another "event" has occurred on the RB1/CA2 pin. The new event will not transfer the Timer3 value to the capture register which protects the previous unread capture value. When the user reads both the high and the low bytes (in any order) of the Capture2 register, the master overflow bit is transferred to the slave overflow bit (CA2OVF) and then the master bit is reset. The user can then read TCON2 to determine the value of CA2OVF.

The operation of the Capture1 feature is identical to Capture2 (as described in Section 12.2.1).

12.2.3 EXTERNAL CLOCK INPUT FOR TIMER3

When TMR3CS is set, the 16-bit TMR3 increments on the falling edge of clock input TCLK3. The input on the RB5/TCLK3 pin is sampled and synchronized by the internal phase clocks twice every instruction cycle. This causes a delay from the time a falling edge appears on TCLK3 to the time TMR3 is actually incremented. For the external clock input timing requirements, see the Electrical Specification section. Figure 12-9 shows the timing diagram when operating from an external clock.

12.2.4 READING/WRITING TIMER3

Since Timer3 is a 16-bit timer and only 8-bits at a time can be read or written, care should be taken when reading or writing while the timer is running. The best method to read or write the timer is to stop the timer, perform any read or write operation, and then restart Timer3 (using the TMR3ON bit). However, if it is necessary to keep Timer3 free-running, care must be taken. For writing to the 16-bit Timer3, Example 12-2 may be used. For reading the 16-bit Timer3, Example 12-3 may be used.

EXAMPLE 12-2: WRITING TO TMR3

```
BSF   CPUSTA, GLINTD ; Disable interrupt
MOVFP RAM_L, TMR3L   ;
MOVFP RAM_H, TMR3H   ;
BCF   CPUSTA, GLINTD ; Done, enable interrupt
```

EXAMPLE 12-3: READING FROM TMR3

```
MOVPF   TMR3L, TMPLO    ;read low tmr0
MOVPF   TMR3H, TMPHI    ;read high tmr0
MOVFP   TMPLO, WREG     ;tmplo -> wreg
CPFSLT  TMR3L, WREG     ;tmr0l < wreg?
RETFIE                  ;no then return
MOVPF   TMR3L, TMPLO    ;read low tmr0
MOVPF   TMR3H, TMPHI    ;read high tmr0
RETFIE                  ;return
```

Interrupts must be disabled during this subroutine.

13.0 SERIAL COMMUNICATION INTERFACE (SCI) MODULE

The Serial Communication Interface (SCI) module is a serial I/O module. The SCI (USART) can be configured as a full duplex asynchronous system that can communicate with peripheral devices such as CRT terminals and personal computers, or it can be configured as a half duplex synchronous system that can communicate with peripheral devices such as A/D or D/A integrated circuits, Serial EEPROMs etc. The SCI can be configured in the following modes:

- Asynchronous (full duplex)
- Synchronous - Master (half duplex)
- Synchronous - Slave (half duplex)

The SPEN (RCSTA<7>) bit has to be set in order to configure RC6 and RC7 as the Serial Communication Interface.

The SCI module will control the direction of the RA4/RX/DT and RA5/TX/CK pins, depending on the states of the SCI configuration bits in the RCSTA and TXSTA registers. The bits that control I/O direction are:

- SPEN
- TXEN
- SREN
- CREN
- CSRC

The Transmit Status And Control Register is shown in Figure 13-1, while the Receive Status And Control Register is shown in Figure 13-2.

FIGURE 13-1: TXSTA REGISTER (ADDRESS: 15H, BANK 0)

R/W - 0	R/W - 0	R/W - 0	R/W - 0	U - 0	U - 0	R - 1	R/W - x
CSRC	TX8/9	TXEN	SYNC	—	—	TRMT	TXD8
bit7							bit0

R = Readable bit
W = Writable bit
-n = Value at POR reset
(x = unknown)

bit 7: **CSRC**: Clock Source Select bit
Synchronous mode:
1 = Master Mode (Clock generated internally from BRG)
0 = Slave mode (Clock from external source)
Asynchronous mode:
Don't care

bit 6: **TX8/9**: Transmit Data Length bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5: **TXEN**: Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled
SREN/CREN overrides TXEN in SYNC mode

bit 4: **SYNC**: SCI mode Select bit
(Synchronous/Asynchronous)
1 = Synchronous mode
0 = Asynchronous mode

bit 3-2: **Unimplemented**, reads as '0'

bit 1: **TRMT**: Transmit Shift Register (TSR) Empty bit
1 = TSR empty
0 = TSR full

bit 0: **TXD8**: 9th bit of transmit data (can be used to calculated the parity in software)

14.0 SPECIAL FEATURES OF THE CPU

What sets a microcontroller apart from other processors are special circuits to deal with the needs of real time applications. The PIC17CXX family has a host of such features intended to maximize system reliability, minimize cost through elimination of external components, provide power saving operating modes and offer code protection. These are:

- OSC selection
- Reset
  - Power-On Reset (POR)
  - Power-Up Timer (PWRT)
  - Oscillator Start-Up Timer (OST)
- Interrupts
- Watchdog Timer (WDT)
- SLEEP
- Code protection

The PIC17CXX has a Watchdog Timer which can be shut off only through EPROM bits. It runs off its own RC oscillator for added reliability. There are two timers that offer necessary delays on power-up. One is the Oscillator Start-Up Timer (OST), intended to keep the chip in RESET until the crystal oscillator is stable. The other is the Power-Up Timer (PWRT), which provides a fixed delay of 96 ms (nominal) on power up only, designed to keep the part in RESET while the power supply stabilizes. With these two timers on-chip, most applications need no external reset circuitry.

The SLEEP mode is designed to offer a very low current power-down mode. The user can wake from SLEEP through external reset, watchdog timer time-out or through an interrupt. Several oscillator options are also made available to allow the part to fit the application. The RC oscillator option saves system cost while the LF crystal option saves power. Configuration bits are used to select various options. This configuration word has the format shown in Figure 14-1.

FIGURE 14-1: CONFIGURATION WORD

U - x	U - x	U - x	U - x	U - x	U - x	U - x	R/P - 1
—	—	—	—	—	—	—	PM2 †

bit15-7 bit0

U - x	R/P - 0	U - x	R/P - 1	R/P - 1	R/P - 1	R/P - 1	R/P - 1
—	PM1	—	PM0	WDTPS1	WDTPS0	FOSC1	FOSC0

bit15-7 bit0

R = Readable bit
P = Programmable bit
U = Unimplemented
- n = Value for Erased Device
(x = unknown)

bit 15-9: **Unimplemented**, reads as a '1'

bit 8,6,4: **PM2, PM1, PM0**, Processor Mode Select bits
111 =Microprocessor Mode
110 =Extended microcontroller mode
101 =Microcontroller mode
000 =Code protected microcontroller mode

bit 7, 5: **Unimplemented**, reads as a '0'

bit 3-2: **WDTPS1:WDTPS0**, WDT Postscaler Select bits
11 = WDT enabled, postscaler = 1
10 = WDT enabled, postscaler = 256
01 = WDT enabled, postscaler = 64
00 = WDT disabled, 16-bit overflow timer

bit 1-0: **FOSC1:FOSC0**, Oscillator Select bits
11 = EC oscillator
10 = XT oscillator
01 = RC oscillator
00 = LF oscillator

† This bit does not exist on the PIC17C42. Reading this bit will return an unknown value (x).

14.1 Configuration Bits

The PIC17CXX has seven configuration locations (see Table 14-1). These locations can be programmed (read as '0') or left unprogrammed (read as '1') to select various device configurations. Any write to a configuration location, regardless of the data, will program that configuration bit. A TABLWT instruction is required to write to program memory locations. The configuration bits can be read by using the TABLRD instructions. Reading any configuration location between FE00h and FE07h will read the low byte of the configuration word (see Figure 14-1) into the TABLATL register. The TABLATH register will be FFh. Reading a configuration location between FE08h and FE0Fh will read the high byte of the configuration word into the TABLATL register. The TABLATH register will be FFh.

Addresses FE00h thru FE0Fh are only in the program memory space for microcontroller and code protected microcontroller modes. A device programmer will be able to read the configuration word in any processor mode. See programming specifications for more detail.

TABLE 14-1: CONFIGURATION LOCATIONS

Bit	Address
FOSC0	FE00h
FOSC1	FE01h
WDTPS0	FE02h
WDTPS1	FE03h
PM0	FE04h
PM1	FE06h
PM2 †	FE08h †

† This location does not exist on the PIC17C42.

Note: When programming the desired configuration locations, they must be programmed in ascending order. Starting with address FE00h.

14.2 Oscillator Configurations

14.2.1 OSCILLATOR TYPES

The PIC17CXX can be operated in four different oscillator modes. The user can program two configuration bits (FOSC1:FOSC0) to select one of these four modes:

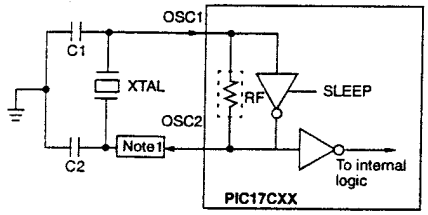
- LF: Low Power Crystal
- XT: Crystal/Resonator
- EC: External Clock Input
- RC: Resistor/Capacitor

14.2.2 CRYSTAL OSCILLATOR / CERAMIC RESONATORS

In XT or LF modes, a crystal or ceramic resonator is connected to the OSC1/CLKIN and OSC2/CLKOUT pins to establish oscillation (Figure 14-2). The PIC17CXX Oscillator design requires the use of a parallel cut crystal. Use of a series cut crystal may give a frequency out of the crystal manufacturers specifications.

For frequencies above 20 MHz, it is common for the crystal to be an overtone mode crystal. Use of overtone mode crystals require a tank circuit to attenuate the gain at the fundamental frequency. Figure 14-3 shows an example of this.

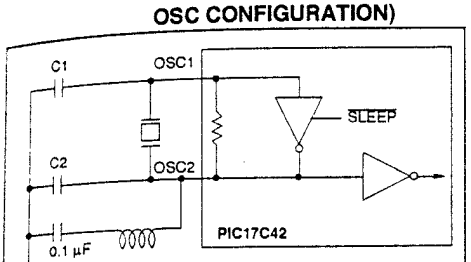
FIGURE 14-2: CRYSTAL OR CERAMIC RESONATOR OPERATION (XT OR LF OSC CONFIGURATION)



See Table 14-2 and Table 14-3 for recommended values of C1 and C2.

Note 1: A series resistor may be required for AT strip cut crystals.

FIGURE 14-3: CRYSTAL OPERATION, OVERTONE CRYSTALS (XT OSC CONFIGURATION)



To filter the fundamental frequency

$$\frac{1}{LC2} = (2\pi f)^2$$

Where f = tank circuit resonant frequency. This should be midway between the fundamental and the 3rd overtone frequencies of the crystal.

TABLE 14-2: CAPACITOR SELECTION FOR CERAMIC RESONATORS

Oscillator Type	Resonator Frequency	Capacitor Range C1 = C2
LF	455 kHz	15 - 68 pF
	2.0 MHz	10 - 33 pF
XT	4.0 MHz	22 - 68 pF
	8.0 MHz	33 - 100 pF
	16.0 MHz	33 - 100 pF

Higher capacitance increases the stability of the oscillator but also increases the start-up time. These values are for design guidance only. Since each resonator has its own characteristics, the user should consult the resonator manufacturer for appropriate values of external components.

Resonators Used:

455 kHz	Panasonic EFO-A455K04B	+/-0.3%
2.0 MHz	Murata Erie CSA2.00MG	+/-0.5%
4.0 MHz	Murata Erie CSA4.00MG	+/-0.5%
8.0 MHz	Murata Erie CSA8.00MT	+/-0.5%
16.0 MHz	Murata Erie CSA16.00MX	+/-0.5%

Resonators used did not have built-in capacitors.

TABLE 14-3: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR

Osc Type	Freq	C1	C2
LF	32 kHz[1]	100-150 pF	100-150 pF
	1 MHz	10-33 pF	10-33 pF
	2 MHz	10-33 pF	10-33 pF
XT	2 MHz	47-100 pF	47-100 pF
	4 MHz	15-68 pF	15-68 pF
	8 MHz [2]	15-47 pF	15-47 pF
	16 MHz	TBD	TBD
	25 MHz	15-47 pF	15-47 pF

Higher capacitance increases the stability of the oscillator but also increases the start-up time and the oscillator current. These values are for design guidance only. Rs may be required in XT mode to avoid overdriving the crystals with low drive level specification. Since each crystal has its own characteristics, the user should consult the crystal manufacturer for appropriate values for external components.

Note 1: For VDD > 4.5V, C1 = C2 ≈ 30 pf is recommended.

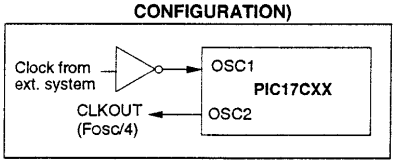
2: Rs of 330Ω is required for a capacitor combination of 15/15 pF.

Crystals Used:		
32.768 kHz	Epson C-001R32.768K-A	± 20 PPM
1.0 MHz	ECS ECS-10-13-2	± 50 PPM
2.0 MHz	ECS ECS-20-S-1	± 50 PPM
4.0 MHz	ECS ECS-40-S-4	± 50 PPM
8.0 MHz	ECS ECS-80-S-4	± 50 PPM
16.0 MHz	TBD	TBD
25 MHz	CTS CTS25M	± 50 PPM

14.2.3 EXTERNAL CLOCK OSCILLATOR

In the EC oscillator mode, the OSC1 input can be driven by CMOS drivers. In this mode, the OSC1/CLKIN pin is hi-impedance and the OSC2/CLKOUT pin is the CLKOUT output (4 Tosc).

FIGURE 14-4: EXTERNAL CLOCK INPUT OPERATION (EC OSC CONFIGURATION)



14.2.4 EXTERNAL CRYSTAL OSCILLATOR CIRCUIT

Either a prepackaged oscillator can be used or a simple oscillator circuit with TTL gates can be built. Prepackaged oscillators provide a wide operating range and better stability. A well-designed crystal oscillator will provide good performance with TTL gates. Two types of crystal oscillator circuits can be used: one with series resonance, or one with parallel resonance.

Figure 14-5 shows implementation of a parallel resonant oscillator circuit. The circuit is designed to use the fundamental frequency of the crystal. The 74AS04 inverter performs the 180-degree phase shift that a parallel oscillator requires. The 4.7 kΩ resistor provides the negative feedback for stability. The 10 kΩ potentiometer biases the 74AS04 in the linear region. This could be used for external oscillator designs.

FIGURE 14-5: EXTERNAL PARALLEL RESONANT CRYSTAL OSCILLATOR CIRCUIT

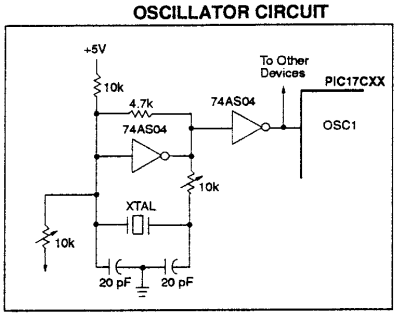
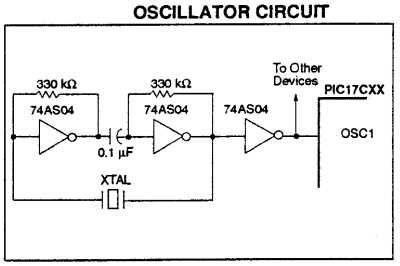


Figure 14-6 shows a series resonant oscillator circuit. This circuit is also designed to use the fundamental frequency of the crystal. The inverter performs a 180-degree phase shift in a series resonant oscillator circuit. The 330 kΩ resistors provide the negative feedback to bias the inverters in their linear region.

FIGURE 14-6: EXTERNAL SERIES RESONANT CRYSTAL OSCILLATOR CIRCUIT

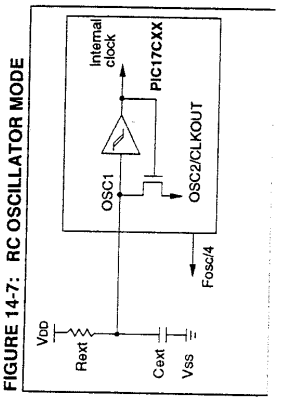


14.2.5 RC OSCILLATOR

For timing insensitive applications, the RC device option offers additional cost savings. RC oscillator frequency is a function of the supply voltage, the resistor (Rext) and capacitor (Cext) values, and the operating temperature. In addition to this, oscillator frequency will vary from unit to unit due to normal process parameter variation. Furthermore, the difference in lead frame capacitance between package types will also affect oscillation frequency, especially for low Cext values. The user also needs to take into account variation due to tolerance of external R and C components used. Figure 14-6 shows how the R/C combination is connected to the PIC17CXX. For Rext values below 2.2 kΩ, the oscillator operation may become unstable, or stop completely. For very high Rext values (e.g. 1 MΩ), the oscillator becomes sensitive to noise, humidity and leakage. Thus, we recommend to keep Rext between 3 kΩ and 100 kΩ.

Although the oscillator will operate with no external capacitor (Cext = 0 pF), we recommend using values above 20 pF for noise and stability reasons. With little or no external capacitance, oscillation frequency can vary dramatically due to changes in external capacitances, such as PCB trace capacitance or package lead frame capacitance.

FIGURE 14-7: RC OSCILLATOR MODE



See Section 18.0 for RC frequency variation from part to part due to normal process variation. The variation is larger for larger R (since leakage current variation will affect RC frequency more for large R) and for smaller C (since variation of input capacitance will affect RC frequency more).

See Section 18.0 for variation of oscillator frequency due to VDD for given Rext/Cext values as well as frequency variation due to operating temperature for given R, C, and VDD values.

The oscillator frequency, divided by 4, is available on the OSC2/CLKOUT pin, and can be used for test purposes or to synchronize other logic (see Figure 3-2 for waveform).

14.3 Watchdog Timer (WDT)

The Watchdog Timer's function is to recover from software malfunction. The WDT uses an internal free running on-chip RC oscillator for its clock source. This does not require any external components. This RC oscillator is separate from the RC oscillator of the OSC1/CLKIN pin. That means that the WDT will run, even if the clock on the OSC1/CLKIN and OSC2/CLKOUT pins of the device has been stopped, for example, by execution of a SLEEP instruction. During normal operation and SLEEP mode, a WDT time-out generates a device RESET. The WDT can be permanently disabled by programming the configuration bits WDTPS1:WDTPS0 as '00' (Section 14.1).

