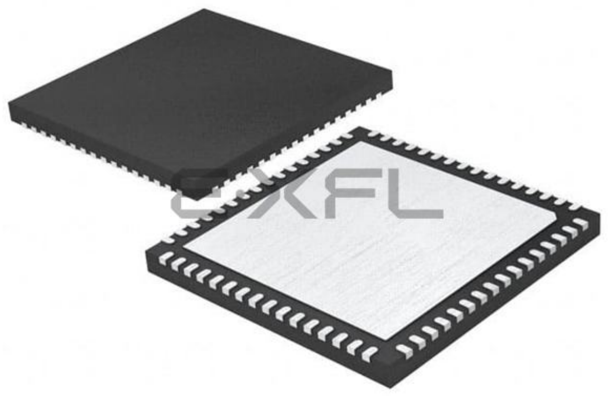
Microchip Technology - PIC18F65K40-E/MR Datasheet



What is "Embedded - Microcontrollers"?

"Embedded - Microcontrollers" refer to small, integrated circuits designed to perform specific tasks within larger systems. These microcontrollers are essentially compact computers on a single chip, containing a processor core, memory, and programmable input/output peripherals. They are called "embedded" because they are embedded within electronic devices to control various functions, rather than serving as standalone computers. Microcontrollers are crucial in modern electronics, providing the intelligence and control needed for a wide range of applications.

Applications of "Embedded - Microcontrollers"

Details	
Product Status	Active
Core Processor	PIC
Core Size	8-Bit
Speed	64MHz
Connectivity	I²C, LINbus, SPI, UART/USART
Peripherals	Brown-out Detect/Reset, POR, PWM, WDT
Number of I/O	60
Program Memory Size	32KB (16K x 16)
Program Memory Type	FLASH
EEPROM Size	1K x 8
RAM Size	2K x 8
Voltage - Supply (Vcc/Vdd)	2.3V ~ 5.5V
Data Converters	A/D 45x10b; D/A 1x5b
Oscillator Type	Internal
Operating Temperature	-40°C ~ 125°C (TA)
Mounting Type	Surface Mount
Package / Case	64-VFQFN Exposed Pad
Supplier Device Package	64-VQFN (9x9)
Purchase URL	https://www.e-xfl.com/product-detail/microchip-technology/pic18f65k40-e-mr

2.0 GUIDELINES FOR GETTING STARTED WITH PIC18(L)F6XK40 MICROCONTROLLERS

2.1 Basic Connection Requirements

Getting started with the PIC18(L)F6xK40 family of 8-bit microcontrollers requires attention to a minimal set of device pin connections before proceeding with development.

The following pins must always be connected:

- All VDD and VSS pins (see **Section 2.2 "Power Supply Pins"**)
- $\overline{\text{MCLR}}$ pin (see **Section 2.3 "Master Clear (MCLR) Pin"**)

These pins must also be connected if they are being used in the end application:

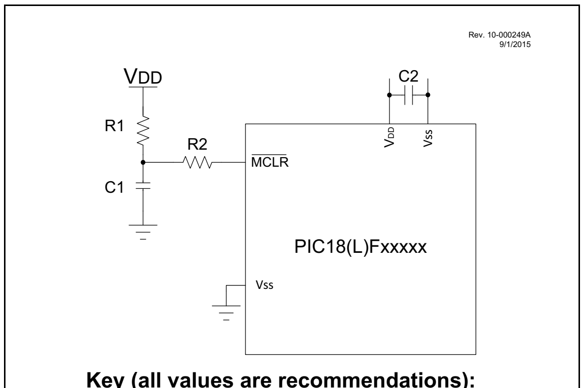
- PGC/PGD pins used for In-Circuit Serial Programming™ (ICSP™) and debugging purposes (see **Section 2.4 "ICSP™ Pins"**)
- OSCI and OSCO pins when an external oscillator source is used (see **Section 2.5 "External Oscillator Pins"**)

Additionally, the following pins may be required:

- VREF+/VREF- pins are used when external voltage reference for analog modules is implemented

The minimum mandatory connections are shown in Figure 2-1.

FIGURE 2-1: RECOMMENDED MINIMUM CONNECTIONS



Key (all values are recommendations):
C1 and C2 : 0.1 μF, 20V ceramic
R1: 10 kΩ
R2: 100Ω to 470Ω

2.2 Power Supply Pins

2.2.1 DECOUPLING CAPACITORS

The use of decoupling capacitors on every pair of power supply pins (VDD and VSS) is required.

Consider the following criteria when using decoupling capacitors:

- **Value and type of capacitor:** A 0.1 μF (100 nF), 10-20V capacitor is recommended. The capacitor should be a low-ESR device, with a resonance frequency in the range of 200 MHz and higher. Ceramic capacitors are recommended.
- **Placement on the printed circuit board:** The decoupling capacitors should be placed as close to the pins as possible. It is recommended to place the capacitors on the same side of the board as the device. If space is constricted, the capacitor can be placed on another layer on the PCB using a via; however, ensure that the trace length from the pin to the capacitor is no greater than 0.25 inch (6 mm).
- **Handling high-frequency noise:** If the board is experiencing high-frequency noise (upward of tens of MHz), add a second ceramic type capacitor in parallel to the above described decoupling capacitor. The value of the second capacitor can be in the range of 0.01 μF to 0.001 μF. Place this second capacitor next to each primary decoupling capacitor. In high-speed circuit designs, consider implementing a decade pair of capacitances as close to the power and ground pins as possible (e.g., 0.1 μF in parallel with 0.001 μF).
- **Maximizing performance:** On the board layout from the power supply circuit, run the power and return traces to the decoupling capacitors first, and then to the device pins. This ensures that the decoupling capacitors are first in the power chain. Equally important is to keep the trace length between the capacitor and the power pins to a minimum, thereby reducing PCB trace inductance.

2.2.2 TANK CAPACITORS

On boards with power traces running longer than six inches in length, it is suggested to use a tank capacitor for integrated circuits, including microcontrollers, to supply a local power source. The value of the tank capacitor should be determined based on the trace resistance that connects the power supply source to the device, and the maximum current drawn by the device in the application. In other words, select the tank capacitor so that it meets the acceptable voltage sag at the device. Typical values range from 4.7 μF to 47 μF.

8.12 Start-up Sequence

Upon the release of a POR or BOR, the following must occur before the device will begin executing:

1. Power-up Timer runs to completion (if enabled).
2. Oscillator start-up timer runs to completion (if required for selected oscillator source).
3. $\overline{\text{MCLR}}$ must be released (if enabled).

The total time out will vary based on oscillator configuration and Power-up Timer configuration. See **Section 4.0 "Oscillator Module (with Fail-Safe Clock Monitor)"** for more information.

The Power-up Timer and oscillator start-up timer run independently of $\overline{\text{MCLR}}$ Reset. If $\overline{\text{MCLR}}$ is kept low long enough, the Power-up Timer and oscillator Start-up Timer will expire. Upon bringing $\overline{\text{MCLR}}$ high, the device will begin execution after 10 FOSC cycles (see Figure 8-4). This is useful for testing purposes or to synchronize more than one device operating in parallel.

FIGURE 8-4: RESET START-UP SEQUENCE

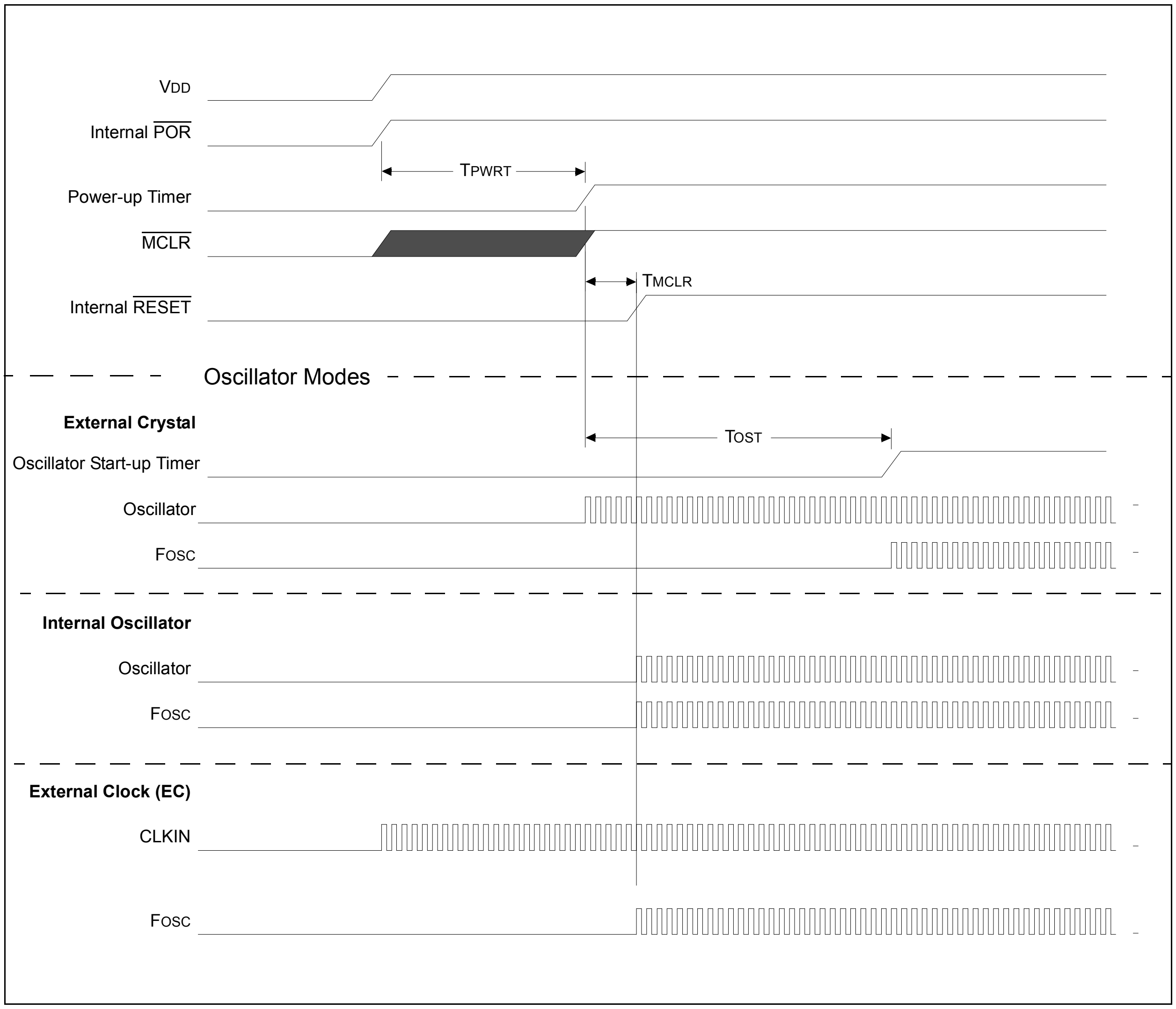
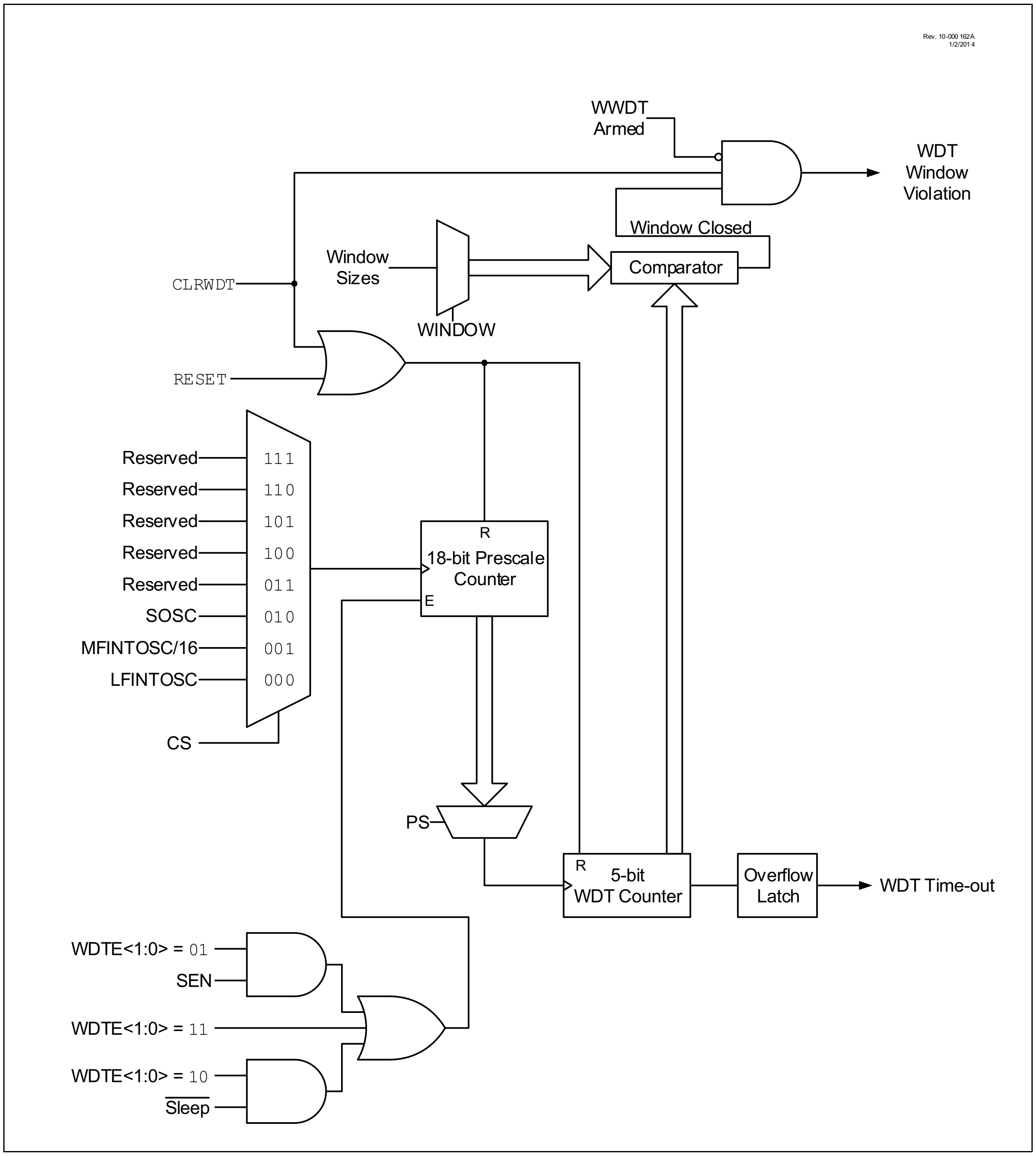