Under normal operation, the WDT must be cleared on a regular interval. This time is less the the minimum WDT overflow time. Not clearing the WDT in this timeframe will cause the WDT to overflow and reset the device.

14.3.1 WDT PERIOD

The WDT has a nominal time-out period of 12 ms, (with postscaler = 1). The time-out periods vary with temperature, VDD and process variations from part to part (see DC specs). If longer time-out periods are desired, a postscaler with a division ratio of up to 1:256 can be assigned to the WDT. Thus, typical time-out periods up to 3.0 seconds can be realized.

The CLRWDT and SLEEP instructions clear the WDT and the postscaler (if assigned to the WDT) and prevent it from timing out thus generating a device RESET condition.

The $\overline{TO}$ bit in the STATUS register will be cleared upon a WDT time-out.

14.3.2 CLEARING THE WDT AND POSTSCALER

The WDT and postscaler are cleared when:

- The device is in the reset state
- A SLEEP instruction is executed
- A CLRWDT instruction is executed
- Wake-up from SLEEP by an interrupt

The WDT counter/postscaler will start counting on the first edge after the device exits the reset state.

14.3.3 WDT PROGRAMMING CONSIDERATIONS

It should also be taken in account that under worst case conditions (VDD = Min., Temperature = Max., max. WDT postscaler) it may take several seconds before a WDT time-out occurs.

The WDT and postscaler is the Power-Up timer during the Power-On Reset sequence.

14.3.4 WDT AS NORMAL TIMER

When the WDT is selected as a normal timer, the clock source is the device clock. Neither the WDT nor the postscaler are directly readable or writable. The overflow time is 65536 TOSC cycles. On overflow, the $\overline{TO}$ bit is cleared (device is not reset). The CLRWDT instruction can be used to set the $\overline{TO}$ bit. This allows this timer to be a simple overflow timer. When in sleep, this timer is stopped.

FIGURE 14-8: WATCHDOG TIMER BLOCK DIAGRAM

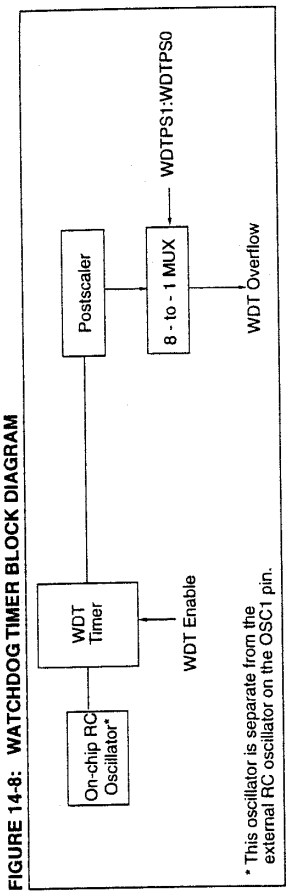


TABLE 14-4: REGISTERS/BITS ASSOCIATED WITH THE WATCHDOG TIMER

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-On Reset	Value on all other resets (Note1)
—	Config	—	PM3	—	PM0	WDTPS1	WDTPS0	FOSC1	FOSC0	(Note 3)	(Note 3)
06h, Unbanked	CPUSTA	—	—	STKAV	GLINTD	$\overline{TO}$	$\overline{PD}$	—	—	--11 11--	--11 ??--

Legend: - = Unimplemented, read as '0', ? - Value depends on condition.
Note 1: Other (non power-up) resets include: external reset through $\overline{MCLR}$ and Watchdog Timer time-out reset.
2: Shaded cells are not used by the WDT.
3: This value will be as the device was programmed, or if unprogrammed, will read as all '1's.

14.4 Power-Down Mode (SLEEP)

The Power-Down mode is entered by executing a SLEEP instruction. This clears the Watchdog Timer and postscaler (if enabled). The $\overline{PD}$ bit is cleared and the $\overline{TO}$ bit is set (in the CPUSTA register). In sleep mode, the oscillator driver is turned off. The I/O ports maintain their status (driving high, low, or hi-impedance).

The $\overline{MCLR}$/VPP pin must be at a logic high level (VIHMC). A WDT time-out RESET does not drive the $\overline{MCLR}$/VPP pin low.

14.4.1 WAKE-UP FROM SLEEP

The device can wake up from SLEEP through one of the following events:

- A POR reset
- External reset input on $\overline{MCLR}$/VPP pin
- WDT time-out reset (if WDT was enabled)
- Interrupt from RA0/INT pin, RB port change, T0CKI interrupt, or some Peripheral Interrupts

The following peripheral interrupts can wake-up from SLEEP:

- Capture1 interrupt
- Capture2 interrupt
- SCI synchronous slave transmit interrupt
- SCI synchronous slave receive interrupt

Other peripherals can not generate interrupts since during SLEEP, no on-chip Q clocks are present.

Any reset event will cause a device reset. Any interrupt event is considered a continuation of program execution. The $\overline{TO}$ and $\overline{PD}$ bits in the CPUSTA register can be used to determine the cause of device reset. The $\overline{PD}$ bit, which is set on power-up, is cleared when SLEEP is invoked. The $\overline{TO}$ bit is cleared if WDT time-out occurred (and caused wake-up).

When the SLEEP instruction is being executed, the next instruction (PC + 1) is pre-fetched. For the device to wake-up through an interrupt event, the corresponding interrupt enable bit must be set (enabled). Wake-up is regardless of the state of the GLINTD bit. If the GLINTD bit is set (disabled), the device continues execution at the instruction after the SLEEP instruction. If the GLINTD bit is clear (enabled), the device executes the instruction after the SLEEP instruction and then branches to the interrupt vector address. In cases where the execution of the instruction following SLEEP is not desirable, the user should have a NOP after the SLEEP instruction.

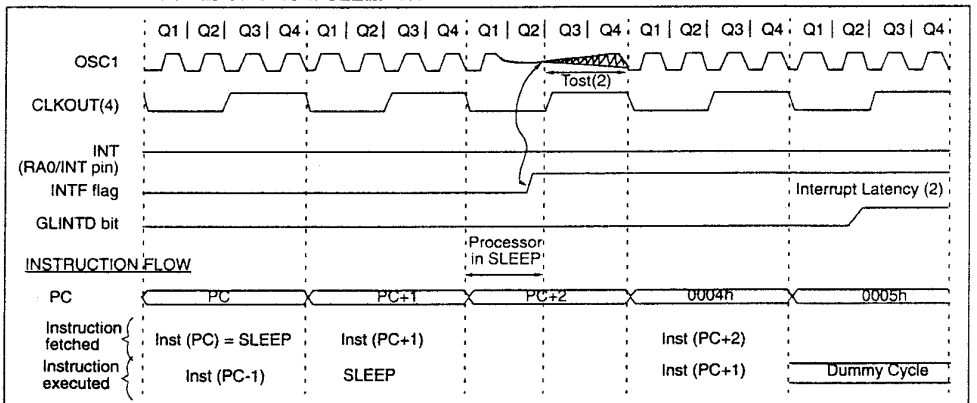
Note: If the global interrupts are disabled (GLINTD is set), but any interrupt source has both its interrupt enable bit and the corresponding interrupt flag bits set, the device will immediately wake-up from sleep. The $\overline{TO}$ bit is set, and the $\overline{PD}$ bit is cleared.

The WDT is cleared when the device wake from SLEEP, regardless of the source of wake-up.

14.4.1.1 WAKE-UP DELAY

When the oscillator type is configured in XT or LF mode, the Oscillator Start-Up Timer (OST) is activated on wake-up. The OST will keep the device in reset for 1024 TOSC. This needs to be taken into account when considering the interrupt response time when coming out of SLEEP.

FIGURE 14-9: WAKE-UP FROM SLEEP THROUGH INTERRUPT



Note 1: XT or LF oscillator mode assumed.
2: Tost = 1024 Tosc (drawing not to scale). This delay will not be there for RC osc mode.
3: When GLINTD = 0 processor jumps to interrupt routine after wake-up. If GLINTD = 1, execution will continue in line.
4: CLKOUT is not available in these osc modes, but shown here for timing reference.

14.4.2 MINIMIZING CURRENT CONSUMPTION

To minimize current consumption, all I/O pins should be either at VDD, or VSS, with no external circuitry drawing current from the I/O pin. I/O pins that are hi-impedance inputs should be pulled high or low externally to avoid switching currents caused by floating inputs. The T0CKI input should be at VDD or VSS. The contributions from on-chip pull-ups on PORTB should also be considered, and disabled when possible.

14.5 Code Protection

The code in the program memory can be protected by selecting the microcontroller in code protected mode (PM2, PM1, PM0 = '000').

Note: PM2 does not exist on the PIC17C42. To select code protected microcontroller mode, PM1, PM0 = '00'.

In this mode, instructions that are in the on-chip program memory space, can continue to read or write the program memory. An instruction that is executed outside of the internal program memory range will be inhibited from writing to or reading from program memory.

Note: Microchip does not recommend code protecting windowed devices. This may inhibit the device from being able to be reprogrammed.

MPASM Quick Reference Guide

This quick reference guide gives all the instructions for the Microchip MPASM Assembler.

MPASM Directive Language Summary

CONTROL DIRECTIVES		
CBANK		Future Feature
CONSTANT	Declare Symbol Constant	constant <label> [= <expr>, ...,<label> [= <expr>]]
#DEFINE	Define a Text Substitution Label	#define <name> [[(<arg>,...,<arg>)]<value>]
END	End Program Block	end
EQU	Define an Assembly Constant	<label> equ <expr>
ERROR	Issue an Error Message	error "<text_string>"
ERRORLEVEL	Set Message Level	errorlevel 0|1|2|<+|-><msg>
INCLUDE	Include Additional Source File	include <<include_file>> include "<include_file>"
LIST	Listing Options	list [<option>,..., <option>]
MESSG	Create User Defined Message	messg "<message_text>"
NOLIST	Turn off Listing Output	nolist
ORG	Set Program Origin	<label> org <expr>
PAGE	Insert Listing Page Eject	page
PROCESSOR	Set Processor Type	processor <processsor_type>
RADIX	Specify Default Radix	radix <default_radix>
SET	Define an Assembler Variable	<label> set <expr>
SPACE	Insert Blank Listing Lines	space [<expr>]
SUBTITLE	Specify Program Subtitle	subtitle "<sub_text>"
TITLE	Specify Program Title	title "<title_text>"
#UNDEFINE	Delete a Substitution Label	#undefine <label>
VARIABLE	Declare Symbol Variable	variable <label> [= <expr>,..., <label> [= <expr>]]
CONDITIONAL ASSEMBLY		
ELSE	Begin Alternative Assembly Block to IF	else
ENDIF	End Conditional Assembly Block	endif
ENDW	End a While Loop	endw
IF	Begin Conditionally Assembled Code Block	if <expr>
IFDEF	Execute If Symbol is Defined	ifdef <label>
IFNDEF	Execute If Symbol is Not Defined	ifndef <label>
WHILE	Perform Loop While Condition is True	while <expr>
DATA		
CBLOCK	Define a Block of Constants	cblock [<expr>]
__CONFIG	Set configuration fuses	__config <expr>
DATA	Create Numeric and Text Data	data <expr>[,<expr>,...,<expr>]
DB	Declare Data of One Byte	db <expr>[,<expr>,...,<expr>]
DE	Define EEPROM Data	de <expr>[,<expr>,...,<expr>]
DT	Define Table	dt <expr>[,<expr>,...,<expr>]
DW	Declare Data of One Word	dw <expr>[,<expr>,...,<expr>]
ENDC	End an Automatic Constant Block	endc
FILL	Specify Memory Fill Value	fill <expr>, <count>
__IDLOCS	Set ID Locations	__idlocs <expr>
RES	Reserve Memory	res <mem_units>
MACROS		
ENDM	End a Macro Definition	endm
EXITM	Exit from a Macro	exitm
EXPAND	Expand Macro Listing	expand
LOCAL	Declare Local Macro Variable	local <label> [, <label>]
MACRO	Declare Macro Definition	<label> macro [<arg>,...,<arg>]
NOEXPAND	Turn off Macro Expansion	noexpand

MPASM Command Line Options

?	N/A	Displays the MPASM Help Panel
a	INHX8M	Generate absolute .COD and hex output directly from assembler: /a<hex-format> where <hex-format> is one of [INHX8M | INHX8S | INHX32]
c	On	Enables/Disables case sensitivity
d	N/A	Define a text string substitution: /d<label>[=<value>]
e	On	Enable/disable/ set path for error file /e Enable /e + Enable /e - Disable /e <path>error.file Enables/sets path
h	N/A	Displays the MPASM Help Panel
l	On	Enable/disable/ set path for list file /l Enable /l + Enable /l - Disable /l <path>list.file Enables/sets path
m	On	Enables/Disable macro expansion
o	Off	Enable/disable/ set path for object file /o Enable /o + Enable /o - Disable /o <path>object.file Enables/sets path
p	None	Set the processor type: /p<processor_type> Where <processor_type> is a member of the PIC16/17 microprocessor family. For example, PIC16C54.
q	Off	Enable/Disable Quiet Mode (suppress screen output)
r	Hex	Defines default radix: /r<radix> where <radix> is one of [HEX | DEC | OCT]
t	8	Set list file tab size
w	0	Set message level: /w<value> Where <value> is: 0: all messages 1: errors and warnings 2: errors
x	Off	Enable/disable/ set path for cross reference file /x Enable /x + Enable /x - Disable /x <path>xref.file Enables/sets path

Radix Types Supported

Decimal	D'<digits>'	D'100'
Hexadecimal (default)	H'<hex_digits>'	H'9f'
Octal	O'<octal_digits>'	O'777'
Binary	B'<binary_digits>'	B'00111001'
Character	'<character>' A'<Character>'	'C' A'C'

MPASM Arithmetic Operators

$	Current program counter	goto $ + 3
(	Left Parenthesis	1 + (d * 4)
)	Right Parenthesis	(Length + 1) * 256
!	Item NOT (logical complement)	if ! (a - b)
~	Complement	flags = ~flags
-	Negation (2's complement)	-1 * Length
high	Return high byte	movlw high CTR_Table
low	Return low byte	movlw low CTR_Table
*	Multiply	a = b * c
/	Divide	a = b / c
%	Modulus	entry_len = tot_len % 16
+	Add	tot_len = entry_len * 8 + 1
-	Subtract	entry_len = (tot - 1) / 8
<<	Left shift	val = flags << 1
>>	Right shift	val = flags >> 1
>=	Greater or equal	if entry_idx >= num_entries
>	Greater than	if entry_idx > num_entries
<	Less than	if entry_idx < num_entries
<=	Less or equal	if entry_idx <= num_entries
==	Equal to	if entry_idx == num_entries
!=	Not equal to	if entry_idx != num_entries
&	Bitwise AND	flags = flags & ERROR_BIT
^	Bitwise exclusive OR	flags = flags ^ ERROR_BIT
|	Bitwise inclusive OR	flags = flags | ERROR_BIT
&&	Logical AND	if (len == 512) && (b == c)
||	Logical OR	if (len == 512) || (b == c)
=	Set equal to	entry_index = 0
+=	Add to, set equal	entry_index += 1
-=	Subtract, set equal	entry_index -= 1
*=	Multiply, set equal	entry_index *= entry_length
/=	Divide, set equal	entry_total /= entry_length
%=	Modulus, set equal	entry_index %= 8
<<=	Left shift, set equal	flags <<= 3
>>=	Right shift, set equal	flags >>= 3
&=	AND, set equal	flags &= ERROR_FLAG
|=	Inclusive OR, set equal	flags |= ERROR_FLAG
^=	Exclusive OR, set equal	flags ^= ERROR_FLAG
++	Increment	i ++
--	Decrement	i --

The Embedded Control Solutions Co.™

Microchip's PIC16/17 microcontroller family provides the unique combination of a high performance RISC processor with cost-effective One-Time-Programmable (OTP) technology for the best price-performance in the industry. Coupled with Microchip's 10 million E/W Cycle *Smart Serial*™ EEPROMs, high-volume applications get to market fast.

Microchip PIC16/17 Microcontrollers

PIC16C54A/CR54	18	12	512 x 12 (OTP/ROM)	25 x 8
PIC16C55	28	20	512 x 12 (OTP)	24 x 8
PIC16C56	18	12	1024 x 12 (OTP)	25 x 8
PIC16C57/CR57A	28	20	2048 x 12 (OTP/ROM)	72 x 8
PIC16C58A	18	12	2048 x 12 (OTP)	73 x 8
PIC16C61	18	13	1024 x 14 (OTP)	36 x 8
PIC16C62	28	22	2048 x 14 (OTP)	128 x 8
PIC16C64	40/44	33	2048 x 14 (OTP)	128 x 8
PIC16C65	40/44	33	4096 x 14 (OTP)	192 x 8
PIC16C620	18	13	512 x 14 (OTP)	80 x 8
PIC16C621	18	13	1024 x 14 (OTP)	80 x 8
PIC16C622	18	13	2048 x 14 (OTP)	128 x 8
PIC16C71(ADC)	18	13	1024 x 14 (OTP)	36 x 8
PIC16C73(ADC)	28	22	4096 x 14 (OTP)	192 x 8
PIC16C74(ADC)	40/44	33	4096 x 14 (OTP)	192 x 8
PIC16C84	18	13	1024 x 14 (EEPROM)	36 x 8
PIC17C42	40/44	33	2048 x 16 (OTP)	256 x 8
PIC17C43	40/44	33	4096 x 16 (OTP)	454 x 8
PIC17C44	40/44	33	8192 x 16 (OTP)	454 x 8

Microchip Serial EEPROMs

24AAXX	1K-16K	1.8V	I2C™
93AAXX	1K-4K	1.8V	3-WIRE
24LCXXB	1K-16K	2.5V	I2C
93LCXX	1K-4K	2.0V	3-WIRE
24XX32/65	32K-64K	High-Density *SMART SERIALS*™	
24LC21	1K	VESA® DDC™ Monitor	

Complete Systems Solutions

MPASM	Universal Assembler	X	X
MPSIM	Software Simulator	X	X
MP-C	C Compiler	X	X
*fuzzy*TECH®	Fuzzy Logic Development Kit	X	X
PICSTART™	Low Cost Development Kit	X	–
PRO MATE™	Universal Device Programmer	X	X
PICMASTER™	Universal In-Circuit Emulator	X	X
Serial EEPROM Designer's Kit – Supports all Serial EEPROMs			

Key To PIC16/17 Family Instruction Sets

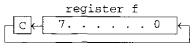
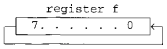
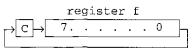
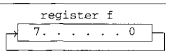
		PIC16C5X/16CXX OPCODES	PIC17CXX OPCODES
b	Bit address within an 8 bit file register		
d	Destination select; d = 0 Store result in W d = 1 Store result in file register f. Default is d = 1.		
f	Register file address (0x00 to 0xFF)		
k	Literal field, constant data or label		
W	Working register (accumulator)		
x	Don't care location		
i	Table pointer control; i = 0 Do not change. i = 1 Increment after instruction execution.		
p	Peripheral register file address (0x00 to 0x1f)		
t	Table byte select; t = 0 Perform operation on lower byte. t = 1 Perform operation on upper byte.		
PH:PL	Multiplication results registers		

PIC17CXX Instruction Set

The PIC17CXX, Microchip's high-performance 8-bit microcontroller family, uses a 16-bit wide instruction set. The PIC17CXX instruction set consists of 55 instructions, each a single 16-bit wide word. Most instructions operate on a file register, f, and the working register, W (accumulator). The result can be directed either to the file register or the W register or to both in the case of some instructions. Some devices in this family also include hardware multiply instructions. A few instructions operate solely on a file register (BSF for example).