FIGURE 9-1: WINDOWED WATCHDOG TIMER BLOCK DIAGRAM



10.1.1 PROGRAM COUNTER

The Program Counter (PC) specifies the address of the instruction to fetch for execution. The PC is 21 bits wide and is contained in three separate 8-bit registers. The low byte, known as the PCL register, is both readable and writable. The high byte, or PCH register, contains the PC<15:8> bits; it is not directly readable or writable. Updates to the PCH register are performed through the PCLATH register. The upper byte is called PCU. This register contains the PC<20:16> bits; it is also not directly readable or writable. Updates to the PCU register are performed through the PCLATU register.

The contents of PCLATH and PCLATU are transferred to the program counter by any operation that writes PCL. Similarly, the upper two bytes of the program counter are transferred to PCLATH and PCLATU by an operation that reads PCL. This is useful for computed offsets to the PC (see **Section 10.2.3.1 "Computed GOTO"**).

The PC addresses bytes in the program memory. To prevent the PC from becoming misaligned with word instructions, the Least Significant bit of PCL is fixed to a value of '0'. The PC increments by two to address sequential instructions in the program memory.

The CALL, RCALL, GOTO and program branch instructions write to the program counter directly. For these instructions, the contents of PCLATH and PCLATU are not transferred to the program counter.

10.1.2 RETURN ADDRESS STACK

The return address stack allows any combination of up to 31 program calls and interrupts to occur. The PC is pushed onto the stack when a CALL or RCALL instruction is executed or an interrupt is Acknowledged. The PC value is pulled off the stack on a RETURN, RETLW or a RETFIE instruction. PCLATU and PCLATH are not affected by any of the RETURN or CALL instructions.

The stack operates as a 31-word by 21-bit RAM and a 5-bit Stack Pointer, or as a 35-word by 21-bit RAM with a 6-bit Stack Pointer in ICD mode. The stack space is not part of either program or data space. The Stack Pointer is readable and writable and the address on the top of the stack is readable and writable through the Top-of-Stack (TOS) Special File registers. Data can also be pushed to, or popped from the stack, using these registers.

A CALL type instruction causes a push onto the stack; the Stack Pointer is first incremented and the location pointed to by the Stack Pointer is written with the contents of the PC (already pointing to the instruction following the CALL). A RETURN type instruction causes a pop from the stack; the contents of the location pointed to by the STKPTR are transferred to the PC and then the Stack Pointer is decremented.