6pff	MOVFP	f,p	Move f to p	f → p
b8kk	MOVLB	k	Move literal to BSR	k → BSR (3:0)
bakx	MOVLR	k	Move literal to RAM page select	k → BSR (7:4)
4pff	MOVPF	p,f	Move p to f	p → f
01ff	MOVWF	f	Move W to F	W → f
a8ff	TABLRD	t,i,f	Read data from table latch into file f, then update table latch with 16-bit contents of memory location addressed by table pointer	TBLATH → f if t=1, TBLATL → f if t=0; ProgMem(TBLPTR) → TBLAT; TBLPTR + 1 → TBLPTR if i=1
acff	TABLWT	t,i,f	Write data from file f to table latch and then write 16-bit table latch to program memory location addressed by table pointer	f → TBLATH if t = 1, f → TBLATL if t = 0; TBLAT → ProgMem(TBLPTR); TBLPTR + 1 → TBLPTR if i=1
a0ff	TLRD	t,f	Read data from table latch into file f (table latch unchanged)	TBLATH → f if t = 1 TBLATL → f if t = 0
a4ff	TLWT	t,f	Write data from file f into table latch	f → TBLATH if t = 1 f → TBLATL if t = 0

PIC17CXX Arithmetic and Logical Instructions

b1kk	ADDLW	k	Add literal to W	(W + k) → W
0eff	ADDWF	f,d	Add W to F	(W + f) → d
10ff	ADDWFC	f,d	Add W and Carry to f	(W + f + C) → d
b5kk	ANDLW	k	AND Literal and W	(W .AND. k) → W
0aff	ANDWF	f,d	AND W with f	(W .AND. f) → d
28ff	CLRF	f,d	Clear f and Clear d	0x00 → f,0x00 → d
12ff	COMF	f,d	Complement f	.NOT. f → d
2eff	DAW	f,d	Dec. adjust W, store in f,d	W adjusted → f and d
06ff	DECF	f,d	Decrement f	(f - 1) → f and d
14ff	INCF	f,d	Increment f	(f + 1) → f and d
b3kk	IORLW	k	Inclusive OR literal with W	(W .OR. k) → W
08ff	IORWF	f,d	Inclusive or W with f	(W .OR. f) → d
b0kk	MOVLW	k	Move literal to W	k → W
bckk	MULLW	k	Multiply literal and W	(k x W) → PH:PL
34ff	MULWF	f	Multiply W and f	(W x f) → PH:PL
2cff	NEGW	f,d	Negate W, store in f and d	(W + 1) → f, (W + 1) → d
1aff	RLCF	f,d	Rotate left through carry	
22ff	RLNCF	f,d	Rotate left (no carry)	
18ff	RRCF	f,d	Rotate right through carry	
20ff	RRNCF	f,d	Rotate right (no carry)	
2aff	SETF	f,d	Set f and Set d	0xff → f, 0xff → d
b2kk	SUBLW	k	Subtract W from literal	(k - W) → W
04ff	SUBWF	f,d	Subtract W from f	(f - W) → d
02ff	SUBWFB	f,d	Subtract from f with borrow	(f - W - c) → d
1cff	SWAPF	f,d	Swap f	f(0:3) → d(4:7), f(4:7) → d(0:3)
b4kk	XORLW	k	Exclusive OR literal with W	(W .XOR. k) → W
0cff	XORWF	f,d	Exclusive OR W with f	(W .XOR. f) → d

PIC17CXX Bit Handling Instructions

8bff	BCF	f,b	Bit clear f	0 → f(b)
8bff	BSF	f,b	Bit set f	1 → f(b)
9bff	BTFSC	f,b	Bit test, skip if clear	skip if f(b) = 0
9bff	BTFSS	f,b	Bit test, skip if set	skip if f(b) = 1
3bff	BTG	f,b	Bit toggle f	.NOT. f(b) → f(b)

PIC17CXX Program Control Instructions

ekkk	CALL	k	Subroutine call (within 8k page)	PC+1 → TOS, k → PC(12:0), k(12:8) → PCLATH(4:0), PC(15:13) → PCLATH(7:5)
31ff	CPFSEQ	f	Compare f/w, skip if f = w	f-W, skip if f = W
32ff	CPFSGT	f	Compare f/w, skip if f > w	f-W, skip if f > W
30ff	CPFSLT	f	Compare f/w, skip if f< w	f-W, skip if f < W
16ff	DECFSZ	f,d	Decrement f, skip if 0	(f-1) → d, skip if 0
26ff	DCFSNZ	f,d	Decrement f, skip if not 0	(f-1) → d, skip if not 0
ckkk	GOTO	k	Unconditional branch (within 8k)	k → PC(12:0) k(12:8) → f3(4:0), PC(15:13) → f3(7:5)
1eff	INCFSZ	f,d	Increment f, skip if zero	(f+1) → d, skip if 0
24ff	INFSNZ	f,d	Increment f, skip if not zero	(f+1) → d, skip if not 0
b7kk	LCALL	k	Long Call (within 64k)	(PC+1) → TOS; k → PCL, (PCLATH) → PCH
0005	RETFIE		Return from interrupt, enable interrupt	(f3) → PCH:k → PCL 0 → GLINTD
b6kk	RETLW	k	Return with literal in W	k → W, TOS → PC, (f3 unchanged)
0002	RETURN		Return from subroutine	TOS → PC (f3 unchanged)
33ff	TSTFSZ	f	Test f, skip if zero	skip if f = 0

PIC17CXX Special Control Instructions

0004	CLRWDT	Clear watchdog timer	0 → WDT, 0 → WDT prescaler, 1 → $\overline{PD}$, 1 → $\overline{TO}$
0000	NOP	No operation	None
0003	SLEEP	Enter Sleep Mode	Stop oscillator, power down, 0 → WDT, 0 → WDT Prescaler 1 → $\overline{PD}$, 1 → $\overline{TO}$

Hexadecimal Record Formats

The hexadecimal data records have the following format:

:BBAAAATTHHHH....HHCC

:	Start Character (prefix)
BB	Two-Digit Byte Count specifying number of data blocks in record
AAAA	Four-Digit Starting Address of data record
TT	Two-Digit Record Type 00 = Data Record 01 = End-of-File Record 02 = Segment Address Record 04 = Extended Linear Address Record (INHX32)
HHHH....HH	Two-Digit Data Blocks (BB specifies number of blocks)
CC	Two-Digit Check Sum – Two's complement of sum of all preceding bytes in data record except the colon.

Where the Output File is INHX8M
The data record is output as described above.

Where the Output File is INHX8S
The INHX8S split 8-bit file format produces two output files:

.HXL	– Stores low bytes
.HXH	– Stores high bytes

Where the Output File is INHX32
The extended linear address record is output to establish upper 16 bits of data address.

ADDLW **ADD Literal to WREG**

Syntax: [*label*] ADDLW k

Operands: 0 ≤ k ≤ 255

Operation: (WREG) + k → (WREG)

Status Affected: OV, C, DC, Z

Encoding:

1011	0001	kkkk	kkkk

Description: The contents of WREG are added to the eight bit literal 'k' and the result is placed in WREG.

Words: 1

Cycles: 1

Example:
```
ADDLW   0x15
```
Before Instruction
WREG = 0x10

After Instruction
WREG = 0x25

ADDWF **ADD WREG to f**

Syntax: [*label*] ADDWF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (WREG) + (f) → (dest)

Status Affected: OV, C, DC, Z

Encoding:

0000	111d	ffff	ffff

Description: Add WREG to register 'f'. If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Example:
```
ADDWF   REG, 0
```
Before Instruction
WREG = 0x17
REG = 0xC2

After Instruction
WREG = 0xD9
REG = 0xC2

ADDWFC **ADD WREG and Carry bit to f**

Syntax: [*label*] ADDWFC f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (WREG) + (f) + C → (dest)

Status Affected: OV, C, DC, Z

Encoding:

0001	000d	ffff	ffff

Description: Add WREG, the Carry Flag and data memory location 'f'. If 'd' is 0, the result is placed in WREG. If 'd' is 1, the result is placed in data memory location 'f'.

Words: 1

Cycles: 1

Example:
```
ADDWFC   REG  0
```
Before Instruction
Carry bit = 1
REG = 0x02
WREG = 0x4D

After Instruction
Carry bit = 0
REG = 0x02
WREG = 0x50

ANDLW **And Literal with WREG**

Syntax: [*label*] ANDLW k

Operands: 0 ≤ k ≤ 255

Operation: (WREG) .AND. (k) → (WREG)

Status Affected: Z

Encoding:

1011	0101	kkkk	kkkk

Description: The contents of WREG are AND'ed with the eight bit literal 'k'. The result is placed in WREG.

Words: 1

Cycles: 1

Example:
```
ANDLW   0x5F
```
Before Instruction
WREG = 0xA3

After Instruction
WREG = 0x03

ANDWF **AND WREG with f**

Syntax: [*label*] ANDWF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (WREG) .AND. (f) → (dest)

Status Affected: Z

Encoding:

0000	101d	ffff	ffff

Description: The contents of WREG are AND'ed with register 'f'. If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Example: `ANDWF REG, 1`

Before Instruction
WREG = 0x17
REG = 0xC2

After Instruction
WREG = 0x17
REG = 0x02

BCF **Bit Clear f**

Syntax: [*label*] BCF f,b

Operands: 0 ≤ f ≤ 255
0 ≤ b ≤ 7

Operation: 0 → (f<b>)

Status Affected: None

Encoding:

1000	1bbb	ffff	ffff

Description: Bit 'b' in register 'f' is cleared.

Words: 1

Cycles: 1

Example: `BCF FLAG_REG, 7`

Before Instruction
FLAG_REG = 0xC7

After Instruction
FLAG_REG = 0x47

BSF **Bit Set f**

Syntax: [*label*] BSF f,b

Operands: 0 ≤ f ≤ 255
0 ≤ b ≤ 7

Operation: 1 → (f<b>)

Status Affected: None

Encoding:

1000	0bbb	ffff	ffff

Description: Bit 'b' in register 'f' is set.

Words: 1

Cycles: 1

Example: `BSF FLAG_REG, 7`

Before Instruction
FLAG_REG= 0x0A

After Instruction
FLAG_REG= 0x8A

BTFSC **Bit Test, skip if Clear**

Syntax: [*label*] BTFSC f,b

Operands: 0 ≤ f ≤ 255
0 ≤ b ≤ 7

Operation: skip if (f<b>) = 0

Status Affected: None

Encoding:

1001	1bbb	ffff	ffff

Description: If bit 'b' in register 'f' is 0 then the next instruction is skipped.
If bit 'b' is 0 then the next instruction fetched during the current instruction execution is discarded, and a NOP is executed instead, making this a 2 cycle instruction.

Words: 1

Cycles: 1(2)

Example:
```
HERE    BTFSC   FLAG,1
FALSE   :
TRUE    :
```

Before Instruction
PC = address (HERE)

After Instruction
if FLAG<1> = 0;
PC = address (TRUE)
if FLAG<1> = 1;
PC = address (FALSE)

BTFSS **Bit Test, skip if Set**

Syntax: [*label*] BTFSS f,b

Operands: 0 ≤ f ≤ 127
0 ≤ b < 7

Operation: skip if (f<b>) = 1

Status Affected: None

Encoding:

1001	0bbb	ffff	ffff

Description: If bit 'b' in register 'f' is 1 then the next instruction is skipped.
If bit 'b' is 1, then the next instruction fetched during the current instruction execution, is discarded and an NOP is executed instead, making this a 2 cycle instruction.

Words: 1

Cycles: 1(2)

Example:
```
HERE    BTFSS   FLAG,1
FALSE   :
TRUE    :
```

Before Instruction
PC = address (HERE)

After Instruction
if FLAG<1> = 0;
PC = address (FALSE)
if FLAG<1> = 1;
PC = address (TRUE)

BTG **Bit Toggle f**

Syntax: [*label*] BTG f,b

Operands: 0 ≤ f ≤ 255
0 ≤ b < 7

Operation: (f<b>) → (f<b>)

Status Affected: None

Encoding:

0011	1bbb	ffff	ffff

Description: Bit 'b' in data memory location 'f' is inverted.

Words: 1

Cycles: 1

Example: `BTG PORTC, 4`

Before Instruction:
PORTC = 0111 0101 [0x75]

After Instruction:
PORTC = 0110 0101 [0x65]

CALL **Subroutine Call**

Syntax: [*label*] CALL k

Operands: 0 ≤ k ≤ 4095

Operation: PC+ 1→ TOS, k → PC<12:0>,
k<12:8> → PCLATH<4:0>;
PC<15:13> → PCLATH<7:5>

Status Affected: None

Encoding:

111k	kkkk	kkkk	kkkk

Description: Subroutine call within 8K page. First, return address (PC+1) is pushed onto the stack. The thirteen bit value is loaded into PC bits<12:0>. Then the upper-eight bits of the PC are copied into PCLATH. `Call` is a two-cycle instruction.
See LCALL for calls outside 8K memory space.

Words: 1

Cycles: 2

Example: `HERE CALL THERE`

Before Instruction
PC = Address (HERE)

After Instruction
PC = Address (THERE)
TOS = Address (HERE + 1)

CLRF **Clear f**

Syntax: [*label*] CLRF f,s

Operands: 0 ≤ f ≤ 255

Operation: 00h → f, s ∈ [0,1]
00h → dest

Status Affected: None

Encoding:

0010	100s	ffff	ffff

Description: Clears the contents of the specified register(s).
s = 0: Data memory location 'f' and WREG are cleared.
s = 1: Data memory location 'f' is cleared.

Words: 1

Cycles: 1

Example: `CLRF FLAG_REG`

Before Instruction
FLAG_REG = 0x5A

After Instruction
FLAG_REG = 0x00

CLRWDT **Clear Watchdog Timer**

Syntax: [*label*] CLRWDT

Operands: None

Operation: 00h → WDT
0 → WDT postscaler,
1 → $\overline{TO}$
1 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

0000	0000	0000	0100

Description: CLRWDT instruction resets the watchdog timer. It also resets the prescaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

Words: 1

Cycles: 1

Example:
```
CLRWDT
```
Before Instruction
WDT counter = ?
After Instruction
WDT counter = 0x00
WDT Postscaler = 0
$\overline{TO}$ = 1
$\overline{PD}$ = 1

COMF **Complement f**

Syntax: [*label*] COMF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: $(\overline{f})$ → (dest)

Status Affected: Z

Encoding:

0001	001d	ffff	ffff

Description: The contents of register 'f' are complemented. If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Example:
```
COMF        REG1,0
```
Before Instruction
REG1 = 0x13
After Instruction
REG1 = 0x13
WREG = 0xEC

CPFSEQ **Compare f with WREG, skip if f = WREG**

Syntax: [*label*] CPFSEQ f

Operands: 0 ≤ f ≤ 255

Operation: (f) – (WREG),
skip if (f) = (WREG)
(unsigned comparison)

Status Affected: None

Encoding:

0011	0001	ffff	ffff

Description: Tests the contents of data memory location 'f' to the contents of WREG.
The subtraction is unsigned.
If 'f' = WREG then the fetched instruction is discarded and an NOP is executed instead making this a two-cycle instruction.

Words: 1

Cycles: 1 (2)

Example:
```
HERE    CPFSEQ REG
NEQUAL  :
EQUAL   :
```
Before Instruction
PC Address = HERE
WREG = ?
REG = ?
After Instruction
if REG = WREG;
PC = Address (EQUAL)
if REG ≠ WREG;
PC = Address (NEQUAL)

CPFSGT **Compare f with WREG, skip if f > WREG**

Syntax: [*label*] CPFSGT f

Operands: 0 ≤ f ≤ 255

Operation: (f) – (WREG),
skip if (f) > (WREG)
(unsigned comparison)

Status Affected: None

Encoding:

0011	0010	ffff	ffff

Description: Tests the contents of data memory location 'f' to the contents of the W register.
The subtraction is unsigned.
If the contents of 'f' > the contents of WREG then the fetched instruction is discarded and an NOP is executed instead making this a two-cycle instruction.

Words: 1

Cycles: 1 (2)

Example:
```
HERE      CPFSGT, REG
NGREATER  :
GREATER   :
```
Before Instruction
PC = Address (HERE)
WREG = ?
After Instruction
if REG > WREG;
PC = Address (GREATER)
if REG ≤ WREG;
PC = Address (NGREATER)

CPFSLT — Compare f with WREG, skip if f < WREG

Syntax: [*label*] CPFSLT f

Operands: 0 ≤ f ≤ 255

Operation: (f) – (WREG), skip if (f) < (WREG) (unsigned comparison)

Status Affected: None

Encoding:

0011	0000	ffff	ffff

Description: Tests the contents of data memory location 'f' to the contents of WREG.

The subtraction is unsigned.

If the contents of 'f' < the contents of WREG, then the fetched instruction is discarded and an NOP is executed instead making this a two-cycle instruction.