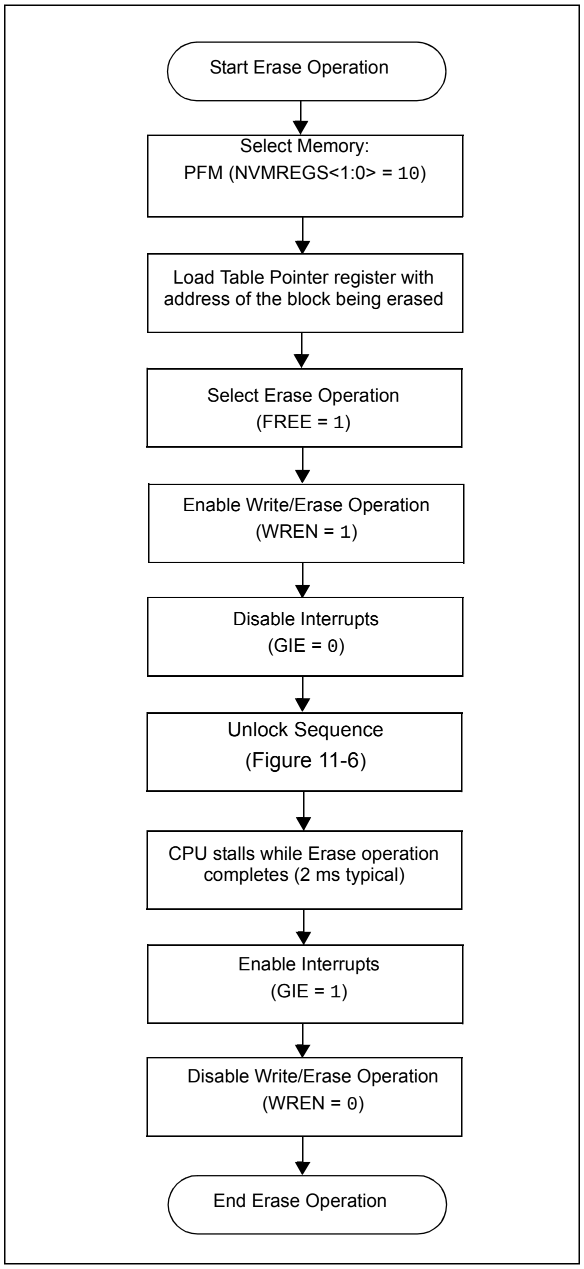
The Stack Pointer is initialized to '00000' after all Resets. There is no RAM associated with the location corresponding to a Stack Pointer value of '00000'; this is only a Reset value. Status bits in the PCON0 register indicate if the stack is full or has overflowed or has underflowed.

10.1.2.1 Top-of-Stack Access

Only the top of the return address stack (TOS) is readable and writable. A set of three registers, TOSU:TOSH:TOSL, hold the contents of the stack location pointed to by the STKPTR register (Figure 10-1). This allows users to implement a software stack if necessary. After a CALL, RCALL or interrupt, the software can read the pushed value by reading the TOSU:TOSH:TOSL registers. These values can be placed on a user defined software stack. At return time, the software can return these values to TOSU:TOSH:TOSL and do a return.

The user must disable the Global Interrupt Enable (GIE) bits while accessing the stack to prevent inadvertent stack corruption.

FIGURE 11-7: PFM ROW ERASE FLOWCHART



11.1.6 WRITING TO PROGRAM FLASH MEMORY

The programming write block size is described in Table 11-3. Word or byte programming is not supported. Table writes are used internally to load the holding registers needed to program the Flash memory. There are only as many holding registers as there are bytes in a write block. Refer to Table 11-3 for write latch size.

Since the table latch (TABLAT) is only a single byte, the TBLWT instruction needs to be executed multiple times for each programming operation. The write protection state is ignored for this operation. All of the table write operations will essentially be short writes because only the holding registers are written. NVMIF is not affected while writing to the holding registers.

After all the holding registers have been written, the programming operation of that block of memory is started by configuring the NVMCON1 register for a program memory write and performing the long write sequence.

If the PFM address in the TBLPTR is write-protected or if TBLPTR points to an invalid location, the WR bit is cleared without any effect and the WREER is signaled.

The long write is necessary for programming the internal Flash. CPU operation is suspended during a long write cycle and resumes when the operation is complete. The long write operation completes in one instruction cycle. When complete, WR is cleared in hardware and NVMIF is set and an interrupt will occur if NVMIE is also set. The latched data is reset to all '1s'. WREN is not changed.

The internal programming timer controls the write time. The write/erase voltages are generated by an on-chip charge pump, rated to operate over the voltage range of the device.

Note:	The default value of the holding registers on device Resets and after write operations is FFh. A write of FFh to a holding register does not modify that byte. This means that individual bytes of program memory may be modified, provided that the change does not attempt to change any bit from a '0' to a '1'. When modifying individual bytes, it is not necessary to load all holding registers before executing a long write operation.

REGISTER 14-10: PIR8: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 8

R/W-0/0	R/W-0/0	R/W-0/0	U-0	U-0	U-0	U-0	R/W-0/0
SCANIF	CRCIF	NVMIF	—	—	—	—	CWG1IF
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 7	**SCANIF:** Memory Scanner Interrupt Flag bit 1 = Interrupt has occurred (must be cleared by software) 0 = Interrupt event has not occurred
bit 6	**CRCIF:** CRC Interrupt Flag bit 1 = Interrupt has occurred (must be cleared by software) 0 = Interrupt event has not occurred
bit 5	**NVMIF:** NVM Interrupt Flag bit 1 = Interrupt has occurred (must be cleared by software) 0 = Interrupt event has not occurred
bit 4-1	**Unimplemented:** Read as ‘0’
bit 0	**CWG1IF:** CWG1 Interrupt Flag bit 1 = Interrupt has occurred (must be cleared by software) 0 = Interrupt event has not occurred