Words: 1

Cycles: 1 (2)

Example:

```
HERE    CPFSLT, REG
NLESS   :
LESS    :
```

Before Instruction
- PC = Address (HERE)
- W = ?

After Instruction
- if REG < WREG;
- PC = Address (LESS)
- if REG ≥ WREG;
- PC = Address (NLESS)

DAW — Decimal Adjust WREG Register

Syntax: [*label*] DAW f,s

Operands: 0 ≤ f ≤ 255
s ∈ [0,1]

Operation:
if [WREG<3:0> >9] .OR. [DC = 1] then
WREG<3:0> + 6 → f<3:0>, s<3:0>;
else
WREG<3:0> → f<3:0>, s<3:0>;

if [WREG<7:4> >9] .OR. [C = 1] then
WREG<7:4> + 6 → f<7:4>, s<7:4>
else
WREG<7:4> → f<7:4>, s<7:4>

Status Affected: C

Encoding:

0010	111s	ffff	ffff

Description: DAW adjusts the eight bit value in WREG resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result.
- s = 0: Result is placed in Data memory location 'f' and WREG.
- s = 1: Result is placed in Data memory location 'f'.

Words: 1

Cycles: 1

Example1:

```
DAW   REG1, 0
```

Before Instruction
- WREG = 0xA5
- REG1 = ??
- C = 0
- DC = 0

After Instruction
- WREG = 0x05
- REG1 = 0x05
- C = 1
- DC = 0

Example 2:

Before Instruction
- WREG = 0xCE
- REG1 = ??
- C = 0
- DC = 0

After Instruction
- WREG = 0x24
- REG1 = 0x24
- C = 1
- DC = 0

DECF — Decrement f

Syntax: [*label*] DECF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (f) – 1 → (dest)

Status Affected: OV, C, DC, Z

Encoding:

0000	011d	ffff	ffff

Description: Decrement register 'f'. If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Example:

```
DECF    CNT,  1
```

Before Instruction
- CNT = 0x01
- Z = 0

After Instruction
- CNT = 0x00
- Z = 1

DECFSZ — Decrement f, skip if 0

Syntax: [*label*] DECFSZ f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (f) – 1 → (dest); skip if result = 0

Status Affected: None

Encoding:

0001	011d	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.

If the result is 0, the next instruction, which is already fetched, is discarded, and an NOP is executed instead making it a two cycle instruction.

Words: 1

Cycles: 1(2)

Example:

```
HERE      DECFSZ   CNT, 1
          GOTO     LOOP
CONTINUE
```

Before Instruction
- PC = Address (HERE)

After Instruction
- CNT = CNT - 1
- if CNT = 0;
- PC = Address (CONTINUE)
- if CNT ≠ 0;
- PC = Address (HERE+1)

DECFSNZ **Decrement f, skip if not 0**

Syntax: [*label*] DECFSNZ f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (f) – 1 → (dest);
skip if not 0

Status Affected: None

Encoding:

0010	011d	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.
If the result is not 0, the next instruction, which is already fetched, is discarded, and an NOP is executed instead making it a two cycle instruction.

Words: 1

Cycles: 1(2)

Example:
```
HERE    DECFSNZ  TEMP, 1
ZERO    :
NZERO   :
```
Before Instruction
TEMP_VALUE = ?
After Instruction
TEMP_VALUE = TEMP_VALUE - 1,
if TEMP_VALUE = 0;
PC = Address (ZERO)
if TEMP_VALUE ≠ 0;
PC = Address (NZERO)

GOTO **Unconditional Branch**

Syntax: [*label*] GOTO k

Operands: 0 ≤ k ≤ 8191

Operation: k → PC<12:0>;
k<12:8> → PCLATH<4:0>,
PC<15:13> → PCLATH<7:5>

Status Affected: None

Encoding:

110k	kkkk	kkkk	kkkk

Description: GOTO allows an unconditional branch anywhere within an 8K page boundary. The thirteen bit immediate value is loaded into PC bits <12:0>. Then the upper eight bits of PC are loaded into PCLATH. GOTO is always a two-cycle instruction.

Words: 1

Cycles: 2

Example:
```
GOTO THERE
```
After Instruction
PC = Address (THERE)

INCF **Increment f**

Syntax: [*label*] INCF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (f) + 1 → (dest)

Status Affected: OV, C, DC, Z

Encoding:

0001	010d	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.

Words: 1

Cycles: 1

Example:
```
INCF    CNT, 1
```
Before Instruction
CNT = 0xFF
Z = 0
C = ?
After Instruction
CNT = 0x00
Z = 1
C = 1

INCFSZ **Increment f, skip if 0**

Syntax: [*label*] INCFSZ f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (f) + 1 → (dest)
skip if result = 0

Status Affected: None

Encoding:

0001	111d	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.
If the result is 0, the next instruction, which is already fetched, is discarded, and an NOP is executed instead making it a two cycle instruction.

Words: 1

Cycles: 1(2)

Example:
```
HERE    INCFSZ   CNT, 1
NZERO   :
ZERO    :
```
Before Instruction
PC = Address (HERE)
After Instruction
CNT = CNT + 1
if CNT = 0;
PC = Address (ZERO)
if CNT ≠ 0;
PC = Address (NZERO)

INCFSNZ **Increment f, skip if not 0**

Syntax: [*label*] INCFSNZ f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (f) + 1 → (dest), skip if not 0

Status Affected: None

Encoding:

0010	010d	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.
If the result is not 0, the next instruction, which is already fetched, is discarded, and an NOP is executed instead making it a two cycle instruction.

Words: 1

Cycles: 1(2)

Example:
```
HERE    INFSNZ  REG, 1
ZERO
NZERO
```
Before Instruction
REG = REG
After Instruction
REG = REG + 1
if REG = 1;
PC = Address (ZERO)
if REG = 0;
PC = Address (NZERO)

IORLW **Inclusive OR Literal with WREG**

Syntax: [*label*] IORLW k

Operands: 0 ≤ k ≤ 255

Operation: (WREG) .OR. (k) → (WREG)

Status Affected: Z

Encoding:

1011	0011	kkkk	kkkk

Description: The contents of WREG are OR'ed with the eight bit literal 'k'. The result is placed in WREG.

Words: 1

Cycles: 1

Example:
```
IORLW   0x35
```
Before Instruction
WREG = 0x9A
After Instruction
WREG = 0xBF

IORWF **Inclusive OR WREG with f**

Syntax: [*label*] IORWF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (WREG) .OR. (f) → (dest)

Status Affected: Z

Encoding:

0000	100d	ffff	ffff

Description: Inclusive OR WREG with register 'f'. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.

Words: 1

Cycles: 1

Example:
```
IORWF   RESULT, 0
```
Before Instruction
RESULT = 0x13
WREG = 0x91
After Instruction
RESULT = 0x13
WREG = 0x93

LCALL **Long Call**

Syntax: [*label*] LCALL k

Operands: 0 ≤ k ≤ 255

Operation: PC + 1 → TOS;
k → PCL, (PCLATH) → PCH

Status Affected: None

Encoding:

1011	0111	kkkk	kkkk

Description: LCALL allows unconditional subroutine call to anywhere within the 64k program memory space.
First, the return address (PC + 1) is pushed onto the stack. A 16-bit destination address is then loaded into the program counter. The lower 8-bits of the destination address is embedded in the instruction. The upper 8-bits of PC is loaded from PC high holding latch, PCLATH.

Words: 1

Cycles: 2

Example:
```
MOVLW   HIGH(SUBROUTINE)
MOVPF   WREG, PCLATH
LCALL   LOW(SUBROUTINE)
```
Before Instruction
SUBROUTINE = 16-bit Address
PC = ?
After Instruction
PC = Address (SUBROUTINE)

MOVFP **Move f to p**

Syntax: [*label*] MOVFP f,p

Operands: 0 ≤ f ≤ 255
0 ≤ p ≤ 31

Operation: (f) → (p)

Status Affected: None

Encoding:

011p	pppp	ffff	ffff

Description: Move data from data memory location 'f' to data memory location 'p'. Location 'f' can be anywhere in the 256 word data space (00h to FFh) while 'p' can be 00h to 1Fh.
Either 'p' or 'f' can be WREG (a useful special situation).
MOVFP is particularly useful to transfer a data memory location to a peripheral register (such as the transmit buffer or an I/O port). Both 'f' and 'p' can be indirectly addressed.

Words: 1

Cycles: 1

Example:
```
MOVFP   REG1, REG2
```
Before Instruction
REG1 = 0x33,
REG2 = 0x11
After Instruction
REG1 = 0x33,
REG2 = 0x33

MOVLB **Move Literal to low nibble in BSR**

Syntax: [*label*] MOVLB k

Operands: 0 ≤ k ≤ 15

Operation: k → (BSR<3:0>)

Status Affected: None

Encoding:

1011	1000	uuuu	kkkk

Description: The constant is loaded in the Bank Select Register (BSR). Only the low 4 bits of the Bank Select Register are affected. The upper half of the BSR is unchanged. The assembler will encode the "u" fields as '0'.

Words: 1

Cycles: 1

Example:
```
MOVLB   0x5
```
Before Instruction
BSR register = 0x22
After Instruction
BSR register = 0x25

Note: For the PIC17C42, only the low four bits of the BSR register are physically implemented. The upper nibble is read as '0'.

MOVLR **Move Literal to high nibble in BSR**

Syntax: [*label*] MOVLR k

Operands: 0 ≤ k ≤ 15

Operation: k → (BSR<7:4>)

Status Affected: None

Encoding:

1011	101x	kkkk	uuuu

Description: The constant is loaded into the most significant 4 bits of the Bank Select Register (BSR). Only the high 4 bits of the Bank Select Register are affected. The lower half of the BSR is unchanged. The assembler will encode the "u" fields as 0.

Words: 1

Cycles: 1

Example:
```
MOVLR   5
```
Before Instruction
BSR register = 0x22
After Instruction
BSR register = 0x52

Note: This instruction is not available in the PIC17C42 device.

MOVLW **Move Literal to WREG**

Syntax: [*label*] MOVLW k

Operands: 0 ≤ k ≤ 255

Operation: k → (WREG)

Status Affected: None

Encoding:

1011	0000	kkkk	kkkk

Description: The eight bit literal 'k' is loaded into WREG.

Words: 1

Cycles: 1

Example:
```
MOVLW   0x5A
```
After Instruction
WREG = 0x5A

MOVPF **Move p to f**

Syntax: [*label*] MOVPF p,f

Operands: 0 ≤ f ≤ 255
0 ≤ p ≤ 31

Operation: (p) → (f)

Status Affected: Z

Encoding:

010p	pppp	ffff	ffff

Description: Move data from data memory location 'p' to data memory location 'f'. Location 'f' can be anywhere in the 256 byte data space (00h to FFh) while 'p' can be 00h to 1Fh.
Either 'p' or 'f' can be WREG (a useful special situation).
MOVPF is particularly useful for transferring a peripheral register (e.g., the timer or an I/O port) to a data memory location.

Words: 1

Cycles: 1

Example:
```
MOVPF   REG1, REG2
```
Before Instruction
REG1 = 0x11
REG2 = 0x33
After Instruction
REG1 = 0x11
REG2 = 0x11

MOVWF **Move WREG to f**

Syntax: [*label*] MOVWF f

Operands: 0 ≤ f ≤ 255

Operation: (WREG) → (f)

Status Affected: None

Encoding:

0000	0001	ffff	ffff

Description: Move data from WREG to register 'f'. Location 'f' can be anywhere in the 256 word data space.

Words: 1

Cycles: 1

Example:
```
MOVWF   REG
```
Before Instruction
WREG = 0x4F
REG = 0xFF
After Instruction
WREG = 0x4F
REG = 0x4F

MULLW **Multiply Literal with WREG**

Syntax: [*label*] MULLW k

Operands: 0 ≤ k ≤ 255

Operation: (k x WREG) → ProdH:ProdL

Status Affected: None

Encoding:

1011	1100	kkkk	kkkk

Description: An unsigned multiplication is carried out between the contents of WREG and the 8-bit literal 'k'. The 16 bit result is placed in ProdH:ProdL register pair. ProdH contains the high byte.
WREG is unchanged.
None of the status flags are affected.
Note that overflow or carry is not possible in this operation. Zero result is possible but not detected.

Words: 1

Cycles: 1

Example:
```
MULLW   0xC4
```
Before Instruction
WREG = 0xE2
PRODH = ?
PRODL = ?
After Instruction
WREG = 0xC4
PRODH = 0xAD
PRODL = 0x08

Note: This instruction is not available in the PIC17C42 device.

MULWF **Multiply WREG with f**

Syntax: [*label*] MULWF f

Operands: 0 ≤ f ≤ 255

Operation: (WREG x f) → ProdH:ProdL

Status Affected: None

Encoding:

0011	0100	ffff	ffff

Description: An unsigned multiplication is carried out between the contents of WREG and the register file location 'f'. The 16-bit result is stored in ProdH:ProdL register pair. ProdH contains the high byte.
Both WREG and 'f' are unchanged.
None of the status flags are affected.
Note that overflow or carry is not possible in this operation. Zero result is possible but not detected.

Words: 1

Cycles: 1

Example:
```
MULWF   REG
```
Before Instruction
WREG = 0xC4
REG = 0xB5
PRODH = ?
PRODL = ?
After Instruction
WREG = 0xC4
REG = 0xB5
PRODH = 0x8A
PRODL = 0x94

Note: This instruction is not available in the PIC17C42 device.

NEGW **Negate W**

Syntax: [*label*] NEGW f,s

Operands: 0 ≤ F ≤ 255
s ∈ [0,1]

Operation: $\overline{WREG} + 1 \rightarrow (f)$;
$\overline{WREG} + 1 \rightarrow s$

Status Affected: OV, C, DC, Z

Encoding:

0010	110s	ffff	ffff

Description: WREG is negated using two's complement. If 's' is 0 the result is placed in WREG and data memory location 'f'. If 's' is 1 the result is placed only in data memory location 'f'.

Words: 1

Cycles: 1

Example:
```
NEGW    REG,0
```
Before Instruction
WREG = 0011 1010 [0x3A],
REG = 1010 1011 [0xAB]
After Instruction
WREG = 1100 0111 [0xC6]
REG = 1100 0111 [0xC6]

NOP **No Operation**

Syntax: [*label*] NOP

Operands: None

Operation: No operation

Status Affected: None

Encoding:

0000	0000	0000	0000

Description: No operation.

Words: 1

Cycles: 1

Example:

RETFIE **Return from Interrupt**

Syntax: [*label*] RETFIE

Operands: None

Operation: TOS → (PC);
0 → GLINTD;
PCLATH is unchanged.

Status Affected: GLINTD

Encoding:

0000	0000	0000	0101

Description: Return from Interrupt. Stack is POP'ed and Top of Stack (TOS) is loaded in the PC. Interrupts are enabled by clearing the GLINTD bit. GLINTD is the global interrupt disable bit (CPUSTA<4>).

Words: 1

Cycles: 2

Example:
```
RETFIE
```
After Interrupt
PC = TOS
GLINTD = 0

RETLW **Return Literal to WREG**

Syntax: [*label*] RETLW k

Operands: 0 ≤ k ≤ 255

Operation: k → (WREG); TOS → (PC);
PCLATH is unchanged

Status Affected: None

Encoding:

1011	0110	kkkk	kkkk

Description: WREG is loaded with the eight bit literal 'k'. The program counter is loaded from the top of the stack (the return address). The high address latch (PCLATH) remains unchanged.

Words: 1

Cycles: 2

Example:
```
   CALL TABLE ; WREG contains table
              ;  offset value
              ; WREG now has
               ;  table value
   :
TABLE
   ADDWF PC   ; WREG = offset
   RETLW k0   ; Begin table
   RETLW k1   ;
   :
   :
   RETLW kn   ; End of table
```
Before Instruction
WREG = 0x07
After Instruction
WREG = value of k7

RETURN — **Return from Subroutine**

Syntax: [*label*] RETURN

Operands: None

Operation: TOS → PC;
PCLATH is unchanged

Status Affected: None

Encoding:

0000	0000	0000	0010

Description: Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter.

Words: 1

Cycles: 2

Example:
```
RETURN
```
After Interrupt
PC = TOS

RLCF — **Rotate Left f through Carry**

Syntax: [*label*] RLCF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

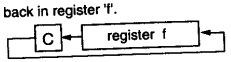
Operation: f<n> → d<n+1>;
f<7> → C;
C → d<0>

Status Affected: C

Encoding:

0001	101d	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the left through the Carry Flag. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is stored back in register 'f'.



Words: 1

Cycles: 1

Example:
```
RLCF        REG,0
```
Before Instruction
REG = 1110 0110
C = 0

After Instruction
REG = 1110 0110
WREG = 1100 1100
C = 1

RLNCF — **Rotate Left f (no carry)**

Syntax: [*label*] RLNCF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

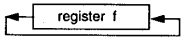
Operation: f<n> → d<n+1>;
f<7> → d<0>

Status Affected: None

Encoding:

0010	001d	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the left. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is stored back in register 'f'.



Words: 1

Cycles: 1

Example:
```
RLNCF       REG, 1
```
Before Instruction
C = 0
REG = 1110 1011

After Instruction
C =
REG = 1101 0111

RRCF — **Rotate Right f through Carry**

Syntax: [*label*] RRCF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

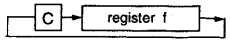
Operation: f<n> → d<n-1>;
f<0> → C;
C → d<7>

Status Affected: C

Encoding:

0001	100d	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed back in register 'f'.



Words: 1

Cycles: 1

Example:
```
RRCF        REG1,0
```
Before Instruction
REG1 = 1110 0110
C = 0

After Instruction
REG1 = 1110 0110
WREG = 0111 0011
C = 0

SUBWF **Subtract WREG from f**

Syntax: [*label*] SUBWF f,d

Operands: $0 \le f \le 255$
$d \in [0,1]$

Operation: (f) – (W) → (dest)

Status Affected: OV, C, DC, Z

Encoding:

0000	010d	ffff	ffff

Description: Subtract WREG from register 'f' (2's complement method). If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Example 1:
```
SUBWF   REG1, 1
```
Before Instruction
```
REG1  =  3
WREG  =  2
C     =  ?
```
After Instruction
```
REG1  =  1
WREG  =  2
C     =  1    ; result is positive
Z     =  1
```
Example 2:

Before Instruction
```
REG1  =  2
WREG  =  2
C     =  ?
```
After Instruction
```
REG1  =  0
WREG  =  2
C     =  1    ; result is zero
Z     =  0
```
Example 3:

Before Instruction
```
REG1  =  1
WREG  =  2
C     =  ?
```
After Instruction
```
REG1  =  FF
WREG  =  2
C     =  0    ; result is negative
Z     =  1
```

SUBWFB **Subtract WREG from f with Borrow**

Syntax: [*label*] SUBWFB f,d

Operands: $0 \le f \le 255$
$d \in [0,1]$

Operation: (f) – (W) – $\overline{C}$ → (dest)

Status Affected: OV, C, DC, Z

Encoding:

0000	001d	ffff	ffff

Description: Subtract WREG and the carry flag (borrow)from register 'f' (2's complement method). If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Example 1:
```
SUBWFB  REG1, 1
```
Before Instruction
```
REG1  =  0x19    (0001 1001)
WREG  =  0x0D    (0000 1101)
C     =  1
```
After Instruction
```
REG1  =  0x0B    (0000 1011)
WREG  =  0x0D    (0000 1101)
C     =  1       ; result is positive
Z     =  1
```
Example2:
```
SUBWFB  REG1,0
```
Before Instruction
```
REG1  =  0x1B    (0001 1011)
WREG  =  0x1A    (0001 1010)
C     =  1
```
After Instruction
```
REG1  =  0x1B    (0001 1011)
WREG  =  0x00
C     =  1       ; result is zero
Z     =  0
```
Example3:
```
SUBWFB  REG1,1
```
Before Instruction
```
REG1  =  0x03    (0000 0011)
WREG  =  0x0E    (0000 1101)
C     =  1
```
After Instruction
```
REG1  =  0xF4    (1111 0100) [2's comp]
WREG  =  0x0E    (0000 1101)
C     =  0       ; result is negative
Z     =  1
```

SWAPF **Swap f**

Syntax: [*label*] SWAPF f,d

Operands: $0 \le f \le 255$
$d \in [0,1]$

Operation: f<3:0> → dest<7:4>;
f<7:4> → dest<3:0>

Status Affected: None

Encoding:

0001	110d	ffff	ffff

Description: The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0 the result is placed in WREG. If 'd' is 1 the result is placed in register 'f'.

Words: 1

Cycles: 1

Example:
```
SWAPF   REG,   0
```
Before Instruction
```
REG   =  0x53
```
After Instruction
```
REG   =  0x35
```

TABLRD **Table Read**

Syntax: [*label*] TABLRD t,i,f

Operands: $0 \le f \le 255$
$i \in [0,1]$
$t \in [0,1]$

Operation:
If t = 1,
TBLATH → f;
if t = 0,
TBLATL → f;
Prog Mem (TBLPTR) → TBLAT;
if i = 1,
TBLPTR + 1 → TBLPTR

Status Affected: None

Encoding:

1010	10ti	ffff	ffff

Description:
1. A byte of the table latch (TBLAT) is moved to register file 'f'.
if t = 0: the high byte is moved;
if t = 1: the low byte is moved
2. Then the contents of the program memory location pointed to by the 16-bit Table Pointer (TBLPTR) is loaded into the 16-bit Table Latch (TBLAT).
3. if i = 1: TBLPTR is incremented;
if i = 0: TBLPTR is not incremented

Words: 1

Cycles: 2 (3 cycle if f = PC)

Example1:
```
TABLRD  1, 1, REG ;
```
Before Instruction
```
REG           =  0x53
TBLATH        =  0xAA
TBLATL        =  0x55
TBLPTR        =  0xA356
MEMORY(TBLPTR)=  0x1234
```
After Instruction (table write completion)
```
REG           =  0xAA
TBLATH        =  0x12
TBLATL        =  0x34
TBLPTR        =  0xA357
MEMORY(TBLPTR)=  0x5678
```
Example2:
```
TABLRD  0, 0, REG ;
```
Before Instruction
```
REG           =  0x53
TBLATH        =  0xAA
TBLATL        =  0x55
TBLPTR        =  0xA356
MEMORY(TBLPTR)=  0x1234
```
After Instruction (table write completion)
```
REG           =  0x55
TBLATH        =  0x12
TBLATL        =  0x34
TBLPTR        =  0xA356
MEMORY(TBLPTR)=  0x1234
```

TABLWT **Table Write**

Syntax: [*label*] TABLWT t,i,f

Operands: $0 \le f \le 255$
$i \in [0,1]$
$t \in [0,1]$

Operation: If t = 0,
f → TBLATL;
if t = 1,
f → TBLATH;
TBLAT → Prog Mem (TBLPTR);
if i = 1,
TBLPTR + 1 → TBLPTR

Status Affected: None

Encoding:

1010	11ti	ffff	ffff

Description:
1. Load value in 'f' into 16-bit table latch (TBLAT)
   if t = 0: load into low byte;
   if t = 1: load into high byte
2. The contents of TBLAT is written to the program memory location pointed to by TBLPTR
   If TBLPTR points to external program memory location, then the instruction takes two cycles.
   If TBLPTR points to an internal EPROM location, then the instruction is terminated when an interrupt is received.

Note: The $\overline{MCLR}/V_{PP}$ pin must be at the programming voltage for successful programming.
If $\overline{MCLR}/V_{PP} = V_{DD}$
the programming sequence will be executed, but will not be successful (although the memory location may be disturbed)

3. The TBLPTR can be automatically incremented
   if i = 0; TBLPTR is not incremented
   if i = 0; TBLPTR is incremented

Words: 1

Cycles: 2 (many if write is to on-chip EPROM program memory)

TABLWT (Cont.) **Table Write**

Example1: `TABLWT 0, 1, REG`

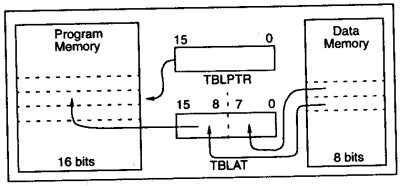
Before Instruction
REG = 0x53
TBLATH = 0xAA
TBLATL = 0x55
TBLPTR = 0xA356
MEMORY(TBLPTR) = 0xFFFF

After Instruction (table write completion)
REG = 0x53
TBLATH = 0x53
TBLATL = 0x55
TBLPTR = 0xA357
MEMORY(TBLPTR) = 0x5355

Example 2: `TABLWT 1, 0, REG`

Before Instruction
REG = 0x53
TBLATH = 0xAA
TBLATL = 0x55
TBLPTR = 0xA356
MEMORY(TBLPTR) = 0xFFFF

After Instruction (table write completion)
REG = 0x53
TBLATH = 0xAA
TBLATL = 0x53
TBLPTR = 0xA356
MEMORY(TBLPTR) = 0xAA53



TLRD **Table Latch Read**

Syntax: [*label*] TLRD t,f

Operands: $0 \le f \le 255$
$t \in [0,1]$

Operation: If t = 0,
TBLATL → f;
if t = 1,
TBLATH → f

Status Affected: None

Encoding:

1010	00tx	ffff	ffff

Description: Read data from 16-bit table latch (TBLAT) into file register 'f'. Table Latch is unaffected.
If t = 1; high byte is read
if t = 0; low byte is read
This instruction is used in conjunction with TABLRD to transfer data from program memory to data memory.

Words: 1

Cycles: 1

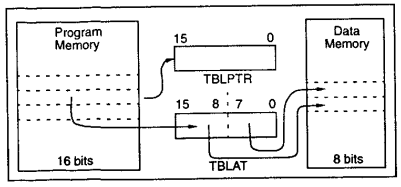
Example: `TLRD t, RAM`

Before Instruction
t = 0
RAM = ?
TBLAT = 0x00AF (TBLATH = 0x00) (TBLATL = 0xAF)

After Instruction
RAM = 0xAF
TBLAT = 0x00AF (TBLATH = 0x00) (TBLATL = 0xAF)

Before Instruction
t = 1
RAM = ?
TBLAT = 0x00AF (TBLATH = 0x00) (TBLATL = 0xAF)

After Instruction
RAM = 0x00
TBLAT = 0x00AF (TBLATH = 0x00) (TBLATL = 0xAF)



TLWT **Table Latch Write**

Syntax: [*label*] TLWT t,f

Operands: $0 \le f \le 255$
$t \in [0,1]$

Operation: If t = 0,
f → TBLATL;
if t = 1,
f → TBLATH

Status Affected: None

Encoding:

1010	01tx	ffff	ffff

Description: Data from file register 'f' is written into the 16-bit table latch (TBLAT).
If t = 1; high byte is written
if t = 0; low byte is written
This instruction is used in conjunction with TABLWT to transfer data from data memory to program memory.

Words: 1

Cycles: 1

Example: `TLWT t, RAM`

Before Instruction
t = 0
RAM = 0xB7
TBLAT = 0x0000 (TBLATH = 0x00) (TBLATL = 0x00)

After Instruction
RAM = 0xB7
TBLAT = 0x00B7 (TBLATH = 0x00) (TBLATL = 0xB7)

Before Instruction
t = 1
RAM = 0xB7
TBLAT = 0x0000 (TBLATH = 0x00) (TBLATL = 0x00)

After Instruction
RAM = 0xB7
TBLAT = 0xB700 (TBLATH = 0xB7) (TBLATL = 0x00)

TSTFSZ — **Test f, skip if 0**

Syntax: [*label*] TSTFSZ f

Operands: 0 ≤ f ≤ 255

Operation: skip if f = 0

Status Affected: None

Encoding:

0011	0011	ffff	ffff

Description: If 'f' = 0, the next instruction, fetched during the current instruction execution, is discarded and an NOP is executed making this a two-cycle instruction.

Words: 1

Cycles: 1 (2)

Example:

```
HERE    TSTFSZ  CNT
NZERO   :
ZERO    :
```

Before Instruction
PC = Address(HERE)

After Instruction
if CNT = 0x00,
PC = Address (ZERO)
if CNT ≠ 0x00,
PC = Address (NZERO)

XORLW — **Exclusive OR Literal with WREG**

Syntax: [*label*] XORLW k

Operands: 0 ≤ k ≤ 255

Operation: (WREG) .XOR. k → (WREG)

Status Affected: Z

Encoding:

1011	0100	kkkk	kkkk

Description: The contents of WREG are XOR'ed with the eight bit literal 'k'. The result is placed in WREG.

Words: 1

Cycles: 1

Example:

```
XORLW  0xAF
```

Before Instruction
WREG = 0xB5

After Instruction
WREG = 0x1A

XORWF — **Exclusive OR WREG with f**

Syntax: [*label*] XORWF f,d

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]

Operation: (WREG) .XOR. (f) → (dest)

Status Affected: Z

Encoding:

0000	110d	ffff	ffff

Description: Exclusive OR the contents of WREG with register 'f'. If 'd' is 0 the result is stored in WREG. If 'd' is 1 the result is stored back in the register 'f'.

Words: 1

Cycles: 1

Example:

```
XORWF   REG, 1
```

Before Instruction
REG = 0xAF
WREG = 0xB5

After Instruction
REG = 0x1A
WREG = 0xB5

10.5.3 Aufbau eines Assembler-Files

Format des Quelltextes

Damit das Assemblerprogramm den Quelltext korrekt übersetzen kann, muß dieser bestimmten Vereinbarungen genügen. Die Schreibweise der Befehle und Label gehört genauso dazu wie die Einhaltung der vom Assembler verlangten Syntax bei arithmetischen und logischen Operationen. Alle weiteren Erläuterungen beziehen sich ausschließlich auf den PIC-ASM17-Assembler Version 1.08 der Firma Microchip Technology Inc., der dem Buch in Form einer Diskette beiliegt.

Der Quelltext darf folgende Zeichen enthalten (mit Ausnahme des Kommentarfeldes, hier sind alle Zeichen zulässig):

- Buchstaben a...z (kein ß!),
- Buchstaben A..Z,
- Ziffern 0...9,
- Sonderzeichen $, &, ?, ., _.

Folgende Zeichen sind für Spezialzwecke reserviert:

‘ “ + - ; \ * ( ) = und das Leerzeichen.

Grundsätzlich besteht jeder Quelltext aus einer Folge von Zeilen, die alle nach einem bestimmten Muster aufgebaut sind:

[Label] Befehl [Operand] [Kommentar]

Eingeklammerte Felder [] können bei bestimmten Zeilen fehlen. Die Bedeutung der einzelnen Felder wird im folgenden beschrieben.

Label

Das Labelfeld ist das erste Feld einer Programmzeile. Es wird als Referenz benötigt, zum Beispiel bei Sprüngen oder beim Zugriff auf Tabellen. Die maximal zulässige Länge eines Labels beim PIC-ASM17 beträgt 132 Zeichen. Das erste Zeichen eines Labels sollte keine Ziffer sein. Gültige Label sind daher:

START
start
_start
A12345
x12_35

Ungültige Label sind:

12345
-abcd
ABC+-

Befehl

Das Befehlsfeld enthält wahlweise einen Maschinenbefehl, eine Assembleranweisung oder einen Makrobefehl. Es ist von den anschließenden Feldern durch mindestens je ein Leerzeichen oder Tab getrennt.

Operand

Der Operand eines Befehls ist das dritte Feld einer Quelltextzeile. Operanden können Symbole oder Label sein, Konstanten, Variable oder mathematische und logische Ausdrücke. Nicht jeder Befehl benötigt einen Operanden, das Feld kann also auch leer bleiben.

Symbole

Symbole werden ähnlich eingesetzt wie Label, sie dienen als Referenz für Variable, Konstanten oder Adressen. Das erste Zeichen eines Symbols muß ein Buchstabe sein, ansonsten gelten die gleichen Konventionen wie beim Labelfeld.

Mathematische Ausdrücke

Mathematische Ausdrücke können Bestandteil des Operandenfelds sein. Es können beliebige Kombinationen aus Konstanten, Label und Symbolen miteinander kombiniert werden. Der PIC-ASM17-Assembler erlaubt Vorwärtsreferenzen, da die Operanden erst im zweiten Durchgang der Assemblierung berechnet werden. Als mathematische Ausdrücke sind zulässig:

+	Addition	1 + 2, abc + def, 2 + (abc + def)
-	Subtraktion	1 - 2, abc - def, 2 - (abc - def)
*	Multiplikation	1 * 2, abc * def, 2 * abc * def
/	Division	1 / 2, abc / def, 2 / abc / def
<<	Links schieben	1 << 2, abc << def, 2 << (abc << def)
>>	Rechts schieben	1 >> 2, abc >> def, 2 >> (abc >> def)

Klammern können beliebig tief geschachtelt werden. Ausdrücke werden von links nach rechts berechnet.

Kommentar

Den Abschluß einer Quelltextzeile bildet das Kommentarfeld. Es beginnt immer mit einem Semikolon und endet mit dem Zeilenende (CR). Innerhalb des Kommentarfeldes gibt es keine Beschränkungen hinsichtlich der verwendbaren Zeichen, auch reservierte Symbole dürfen hier benutzt werden.

Konstanten

Konstanten, die im Operandenfeld benutzt werden, müssen bestimmten Konventionen genügen. Die Grundeinstellung des PIC-ASM17-Assemblers ist hexadezimal, sie kann aber je nach Wunsch auf andere Zahlensysteme umgestellt werden. Hexadezimale Ausdrücke werden in der Form 0xnnnn notiert, dezimale Ausdrücke als 0nnnn und Binärzahlen als nnnnnnnnb. Vom gewählten Zahlensystem abhängig ist natürlich auch der zulässige Wertebereich der Ausdrücke.

System	Zahlenbereich	Syntax
Dezimal	0..9	01234
Hexadezimal	0..F	0x1234
Binär	0..1	00110101b
Zeichen	Alphanumerisch	'C'

Als Vorzeichen für Konstanten sind das Plus- und Minuszeichen zugelassen. Fehlt das Vorzeichen, so wird ein positiver Wert angenommen.

Assembleranweisungen:

Assembleranweisungen sind Bestandteile des Quelltextes, die zur Steuerung Assemblierungsvorgangs dienen. Teilweise nehmen sie Einfluß auf den entstehen Maschinencode, beispielsweise durch Zuweisung von Adressen, teilweise dienen sie Strukturierung der Dokumentation. Der PIC-ASM17-Assembler kennt folge Anweisungen:

Anweisung	Wirkung
CBLOCK	Definition einer Liste von Konstanten
CONSTANT	Definition einer Konstanten
DATA	Zuweisung von Daten im Speicher
#DEFINE	Zuweisung eines Wertes an einen Label
DW	Definition eines 16-bit-Wortes
#ELSE,#ELIF	bedingte Assemblierung
END	Ende des Programmtextes
ENDC	Abschluß einer Liste von Konstanten
#ENDIF	Abschluß bedingte Assemblierung
EQU	Zuweisung eines Ausdrucks zu einem Symbol
ERROR	Definition einer eigenen Fehlermeldung
ERRORLEVEL	Zuweisung eines Fehlergrades
FILL	Definition eines Füllwertes für freien Speicher
HIGH	High-Byte einer Adresse
#IF	bedingte Assemblierung
#INCLUDE	Einbindung externer Quelltexte
LIST	Definition des Druckformats
MESSG	Definition eines Hinweistextes
NOLIST	Listing der folgenden Zeilen ausschalten
ORG	Steuerung der Programmadressen
PAGE	Seitenvorschub
PROCESSOR	Auswahl des Zielprozessors
RESV	reserviert Platz im Speicher
SET	Zuweisung eines Ausdrucks zu einem Symbol
STITL	Untertitel
TITLE	Titelzeile
#UNDEFINE	Rücknahme einer Zuweisung
VARIABLE	Definition einer Variablen

10.5.3.1 Beispiel für ein Assemblerfile

```
list p=17c43, f=inhx32                  ;Device name und Art des HEX-Codes,
                                        ;der erzeugt werden soll
include c:\pic\mplab\p17c43.inc         ;Das File enthält die Definition der vorgegeben Register
                                        ;im RAM, so dass man diese namentlich gleich
                                        ;ins Programm einbinden kann (MPLAB-Software)
                                        ;Für die PICMASTER-Software heisst der Befehl:
                                        ;include "p17cxx.inc"

; Definition von verschiedenen 8-bit-RAM-Adressen als Variable, die man
; dann hinterher unter dem definierten Namen ansprechen kann

wreg            equ     0x0a
bsr             equ     0x0f
add1hi          equ
.
.
.

; Das eigentliche Programm beginnt nun
; Es beginnt bei der Adresse 0000 im ROM

                goto start

org             0100                    ; sub1 will man an der ROM-Adresse 0100 stehen haben
sub1            Befehl1_sub
                Befehl2_sub2
                return                  ; Rücksprung

org             0018                    ; start-Routine soll an der Adresse 0018 im ROM stehen

start           Befehl 1                ; Kommentare ganz wichtig
                Befehl 2
                call sub1               ; Unterroutine sub 1 wird aufgerufen
                .
                .
                .
                Befehl n
                end                     ; programmende
```

10.5.3.2 Beispiel für die Interruptbehandlung in einem Assemblerfile

Der PIC17C42 stellt für jeden der vier Interruptvektoren jeweils 8 Byte an Programmspeicher zur Verfügung. Wenn die Interruptroutine länger wird, muß sie an eine andere Stelle im Programmspeicher verschoben werden und mit einem Sprungbefehl angesprungen werden:

```
            org     0000
reset       goto    start           ; Start Hauptprogramm

            org     0008
int_vec     goto    ext_int         ; Sprung in Interrupt-
                                    ; routine
            org     0010
rtcc_vec    retfie                  ; Dieser Interrupt wird
                                    ; nicht benutzt, aus
                                    ; Sicherheitsgründen wird
                                    ; eine Minimalroutine
                                    ; eingebaut, die falsch
                                    ; ausgelöste Interrupts
                                    ; abfängt.

            org     0018
rt_vect     incf    zaehler,f       ; Dieses Interrupt-
                                    ; programm zählt Impulse
                                    ; am RT-Pin. Es verändert
                                    ; kein Register.
            retfie

            org     0020
peri_vec    btfsc   pir,irb         ; Test auf Port B
                                    ; Interrupt
            goto    portb_int       ; gefunden
            btfsc   pir,tbmt        ; Test Sendepuffer
            goto    senden          ; gefunden
            .
            .
```

10.5.4 Einführung in die Software MPLAB

Introduction

This chapter will give an overview of MPLAB IDE.