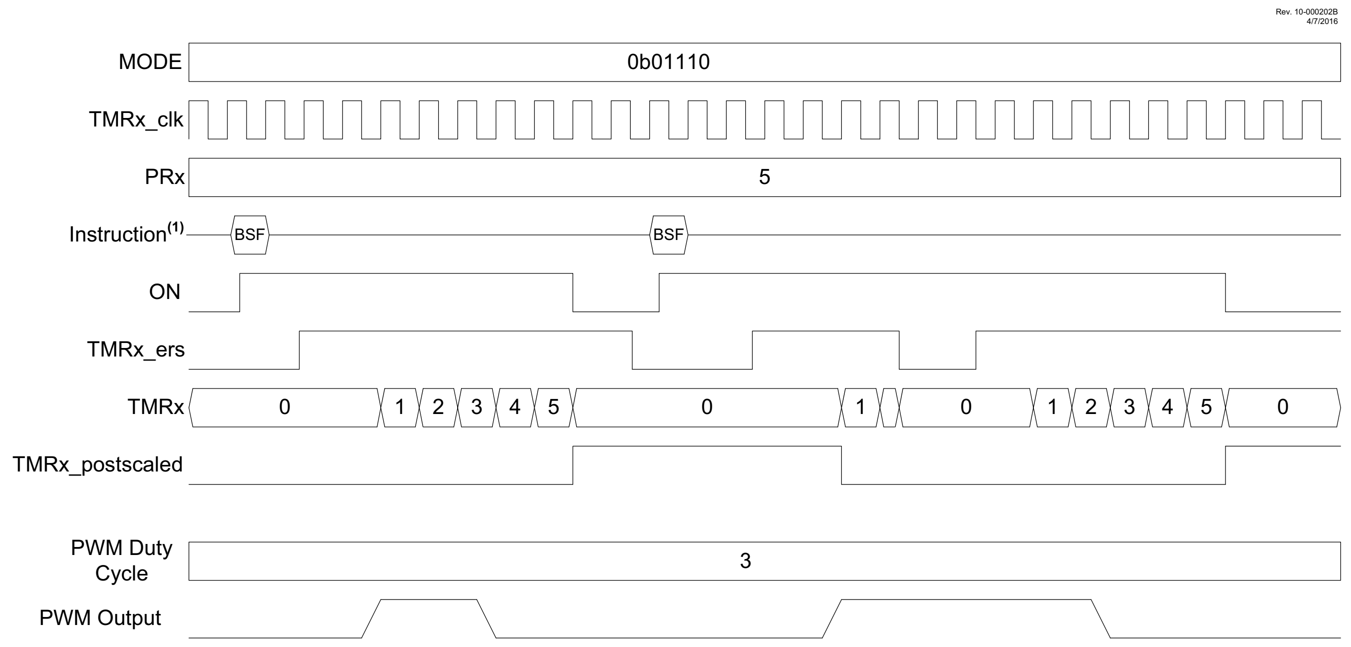
REGISTER 14-29: IPR7: PERIPHERAL INTERRUPT PRIORITY REGISTER 7

U-0	U-0	U-0	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
—	—	—	CCP5IP	CCP4IP	CCP3IP	CCP2IP	CCP1IP
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 7-5	**Unimplemented:** Read as ‘0’
bit 4	**CCP5IP:** ECCP5 Interrupt Priority bit 1 = High priority 0 = Low priority
bit 3	**CCP4IP:** ECCP4 Interrupt Priority bit 1 = High priority 0 = Low priority
bit 2	**CCP3IP:** ECCP3 Interrupt Priority bit 1 = High priority 0 = Low priority
bit 1	**CCP2IP:** ECCP2 Interrupt Priority bit 1 = High priority 0 = Low priority
bit 0	**CCP1IP:** ECCP1 Interrupt Priority bit 1 = High priority 0 = Low priority

FIGURE 20-11: LOW LEVEL RESET, EDGE-TRIGGERED HARDWARE LIMIT ONE-SHOT MODE TIMING DIAGRAM (MODE = `01110`)



Note 1: BSF and BCF represent Bit-Set File and Bit-Clear File instructions executed by the CPU to set or clear the ON bit of TxCON. CPU execution is asynchronous to the timer clock input.