Highlights

In this chapter, we discuss:

- What is MPLAB IDE
- How MPLAB IDE Helps You
- MPLAB IDE – An Integrated Development Environment (IDE)
- MPLAB IDE Development Tools

What is MPLAB IDE

MPLAB IDE is a Windows®-based Integrated Development Environment (IDE) for the Microchip Technology Incorporated PICmicro® microcontroller (MCU) families. MPLAB IDE allows you to write, debug, and optimize PICmicro MUC applications for firmware product designs. MPLAB IDE includes a text editor, simulator, and project manager. MPLAB IDE also supports the MPLAB-ICE and PICMASTER® emulators, PICSTART® Plus and PRO MATE® II programmers, and other Microchip or third party development system tools.

How MPLAB IDE Helps You

The organization of MPLAB IDE tools by function helps make pull-down menus and customizable quick keys easy to find and use. MPLAB IDE tools allow you to:

- Assemble, compile, and link source code
- Debug the executable logic by watching program flow with the simulator, or in real time with the MPLAB-ICE emulator
- Make timing measurements
- View variables in watch windows
- Program firmware with PICSTART Plus or PRO MATE II
- Find quick answers to questions from the MPLAB IDE on-line Help

and much more.

MPLAB IDE – An Integrated Development Environment (IDE)

MPLAB IDE is an easy-to-learn and use Integrated Development Environment (IDE). The IDE provides firmware development engineers the flexibility to develop and debug firmware for Microchip's PICmicro MCU families. The MPLAB IDE runs under Microsoft Windows 3.1x, Windows 95/98, Windows NT, or Windows 2000.

> **Note:** Not all hardware components that function under the MPLAB IDE, such as emulators and device programmer, function under all operating systems. Refer to the user's guide of the specific hardware device for details.

MPLAB IDE provides functions that allow you to:

- Create and Edit Source Files
- Group Files into Projects
- Debug Source Code
- Debug Executable Logic Using the Simulator or Emulator(s)

The MPLAB IDE allows you to create and edit source code by providing you with a full-featured text editor.

Further, you can easily debug source code with the aid of a Build Results window that displays the errors found by the compiler, assembler, and linker when generating executable files.

A Project Manager allows you to group source files, precompiled object files, libraries, and linker script files into a project format.

The MPLAB IDE also provides feature-rich simulator and emulator environments to debug the logic of executables. Some of the features are:

- A variety of windows allowing you to view the contents of all data and program memory locations
- Source Code, Program Memory, and Absolute Listing windows allowing you to view the source code and its assembly-level equivalent separately and together (Absolute Listing)
- The ability to step through execution, or apply Break, Trace, Standard, or Complex Trigger points

MPLAB IDE Development Tools

The MPLAB IDE integrates several tools to provide a complete development environment.

- **MPLAB Project Manager**

  Use the Project Manager to create a project and work with the specific files related to the project. When using a project, you can rebuild source code and download it to the simulator or emulator with a single mouse click.

- **MPLAB Editor**

  Use the MPLAB Editor to create and edit text files such as source files, code, and linker script files.

- **MPLAB-ICD In-Circuit Debugger**

  The MPLAB-ICD In-Circuit Debugger is a powerful, low-cost development and evaluation kit for the FLASH PIC16F87X MCU family.

- **MPLAB-SIM Simulator**

  The software simulator models the instruction execution and I/O of the PICmicro MCUs.

- **MPLAB-ICE Emulator**

  The MPLAB-ICE emulator uses hardware to emulate PICmicro MCUs in real time, either with or without a target system.

- **MPASM Assembler/MPLINK Linker/MPLIB Librarian**

  The MPASM assembler allows source code to be assembled without leaving MPLAB IDE. MPLINK creates the final application by linking relocatable modules from MPASM, MPLAB-C17, and MPLAB-C18. MPLIB manages custom libraries for maximum code reuse.

- **MPLAB-CXX C Compilers**

  The MPLAB-C17 and MPLAB-C18 C Compilers provide ANSI-based high level source code solutions. Complex projects can use a combination of C and assembly source files to obtain the maximum benefits of speed and maintainability.

- **PRO MATE® II and PICSTART® Plus Programmers**

  Develop code with the simulator or an emulator, assemble or compile it, then use one of these tools to program devices. This can all be accomplished with MPLAB IDE. Although PRO MATE II does not require MPLAB IDE to operate, programming is easier using MPLAB IDE.

- **PICMASTER and PICMASTER-CE Emulators**

  These emulators use hardware to emulate PICmicro MCUs in real time, either with or without a target system. MPLAB-ICE is the newest emulator from Microchip.

• Third Party Tools

Many other companies have development tools for Microchip products that work with MPLAB IDE. Consult the *Microchip Third Party Guide* (DS00104).

10.5.5 Beschreibung der Toolbars von MPLAB

Edit Toolbar

The Edit toolbar contains buttons that are commonly used when editing source code.

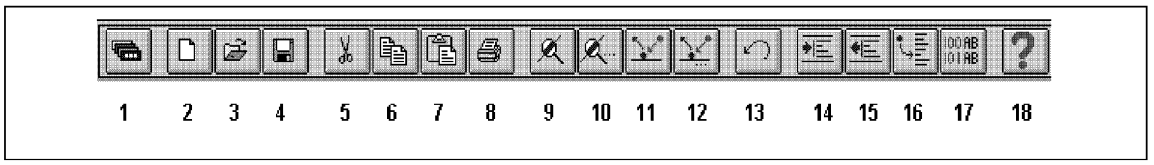


Figure 7.82: Edit Toolbar

The default buttons are:

1. Change toolbar
2. File New
3. File Open
4. File Save
5. Cut
6. Copy
7. Paste
8. Print
9. Find
10. Repeat Last Find
11. Replace
12. Repeat Last Replace
13. Undo Edit
14. Indent
15. Unindent
16. Goto Line
17. Toggle Line Numbers
18. Help Contents

Debug Toolbar

The Debug toolbar contains buttons that are commonly used when running and debugging code.

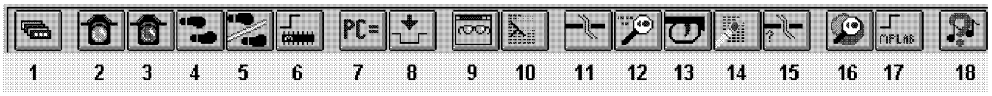


Figure 7.83: DebugToolbar

The default buttons are:

1. Change toolbar
2. Run Program
3. Halt Program
4. Step Through Program
5. Step Over
6. Reset System
7. Change PC
8. Execute Opcode
9. New Watch Window
10. Modify Window
11. Break Point
12. Trace Point
13. Trigger
14. Clear All Breaks
15. Conditional Break
16. Halt Trace
17. System Reset
18. Help Release Notes

Project Toolbar

The Project toolbar contains icons that are commonly used when running and debugging code.

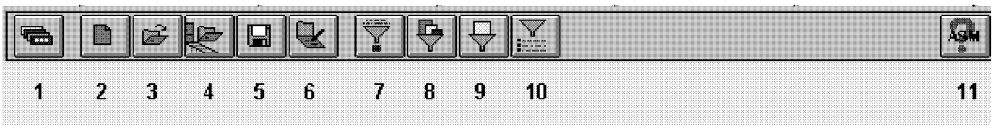


Figure 7.84: Project Toolbar

The default buttons are:

1. Change toolbar
2. New Project
3. Open Project
4. Close Project
5. Save Project
6. Edit Project
7. Make Project
8. Build All
9. Build Node
10. Install Tools
11. Help

User Defined Toolbar

The User Defined toolbar is intended to be customized to contain buttons that meet the individual user's needs.

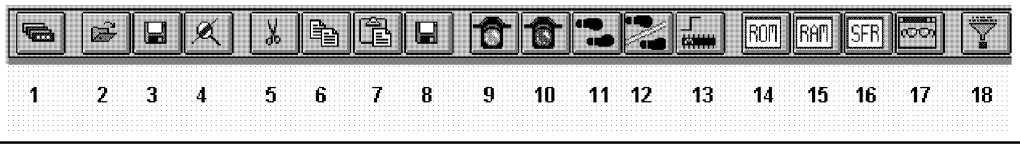


Figure 7.85: User Defined Toolbar

It initially contains the following buttons:

1. Change Toolbar
2. Open Project
3. Save Project
4. Find
5. Cut
6. Copy
7. Paste
8. Save File
9. Run
10. Halt
11. Step
12. Step Over
13. Reset
14. Program Memory Window
15. File Registers Window
16. Special Function Registers Window
17. New Watch Window
18. Make Project

MPLAB IDE Status Bar

The Status Bar indicates such current information as cursor position, development mode and device, and active toolbar.

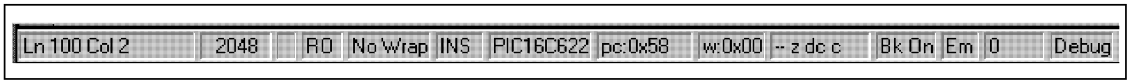


Figure 7.86: Status Bar

Title	Typical Entry	Description	Result from Double Clicking
Line No., Column–Windows Open Or, displays MPLAB IDE Version Number when no windows are open	Ln 1 Col 1 2.00.00	Displays current line number and column in file MPLAB IDE Version Number	Opens Goto Line Dialog No Action
Lines in File	72	Displays number of lines in current text file	No Action
File Modified	#	Displays # Symbol if file has been changed since opening	No Action
Write/Read Only	WR	Displays Write/Read Only Status. WR = Editable File RO = Read Only File	Toggles between write and read only for files that you have access to
Text Wrap	No Wrap	Displays current wrap mode and wrap column if text wrap is on Example 1: NoWrap Example 2: WR 72 Useful for text files. Use *Options > Current Editor Modes* to change wrap column.	Toggles between wrap and no wrap No Wrap Wrap at Column 72
Insert/Strikeover	INS	Toggles typing mode between insert and strikeover INS = Insert Characters OVR = Type over characters	Toggles between INS and OVR
Current Processor	PIC16C61	Displays the currently selected processor	No Action
Current Program Counter	pc:0x5f	Displays the current program counter	Opens Change Program Counter dialog
Current w Register Value	W:0x00	Displays current w register value	No Action
Status Bits	ov Z dc c	Upper Case = Set (1) Lower Case = Reset (0)	No Action
Global Break Enable	Bk On	Displays current status of Global Break Enable	Toggles Global Break Enable On and Off

Title	Typical Entry	Description	Result from Double Clicking
Current Development Mode	Sim	Displays Current Development Mode. Examples: EO = Editor Only Sim = Simulator – MPLAB-SIM Si = Simulator – SIMICE ICE = Emulator – MPLAB-ICE Em = Emulator – PICMASTER emulator	Displays Development Mode Dialog
Processor Frequency	4 MHz	Displays current processor frequency	Opens processor clock dialog
Current Tool Bar	Edit	Displays current tool bar	No Action

10.5.6 Tutorial zur Software MPLAB

Introduction

This tutorial is intended to be a quick introduction to the MPLAB IDE user interface. It should take about 1 to 2 hours to complete the Tutorial.

This is not intended to discuss all of the details of MPLAB IDE, only to provide a beginning understanding so you can use MPLAB IDE right away.

Highlights

In this tutorial, you will learn about:

- Setting up the Development Mode
- Creating a Simple New Project
- Creating a Simple New Source File
- Entering Source Code
- Assembling the Source File
- Running Your Program
- Opening Other Windows for Debugging
- Creating a Watch Window
- Saving the Watch Window
- Setting a Break Point

In addition, there is an overview of other features to be discussed in later chapters.

With the operation of MPLAB IDE provided by this tutorial, you should be able to:

- Become familiar with the MPLAB IDE Desktop
- Create a new assembly source code file and enter it into a new project for the PIC16F84
- Identify and correct simple errors
- Run the built in simulator
- Set break points
- Create Watch windows
- Become familiar with the various debugging windows

Setting up the Development Mode

The previous chapter discussed how to install MPLAB IDE. Now you will begin setting up the application.

The MPLAB IDE desktop (Figure 3.1) contains the following major elements:

1. A menu across the top line
2. A toolbar below the menu
3. A workspace in which various files, windows, and dialogs can be displayed
4. A status bar at the bottom

Notice that the status bar includes information about how the system is currently configured. We'll cover some of these features in more detail later. For now, let's see how to set the development mode.

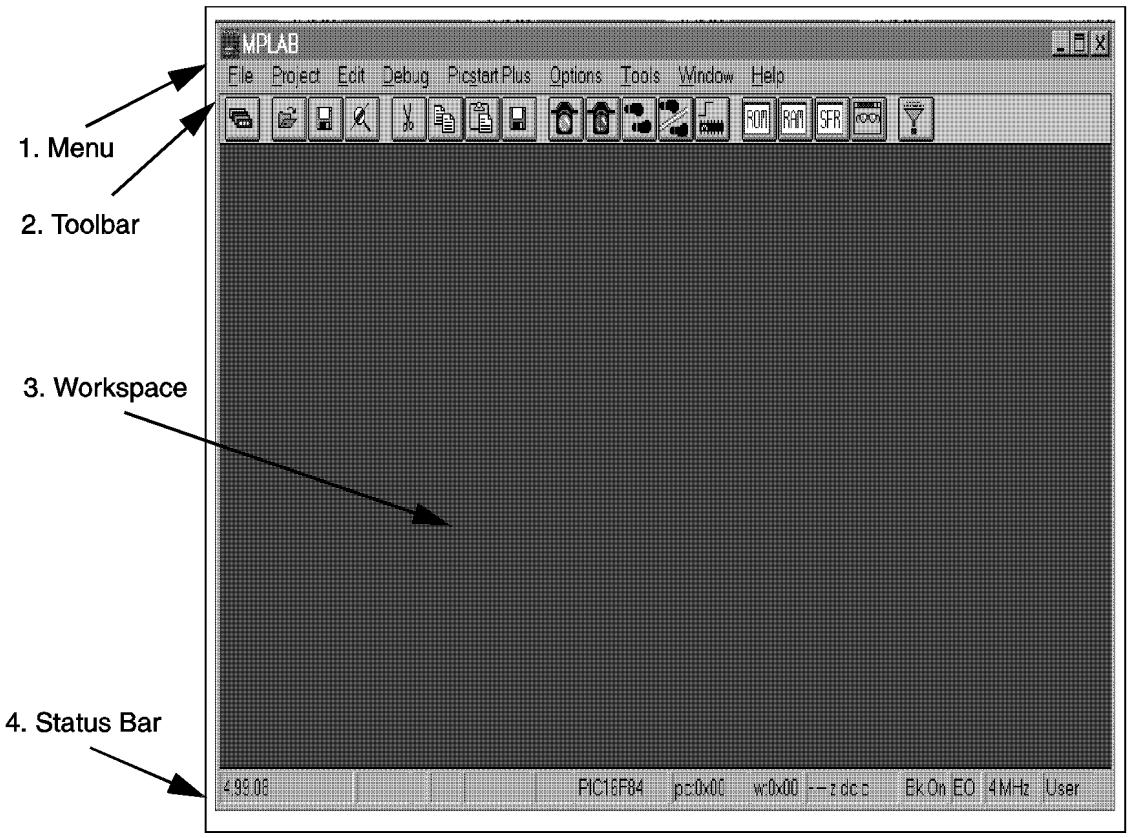


Figure 3.1: MPLAB IDE Desktop

The development mode sets which tool, if any, will execute code. For this tutorial we will use MPLAB-SIM, the software simulator. Later you may switch to one of the emulator operations if you have an emulator. Operation will be similar. "Editor Only" mode does not allow code execution, and is mainly useful if you have not installed the simulator, do not have an emulator, and are just creating code to program a PICmicro microcontroller (MCU).

Select the *Options > Development Mode* menu item and click the **Tools** tab to select the development tool and processor for your project.

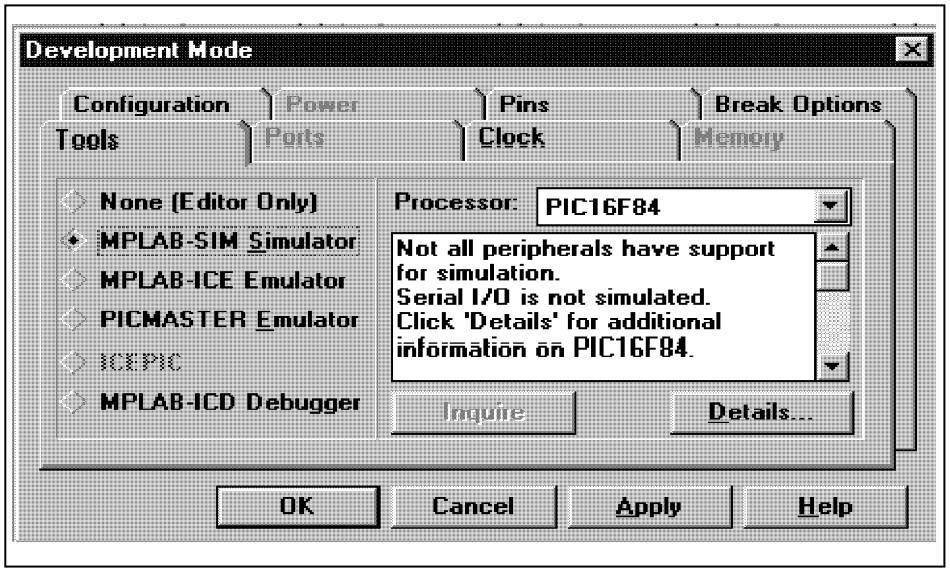


Figure 3.2: Development Mode Tools Dialog

MPLAB IDE is a constantly evolving product, so there may be subtle differences between what you see and the picture here. Select MPLAB-SIM Simulator and choose the PIC16F84 from the pull down list of available processors supported by the simulator. Click **OK**. The simulator will initialize and you should see "PIC16F84" and "Sim" in the status bar on the bottom of the MPLAB IDE desktop. You are now in simulator mode for the PIC16F84 device.

Creating a Simple New Project

The simulator runs from the same file (a hex file) that can be programmed into the PICmicro MCU. For the simulator to run you must first create a source code file and successfully assemble the source code.

The assembler produces, among other things, a hex file. This file has the file extension `.HEX`. In this tutorial the file will be named `tutor84.hex`. Later this file can be loaded directly into a device programmer without using the assembler or an MPLAB IDE project. This file can also be loaded by most other third party programmers.

Select *File > New* from the menu and you will see a dialog that looks like Figure 3.3.

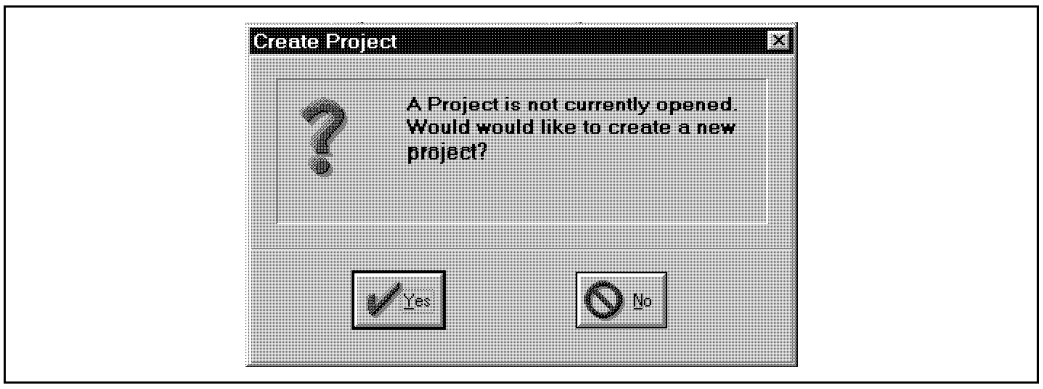


Figure 3.3: Create Project Dialog

Click **Yes**, and a standard Windows browsing dialog will appear (Figure 3.4). In this dialog, indicate the location where you want your project stored. Remember where you put it. You'll need this information later. This tutorial uses a directory in `c:\Program Files\MPLAB` and creates the project file named `tutor84.pjt`.

"PJT" is a standard suffix for MPLAB IDE project files. The prefix of the project file name, in this case `tutor84`, will become a default prefix for many of the files that MPLAB IDE will use or create for this tutorial.

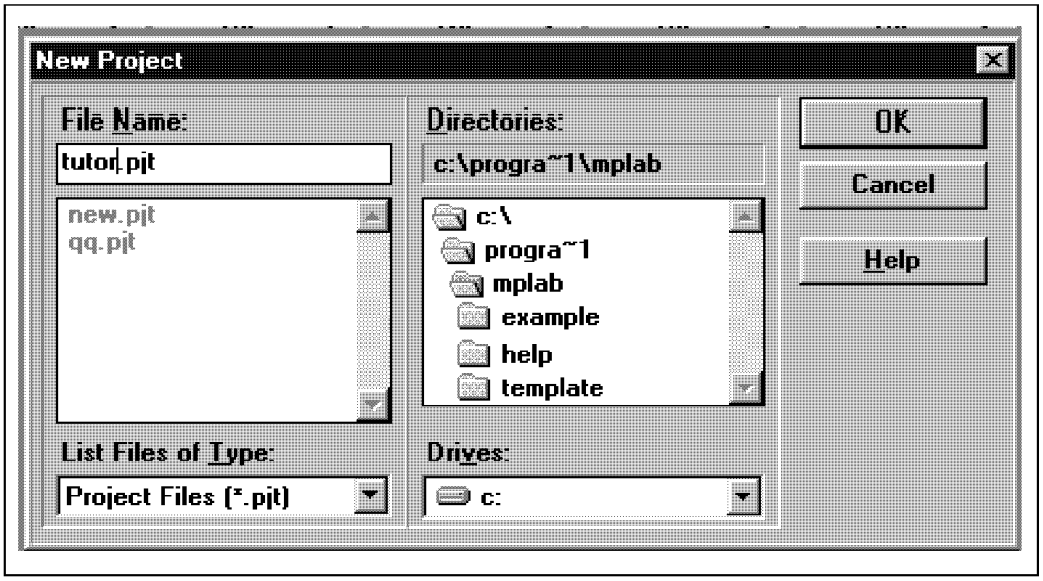


Figure 3.4: New Project Dialog

Using the mouse to click **OK** will bring you to the Edit Project Dialog (Figure 3.5).

The simulator, programmers, and emulator systems that work with MPLAB IDE use a hex file created by assembling, compiling, and/or linking source code. Several different tools can create hex files, and these tools are part of each project. Projects give you flexibility to describe how the application will be built and which software tools will be used to create the `.HEX` file. We will

not get into these details in this tutorial, but as you need these features you will be using the Node Properties to set these. See Chapter 4 for information on more complex projects.

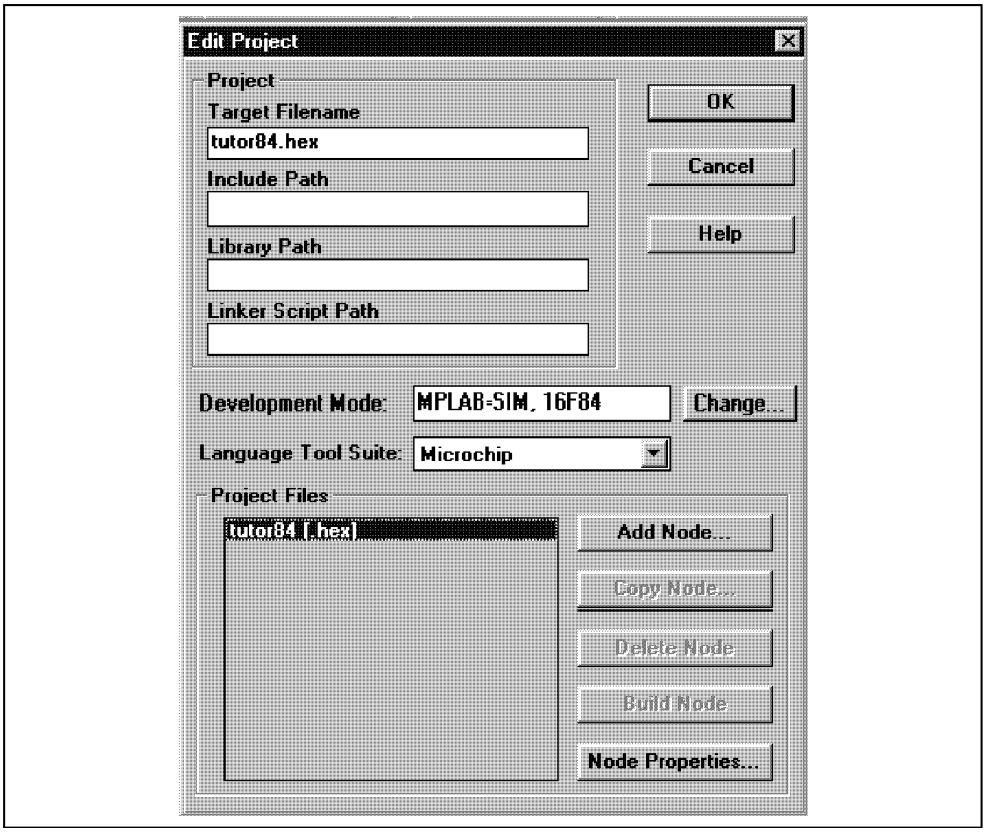


Figure 3.5: Edit Project Dialog – Node Properties Enabled

Notice that the target file name of the Edit Project dialog has been filled in for you. It uses the development mode that we set previously and defaults to using the Microchip language tool suite.

In addition, the default language suite, paths, and nodes for all projects are set selecting *Options > Environment Setup* and clicking the **Projects** tab. These defaults appear in the Edit Project dialog for all new projects.

In the Project Files window, you will find `tutor84.[hex]`. Highlighting this name will cause the Node Properties button to become usable.

Before we do anything else, we must tell MPLAB IDE how to create the hex file. Do this by clicking the **Node Properties** button. The Node Properties dialog will appear (Figure 3.6).

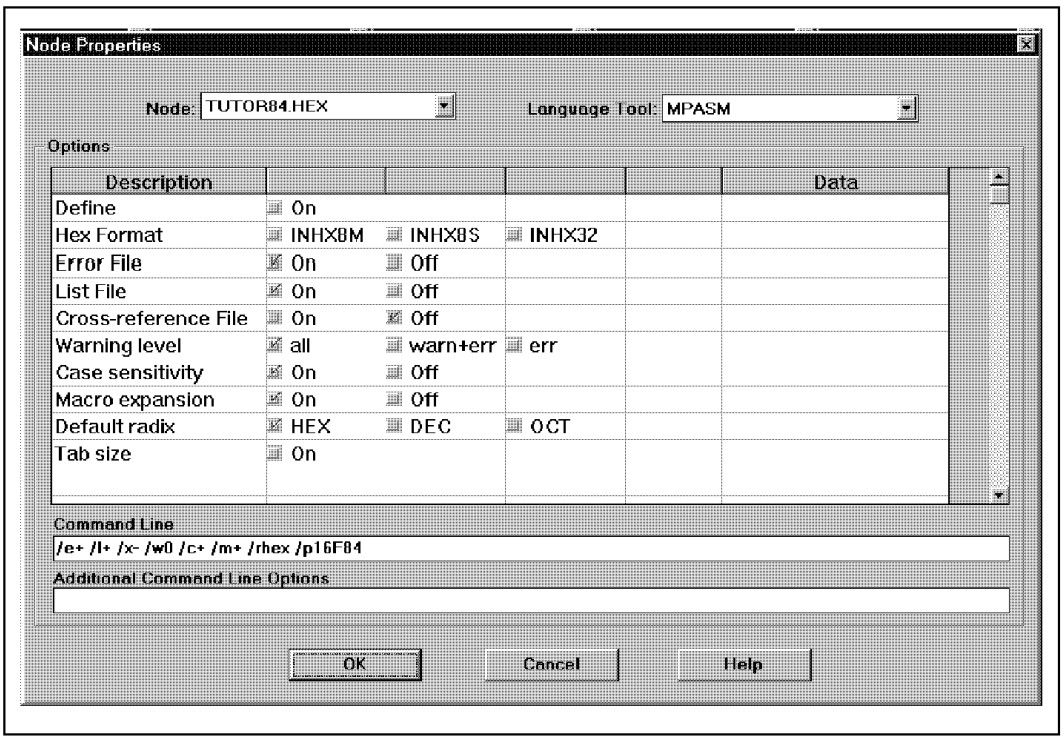


Figure 3.6: Node Properties Dialog

This dialog contains all of the default settings for the language tool shown in the upper right of the dialog (in this case MPASM). In the simplest form, a project contains a hex file created from one assembly source file. This is the default as the Node Properties dialog appears.

You can see that there are a lot of rows and columns on this dialog. Each row usually corresponds to a "switch," those things that are often set on the command line when the tool is invoked. In fact, the setting of these switches is reflected in the Command Line window near the bottom. This is the actual command line that will be issued to MPASM when it is invoked from MPLAB IDE.

For now you can use the default settings for all other entries, but as you become more familiar with building an application, you will probably find that you'll want to change some of these.

Clicking **OK** will apply these defaults, bring you back to the Edit Project dialog and enable the **Add Node** button (Figure 3.7).

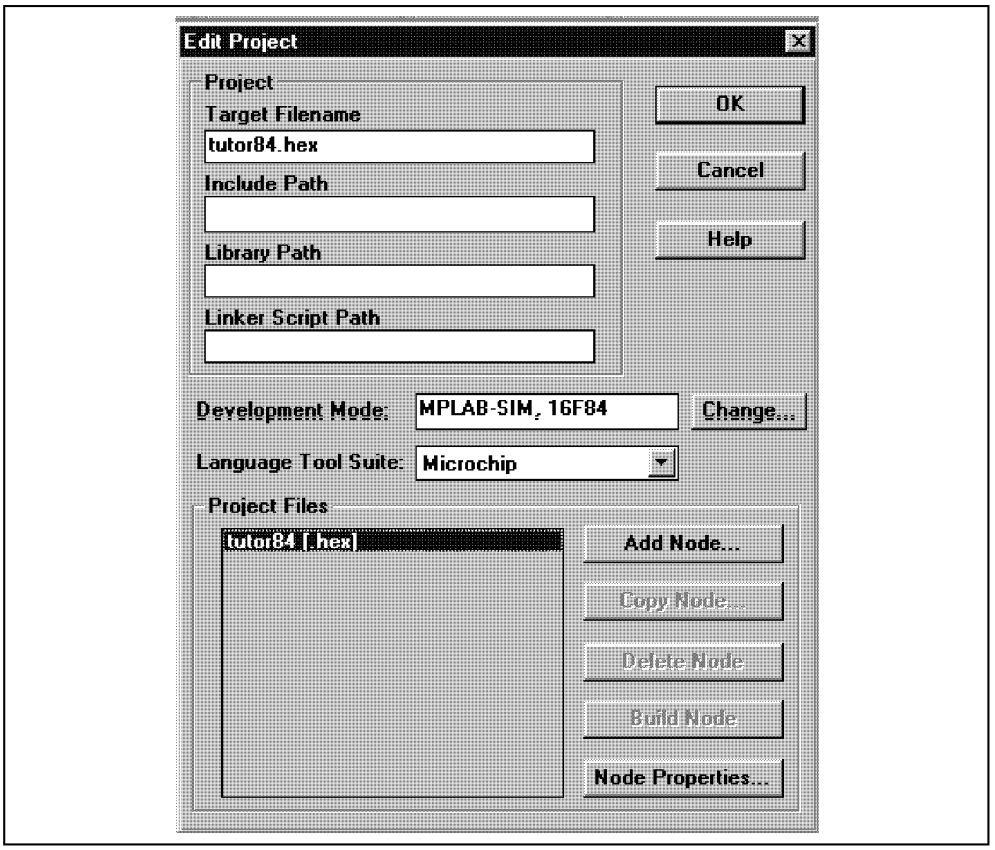


Figure 3.7: Edit Project Dialog – Add Node Enabled

Click **Add Node**. You will see the standard windows browse dialog (Figure 3.8), and the working directory will be the same as the project directory. Enter the file name `tutor84.asm`, and click **OK**.

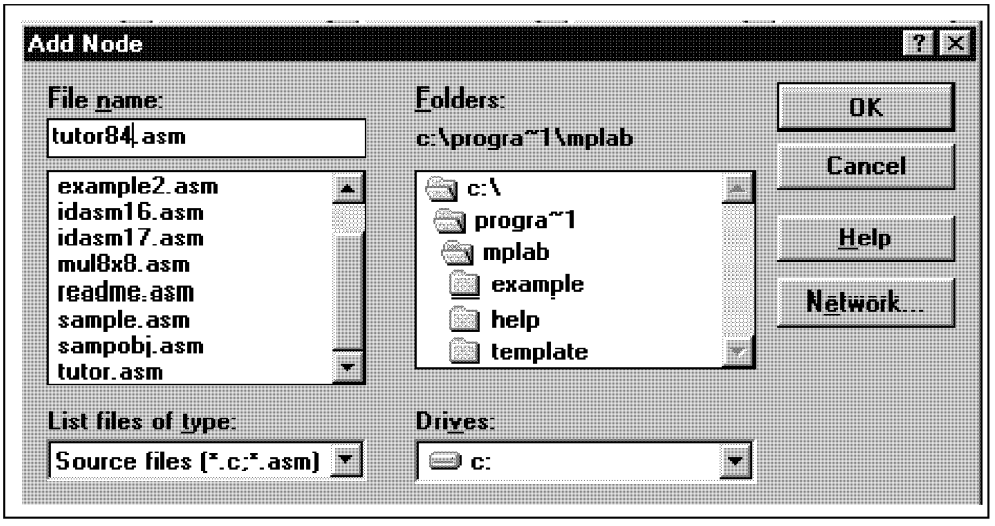


Figure 3.8: Add Node Dialog

Creating a Simple New Source File

Click once in the blank space of the empty file window that has been created for you. It is probably titled "Untitled." This gives the window "focus." Use the *File > Save As...* menu option and save the empty file as `tutor84.asm`. When the standard browse dialog opens, you will find `tutor84.asm` located in the current working directory of the project. Enter the file name and click **OK**.