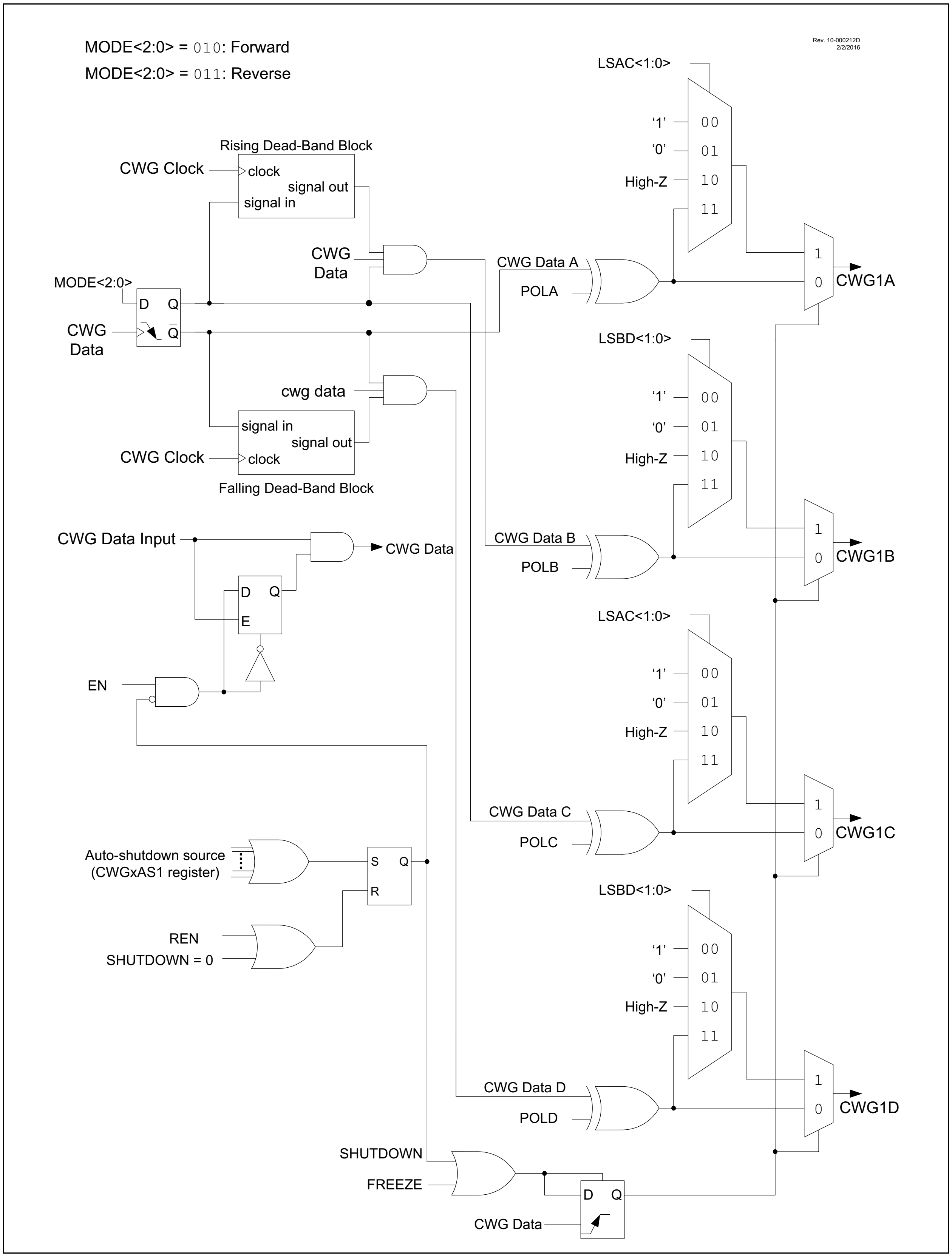
REGISTER 22-2: CCPTMRS1: CCP TIMERS CONTROL REGISTER 1

U-0	U-0	R/W-0/0	R/W-1/1	R/W-0/0	R/W-1/1	R/W-0/0	R/W-1/1
—	—	P7TSEL<1:0>		P6TSEL<1:0>		C5TSEL<1:0>	
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-6	**Unimplemented:** Read as '0'
bit 5-4	**P7TSEL<1:0>:** PWM7 Timer Selection bits 11 = PWM7 based on TMR8 10 = PWM7 based on TMR6 01 = PWM7 based on TMR4 00 = PWM7 based on TMR2
bit 3-2	**P6TSEL<1:0>:** PWM6 Timer Selection bits 11 = PWM6 based on TMR8 10 = PWM6 based on TMR6 01 = PWM6 based on TMR4 00 = PWM6 based on TMR2
bit 1-0	**C5TSEL<1:0>:** CCP5 Timer Selection bits 11 = CCP5 is based off Timer7 in Capture/Compare mode and Timer8 in PWM mode 10 = CCP5 is based off Timer5 in Capture/Compare mode and Timer6 in PWM mode 01 = CCP5 is based off Timer3 in Capture/Compare mode and Timer4 in PWM mode 00 = CCP5 is based off Timer1 in Capture/Compare mode and Timer2 in PWM mode

FIGURE 24-6: SIMPLIFIED CWG BLOCK DIAGRAM (FORWARD AND REVERSE FULL-BRIDGE MODES)



REGISTER 25-5: SMTxWIN: SMTx WINDOW INPUT SELECT REGISTER

U-0	U-0	U-0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
—	—	—	WSEL<4:0>				
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
‘1’ = Bit is set	‘0’ = Bit is cleared	q = Value depends on condition

bit 7-5 **Unimplemented**: Read as ‘0’

bit 4-0 **WSEL<4:0>:** SMTx Window Selection bits

WSEL	SMT1 Window Source	SMT2 Window Source
11111-10110	Reserved	Reserved
10101	ZCDOUT	ZCDOUT
10100	C3OUT	C3OUT
10011	C2OUT	C2OUT
10010	C1OUT	C1OUT
10001	PWM7OUT	PWM7OUT
10000	PWM6OUT	PWM6OUT
01111	CCP5OUT	CCP5OUT
01110	CCP4OUT	CCP4OUT
01101	CCP3OUT	CCP3OUT
01100	CCP2OUT	CCP2OUT
01011	CCP1OUT	CCP1OUT
01010	SMT2 overflow	Reserved
01001	Reserved	SMT1 overflow
01000	TMR8_postscaler	TMR8_postscaler
00111	TMR6_postscaler	TMR6_postscaler
00110	TMR4_postscaler	TMR4_postscaler
00101	TMR2_postscaler	TMR2_postscaler
00100	TMR0_overflow	TMR0_overflow
00011	SOSC	SOSC
00010	MFINTOSC (31 kHz)	MFINTOSC (31 kHz)
00001	LFINTOSC (31 kHz)	LFINTOSC (31 kHz)
00000	Pin selected by SMT1WINPPS	Pin selected by SMT2WINPPS

25.7.8 CAPTURE MODE

This mode captures the Timer value based on a rising or falling edge on the SMTWINx input and triggers an interrupt. This mimics the capture feature of a CCP module. The timer begins incrementing upon the SMTxGO bit being set, and updates the value of the SMTxCPR register on each rising edge of SMTWINx, and updates the value of the CPW register on each falling edge of the SMTWINx. The timer is not reset by any hardware conditions in this mode and must be reset by software, if desired. See Figure 25-16 and Figure 25-17.

28.5.2.3 EUSART Synchronous Slave Reception

The operation of the Synchronous Master and Slave modes is identical (**Section 28.5.1.5 "Synchronous Master Reception"**), with the following exceptions:

- Sleep
- CREN bit is always set, therefore the receiver is never idle
- SREN bit, which is a "don't care" in Slave mode

A character may be received while in Sleep mode by setting the CREN bit prior to entering Sleep. Once the word is received, the RSR register will transfer the data to the RCxREG register. If the RCxIE enable bit is set, the interrupt generated will wake the device from Sleep and execute the next instruction. If the GIE bit is also set, the program will branch to the interrupt vector.

28.5.2.4 Synchronous Slave Reception Setup:

1. Set the SYNC and SPEN bits and clear the CSRC bit.
2. Clear the ANSEL bit for both the CKx and DTx pins (if applicable).
3. If interrupts are desired, set the RCxIE bit of the PIE3/4 registers and the GIE and PEIE bits of the INTCON register.
4. If 9-bit reception is desired, set the RX9 bit.
5. Set the CREN bit to enable reception.
6. The RCxIF bit will be set when reception is complete. An interrupt will be generated if the RCxIE bit was set.
7. If 9-bit mode is enabled, retrieve the Most Significant bit from the RX9D bit of the RCxSTA register.
8. Retrieve the eight Least Significant bits from the receive FIFO by reading the RCxREG register.
9. If an overrun error occurs, clear the error by either clearing the CREN bit of the RCxSTA register or by clearing the SPEN bit which resets the EUSART.

TABLE 28-10: SUMMARY OF REGISTERS ASSOCIATED WITH SYNCHRONOUS SLAVE RECEPTION

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Register on Page
BAUDxCON	ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	451
INTCON	GIE/GIEH	PEIE/GIEL	IPEN	—	INT3EDG	INT2EDG	INT1EDG	INT0EDG	173
PIE3	RC2IE	TX2IE	RC1IE	TX1IE	BCL2IE	SSP2IE	BCL1IE	SSP1IE	188
PIR3	RC2IF	TX2IF	RC1IF	TX1IF	BCL2IF	SSP2IF	BCL1IF	SSP1IF	177
IPR3	RC2IP	TX2IP	RC1IP	TX1IP	BCL2IP	SSP2IP	BCL1IP	SSP1IP	198
PIE4	—	—	RC5IE	TX5IE	RC4IE	TX4IE	RC3IE	TX3IE	189
PIR4	—	—	RC5IF	TX5IF	RC4IF	TX4IF	RC3IF	TX3IF	177
IPR4	—	—	RC5IP	TX5IP	RC4IP	TX4IP	RC3IP	TX3IP	199
RCxREG	EUSART Receive Data Register								455*
RCxSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	450
RxyPPS	—	—	RxyPPS<5:0>						228
RXxPPS	—	—	RXPPS<5:0>						225
TXxSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	449

Legend: — = unimplemented location, read as '0'. Shaded cells are not used for synchronous slave reception.
* Page provides register information.

28.6 EUSART Operation During Sleep

The EUSART will remain active during Sleep only in the Synchronous Slave mode. All other modes require the system clock and therefore cannot generate the necessary signals to run the Transmit or Receive Shift registers during Sleep.

Synchronous Slave mode uses an externally generated clock to run the Transmit and Receive Shift registers.

28.6.1 SYNCHRONOUS RECEIVE DURING SLEEP

To receive during Sleep, all the following conditions must be met before entering Sleep mode:

- RCxSTA and TXxSTA Control registers must be configured for Synchronous Slave Reception (see **Section 28.5.2.4 “Synchronous Slave Reception Setup:”**).
- If interrupts are desired, set the RCxIE bit of the PIE3/4 registers and the GIE and PEIE bits of the INTCON register.
- The RCxIF interrupt flag must be cleared by reading RCxREG to unload any pending characters in the receive buffer.

Upon entering Sleep mode, the device will be ready to accept data and clocks on the RXx/DTx and TXx/CKx pins, respectively. When the data word has been completely clocked in by the external device, the RCxIF interrupt flag bit of the PIR3/4 registers will be set. Thereby, waking the processor from Sleep.

Upon waking from Sleep, the instruction following the SLEEP instruction will be executed. If the Global Interrupt Enable (GIE) bit of the INTCON register is also set, then the Interrupt Service Routine at address 004h will be called.

28.6.2 SYNCHRONOUS TRANSMIT DURING SLEEP

To transmit during Sleep, all the following conditions must be met before entering Sleep mode:

- The RCxSTA and TXxSTA Control registers must be configured for synchronous slave transmission (see **Section 28.5.2.2 “Synchronous Slave Transmission Setup”**).
- The TXxIF interrupt flag must be cleared by writing the output data to the TXxREG, thereby filling the TSR and transmit buffer.
- If interrupts are desired, set the TXxIE bit of the PIE3/4 registers and the PEIE bit of the INTCON register.
- Interrupt enable bits TXxIE of the PIE3 register and PEIE of the INTCON register must set.

Upon entering Sleep mode, the device will be ready to accept clocks on TXx/CKx pin and transmit data on the RXx/DTx pin. When the data word in the TSR has been completely clocked out by the external device, the pending byte in the TXxREG will transfer to the TSR and the TXxIF flag will be set. Thereby, waking the processor from Sleep. At this point, the TXxREG is available to accept another character for transmission, which will clear the TXxIF flag.

Upon waking from Sleep, the instruction following the SLEEP instruction will be executed. If the Global Interrupt Enable (GIE) bit is also set then the Interrupt Service Routine at address 0004h will be called.

32.2.6 ADC CONVERSION PROCEDURE (BASIC MODE)

This is an example procedure for using the ADC to perform an Analog-to-Digital conversion:

1. Configure Port:
   - Disable pin output driver (Refer to the TRISx register)
   - Configure pin as analog (Refer to the ANSELx register)
2. Configure the ADC module:
   - Select ADC conversion clock
   - Configure voltage reference
   - Select ADC input channel (precharge+acquisition)
   - Turn on ADC module
3. Configure ADC interrupt (optional):
   - Clear ADC interrupt flag
   - Enable ADC interrupt
   - Enable peripheral interrupt (PEIE bit)
   - Enable global interrupt (GIE bit)(1)
4. If ADACQ = `0`, software must wait the required acquisition time(2).
5. Start conversion by setting the ADGO bit.
6. Wait for ADC conversion to complete by one of the following:
   - Polling the ADGO bit
   - Waiting for the ADC interrupt (interrupts enabled)
7. Read ADC Result.
8. Clear the ADC interrupt flag (required if interrupt is enabled).