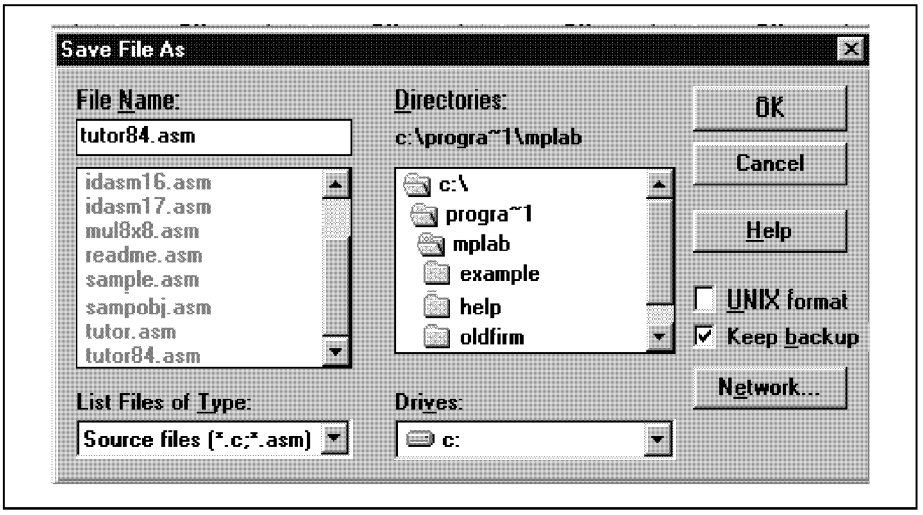


Figure 3.10: Save Source File

You will now be presented with the MPLAB IDE desktop and the empty file window, but the name of the file window will reflect its new name.

The name of the source file and the name of the project (`tutor84` in this tutorial) must be the same in this kind of project. If you change the name of the source file, you must also change the name of the project to match. Other projects that use the linker allow the output file name to be different from the input file (Section 4.6 provides a tutorial on creating projects using the linker).

Note: For a single source file project, MPASM creates its output hex file with the same name as the source file. The project name, hex file, and source file MUST have the same name.

Entering Source Code

Use the mouse to locate the cursor at the beginning of the `tutor84.asm` empty file window, and enter the following text, exactly as written on each line. You don't have to enter the comments, the text following the semicolons.

```
    list    p=16f84
    include <p16F84.inc>
c1  equ     0x0c    ; Set temp variable counter c1 at address 0x0c

    org     0x00    ; Set program memory base at reset vector 0x00
reset
    goto    start   ; Go to start of the main program

    org     0x04    ; Set program memory base to beginning of user code
start
    movlw   0x09    ; Initialize counter to arbitrary value greater than zero
    movwf   c1      ; Store value in temp variable a defined above
loop
    incfsz  c1,F    ; Increment counter, place results in file register
    goto    loop    ; Loop until counter overflows

    goto    bug     ; When counter overflows, got to start to re-initialize
    end
```

This code is a very simple program that increments a counter and resets to a predetermined value when the counter rolls over to zero.

All labels start in the first column, and the last line has an `end` directive. Refer to the *MPASM with MPLINK and MPLIB User's Guide* for more information about directives. The PICmicro MCU data sheets contain full information about instructions along with examples of their use.

Save the file by using the *File > Save* menu item.

Assembling the Source File

Assembling the file can be accomplished in several ways. The method described here uses the *Project > Build All* menu item. This will execute the MPASM assembler in the background using the defaults saved with the project as noted before. Once the assembly process is complete, the Build Results window will appear (Figure 3.11).

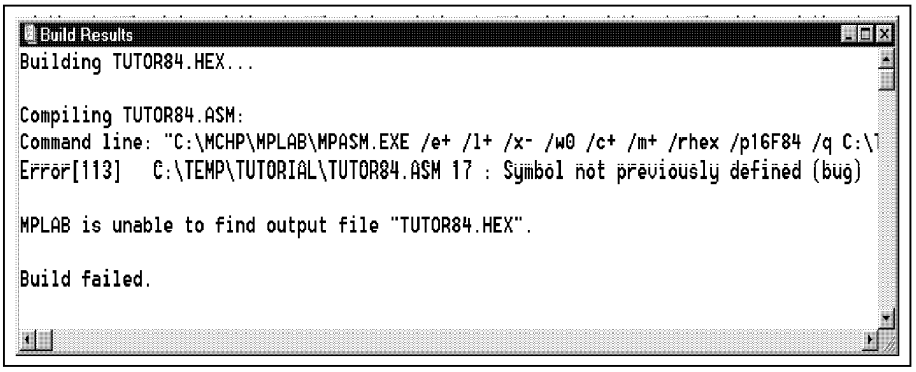


Figure 3.11: Build Results Window – Build Failed

You have intentionally entered at least one error if you entered the code as written in Section 3.6. The last `goto` in the program references a nonexistent label called `bug`. Since this label has not been defined before, the assembler reports an error. You may have other errors as well.

Using the mouse, double click on the error message. This will bring the cursor to the line in the source code that contains the error. Change `bug` to `start`. Use the Build Results window to help find the errors, and repair any other bugs in your source code. Reassemble by executing the *Project > Build All* menu function. This process may take a couple of iterations.

> **Note:** Whenever you rebuild a project all of your source files will be saved to disk.

When you've fixed all problems in the source code, the Build Results window will display "Build completed successfully" (Figure 3.12). You now have a complete project that can be executed using the simulator.

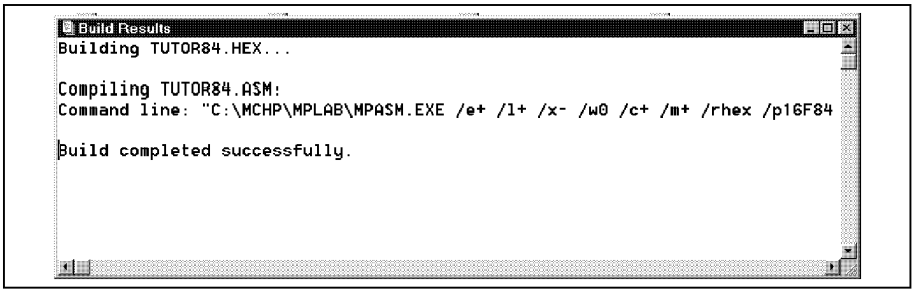


Figure 3.12: Build Results Window – Build Successful

Running Your Program

Use the *Debug > Run > Reset* to initialize the system. The program counter will be reset to zero, which is the reset vector on the PIC16F84. The source code line at this address may be highlighted with a dark bar. Also, you may notice that PC is set to 0x00 in the status bar at the bottom of MPLAB IDE.

Use the *Debug > Run > Step* menu item. This causes the program counter to advance to the next instruction location. The dark bar will follow the source code and the program counter displayed in the status bar should advance to "pc:0x04."

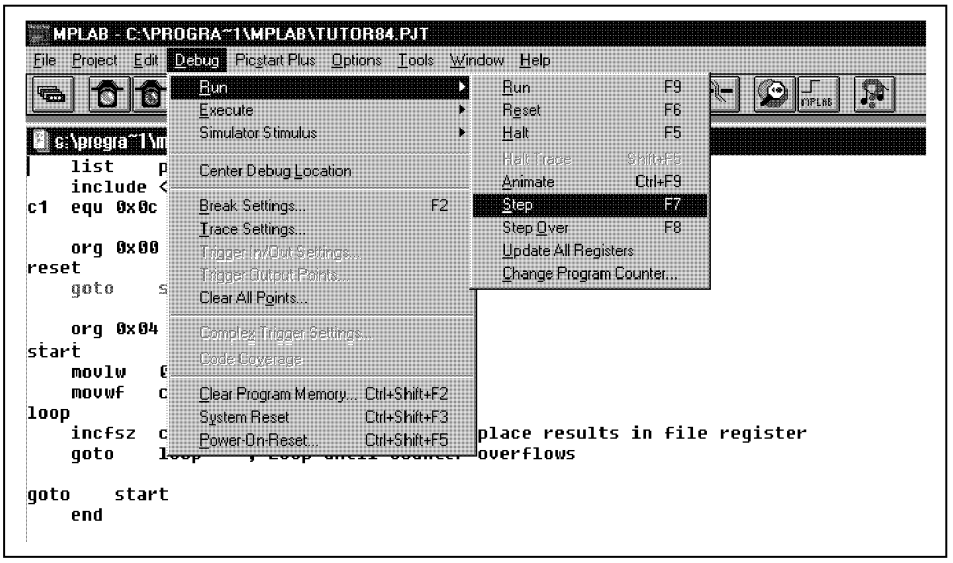


Figure 3.13: *Debug > Run > Step* Menu Item

You may notice as you execute the *Debug > Run > Step* menu item that there is text on the right side of the menu item that says <F7>. This stands for "function key seven" on your keyboard. Many MPLAB IDE functions are assigned to "shortcut keys." These keys have the same affect as executing the menu item itself. Press <F7> a few times and watch the program counter and dark bar advance through the program.

Execute the *Debug > Run > Run* menu item or press <F9> to start the program running from the current location counter. The status bar will change colors indicating the program is executing instructions. None of the other fields on the status bar will be updated until the program is halted.

Stop the program by executing the *Debug > Run > Halt* menu item or by pressing <F5>. The status bar will change back to its original color, and the current program counter and other status information will be updated.

Another way to execute functions is to use the tool bar at the top of the screen. If you place the cursor over the items in the tool bar, you can see the name of the function in the status bar at the bottom. The left button is a standard **Change Tool bar** button that allows you to scroll through the

available toolbars. Toolbars can be customized (see Section 7.8.5.1.4). On the debug toolbar, the green light is equivalent to <F9> (Run) and the red light is the same as <F5> (Halt).

Opening Other Windows for Debugging

There are many ways to look at your program and its execution using MPLAB IDE. For example, this program is intended to increment a temporary counter, but how do you know for sure that is happening? One way is to open and inspect the file register window. Do this by executing the *Window > File Registers* menu item. A small window with all of the file registers, or RAM, of the PIC16F84 will appear.

Press <F7> (Execute Single-Step) a few times and watch the values update in the file register window. We put the counter variable at address location 0x0C. As the temporary counter is incremented, this is reflected in the file register window. File registers change colors when their value changes so that they can easily be noticed on inspection. However, in very complex programs, many values may change, making it difficult to focus on one or two variables. This problem can be solved by using a Watch window.

Using a Watch Window

MPLAB IDE allows the contents of file registers to be monitored through a Watch window.

Creating a Watch Window

To create a Watch window, select *Window > Watch Window > New Watch Window*. If you have already created a Watch window and saved it to disk, select *Window > Watch Window > Load Watch Window*. Select the Watch window file to load and click **OK**, or double click the desired file.

The Add Watch Symbol dialog will appear (Figure 3.14).

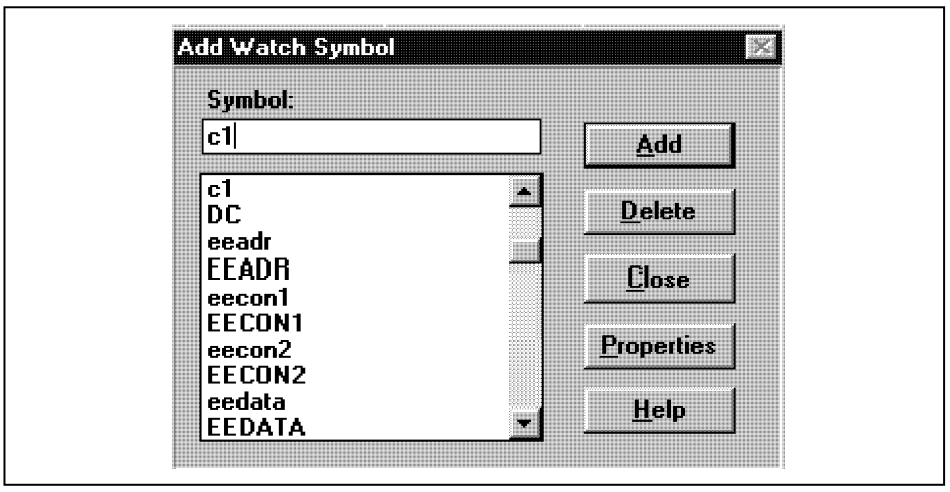


Figure 3.14: Add Watch Symbol Dialog

Typing ‘c1’ in the symbol name box will cause the list to scroll to the c1 symbol. Highlight the symbol and click the **Add** button, then click the **Close** button. You will be left with the Watch window on your MPLAB IDE desktop (Figure 3.15) displaying the current value of the temporary counter value ‘c1.

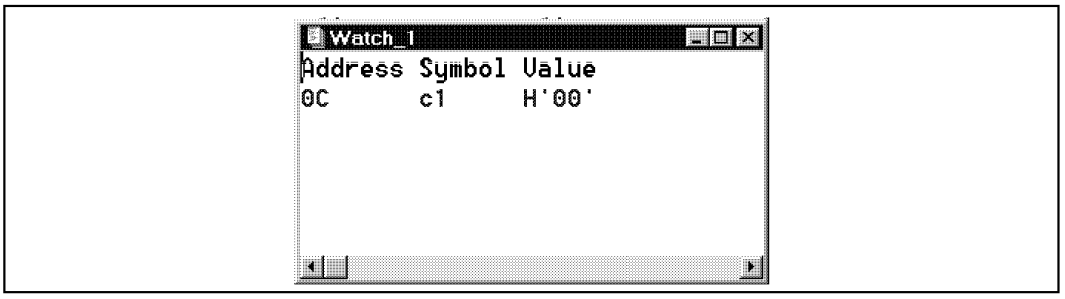


Figure 3.15: Watch Window

You can display the contents of the Watch window with or without line numbers. To change this setting, select *Toggle Line Numbers* from the system menu inside the Watch window.

Press <F7> to single step the program a few times and notice that as the counter value is incremented, the display is updated in the Watch window. If you’ve left the file register window open, it will update as well.

Saving the Watch Window

You can save the Watch window and its settings by selecting *Window > Watch WIndow > Save Active Watch* from the MPLAB IDE menu or by selecting *Save Watch* from the system menu inside the Watch window. (The system menu button is located in the upper left-hand corner of the Watch window. Clicking this button once will cause the menu underneath to cascade down.) Choose a name and click **OK**.

Figure 3.16: Save Watch Window Dialog

The window's open or closed status and location on the screen is saved with the project so the next time you open your project, your Watch windows will be restored as well.

Editing the Watch Window

You can also edit the Watch windows after you've created them.

Use either the *Window > Watch Window* submenu or the system menu inside the Watch window to edit the information in the Watch window.

Add a symbol to the Watch window	Select *Window > Watch Window > Add to Active Watch* from the MPLAB IDE menu or select **Add Watch** from the system menu inside the Watch window.
Delete a symbol from the Watch window	Click on the symbol in the Watch window, then select *Delete Watch* from the system menu.
Change the display format of symbols	Select *Window > Watch Window > Edit Active Watch* from the MPLAB IDE menu or select *Edit Watch* from the system menu inside the Watch window. Then, click **Properties**. The Properties dialog allows you to select the format, size, byte order, and display bits for display in the Watch Window.

Setting a Break Point

Press <F5> (*Debug > Run > Halt*) to make sure that the simulator processor is halted. Click in the source code window on the line immediately after the `start` label that says `movlw 0x09`. Click the right mouse button and a small shortcut menu will appear (Figure 3.17).

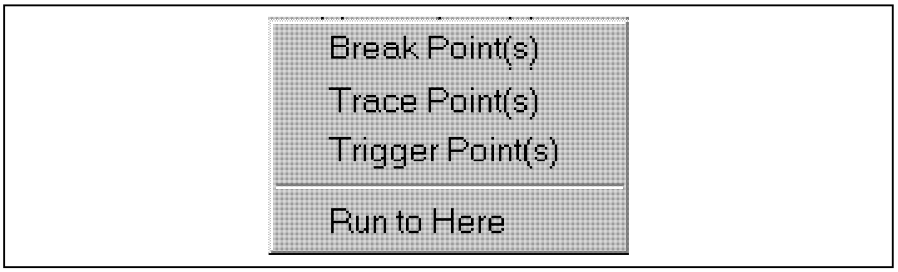


Figure 3.17: Right Mouse Button Pop-up Menu

Select the *Break Point(s)* menu item and the menu will disappear and the line where the cursor was located will change colors, indicating that a break point has been set at that location.

Press <F6> or execute the *Debug > Run > Reset* menu item to reset the system. Then run the system by pressing <F9>. The program will run and then halt at the instruction just after the break point. 'c1', as displayed in either the Watch window or the file register window (if one is still open), will reflect the reset status of zero; stepping once will execute the code and 'c1' will reflect a value of 0x09. Press <F9> a few times and notice the status bar change color while running, and will change again when the processor halts.

Note: If execution doesn't halt at the break point, select *Options > Development Mode* and click the **Break Options** tab. Make sure that Global Break Enable is selected (check marked).

Summary

This tutorial has shown you how to:

- set up a new project
- create and enter a source file into a project
- assemble code
- run your code using the simulator
- set break points and single step your code
- watch variables in your code

Once you are comfortable with the topics introduced here, you should look at the next section for more information on MPLAB IDE.

Some hints and tips:

Break Points – You can set break points in the *Window > Program Memory* window, in the source file window (in this case `tutor84.asm`), or in the *Window > Absolute Listing* window.

Source Files – Use the *Window > Project Window* to bring up a list of your source files. You can double click on the file name here to bring up that file in the editor.

MPASM Errors – If MPASM gives you an error, double click on that error in the error window to go to the error in the source code. If you've got multiple errors, always choose the first error. Often one error will cause subsequent errors and fixing the first one may fix them all.

Configuration Bits and Processor Mode – Configuration bits in the source file will not set the mode of the processor for the simulator (or emulators). For instance, the Watch Dog Timer Enable configuration bit can be set so that when you program a device, the Watch Dog Timer (WDT) will be turned on. You will also need to select *Options > Development Mode* and click the **Configuration** tab to enable the WDT for the simulator or emulator. This allows you to debug with it on or off without changing your source code. Use the *Options > Development Mode* Configuration tab to set the processor mode as well. Even though you can set these bits in your MPASM or MPLAB-CXX source file, MPLAB IDE does not automatically change modes.

Options – Go to *Options > Environment Setup* and click the **General** tab to do the following:

- Change the screen font or font size
- Position the tool bar on the side or bottom of the screen
- Modify the tool bar
- Change the number of characters displayed for labels

Before you close the dialog, click the **Key Mappings** tab to map European Keys to MPLAB IDE functions and special ASCII characters.

Map Files – Go to *Project > Edit Project* dialog and change MPASM's Node Properties to produce a MAP file named `tutor84.map`. After you've built the project, look at `tutor84.map` to see build information.

Grayed Out Menus – If you find menus "grayed out," check to make sure that you haven't somehow entered the Editor Only mode. If you're sure everything is set up correctly, try exiting MPLAB IDE and restarting the program.