Note 1: The global interrupt can be disabled if the user is attempting to wake-up from Sleep and resume in-line code execution.

2: Refer to **Section 32.3 "ADC Acquisition Requirements"**.

EXAMPLE 32-1: ADC CONVERSION

```
;This code block configures the ADC
;for polling, VDD and VSS references, FRC
;oscillator and AN0 input.
;
;Conversion start & polling for completion
;are included.
;
BANKSEL   ADCON1       ;
MOVLW     B'11110000'  ;Right justify,
                       ;FRC oscillator
MOVWF     ADCON1       ;Vdd and Vss Vref
BANKSEL   TRISA        ;
BSF       TRISA,0      ;Set RA0 to input
BANKSEL   ANSEL        ;
BSF       ANSEL,0      ;Set RA0 to analog
BANKSEL   ADCON0       ;
MOVLW     B'00000001'  ;Select channel AN0
MOVWF     ADCON0       ;Turn ADC On
CALL      SampleTime   ;Acquisiton delay
BSF       ADCON0,ADGO  ;Start conversion
BTFSC     ADCON0,ADGO  ;Is conversion done?
GOTO      $-1          ;No, test again
BANKSEL   ADRESH       ;
MOVF      ADRESH,W     ;Read upper 2 bits
MOVWF     RESULTHI     ;store in GPR space
BANKSEL   ADRESL       ;
MOVF      ADRESL,W     ;Read lower 8 bits
MOVWF     RESULTLO     ;Store in GPR space
```

TABLE 36-2: INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
BYTE-ORIENTED OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	`0010`	`01da`	`ffff`	`ffff`	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and CARRY bit to f	1	`0010`	`00da`	`ffff`	`ffff`	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	`0001`	`01da`	`ffff`	`ffff`	Z, N	1,2
CLRF	f, a	Clear f	1	`0110`	`101a`	`ffff`	`ffff`	Z	2
COMF	f, d, a	Complement f	1	`0001`	`11da`	`ffff`	`ffff`	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	`0110`	`001a`	`ffff`	`ffff`	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	`0110`	`010a`	`ffff`	`ffff`	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	`0110`	`000a`	`ffff`	`ffff`	None	1, 2
DECF	f, d, a	Decrement f	1	`0000`	`01da`	`ffff`	`ffff`	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	`0010`	`11da`	`ffff`	`ffff`	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	`0100`	`11da`	`ffff`	`ffff`	None	1, 2
INCF	f, d, a	Increment f	1	`0010`	`10da`	`ffff`	`ffff`	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	`0011`	`11da`	`ffff`	`ffff`	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	`0100`	`10da`	`ffff`	`ffff`	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	`0001`	`00da`	`ffff`	`ffff`	Z, N	1, 2
MOVF	f, d, a	Move f	1	`0101`	`00da`	`ffff`	`ffff`	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word f_d (destination) 2nd word	2	`1100` `1111`	`ffff` `ffff`	`ffff` `ffff`	`ffff` `ffff`	None	
MOVWF	f, a	Move WREG to f	1	`0110`	`111a`	`ffff`	`ffff`	None	
MULWF	f, a	Multiply WREG with f	1	`0000`	`001a`	`ffff`	`ffff`	None	1, 2
NEGF	f, a	Negate f	1	`0110`	`110a`	`ffff`	`ffff`	C, DC, Z, OV, N	
RLCF	f, d, a	Rotate Left f through Carry	1	`0011`	`01da`	`ffff`	`ffff`	C, Z, N	1, 2
RLNCF	f, d, a	Rotate Left f (No Carry)	1	`0100`	`01da`	`ffff`	`ffff`	Z, N	
RRCF	f, d, a	Rotate Right f through Carry	1	`0011`	`00da`	`ffff`	`ffff`	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	`0100`	`00da`	`ffff`	`ffff`	Z, N	
SETF	f, a	Set f	1	`0110`	`100a`	`ffff`	`ffff`	None	1, 2
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	`0101`	`01da`	`ffff`	`ffff`	C, DC, Z, OV, N	
SUBWF	f, d, a	Subtract WREG from f	1	`0101`	`11da`	`ffff`	`ffff`	C, DC, Z, OV, N	1, 2
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	`0101`	`10da`	`ffff`	`ffff`	C, DC, Z, OV, N	
SWAPF	f, d, a	Swap nibbles in f	1	`0011`	`10da`	`ffff`	`ffff`	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	`0110`	`011a`	`ffff`	`ffff`	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	`0001`	`10da`	`ffff`	`ffff`	Z, N	

Note 1: When a PORT register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), the value used will be that value present on the pins themselves. For example, if the data latch is '`1`' for a pin configured as input and is driven low by an external device, the data will be written back with a '`0`'.

2: If this instruction is executed on the TMR0 register (and where applicable, 'd' = `1`), the prescaler will be cleared if assigned.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOP`.

4: Some instructions are two-word instructions. The second word of these instructions will be executed as a `NOP` unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

BNOV Branch if Not Overflow

Syntax:	BNOV n
Operands:	-128 ≤ n ≤ 127
Operation:	if OVERFLOW bit is ‘0’ (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	`1110` `0101` `nnnn` `nnnn`
Description:	If the OVERFLOW bit is ‘0’, then the program will branch. The 2’s complement number ‘2n’ is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a 2-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal ‘n’	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal ‘n’	Process Data	No operation

Example: `HERE      BNOV  Jump`

Before Instruction

PC	=	address (HERE)

After Instruction

If OVERFLOW	=	0;
PC	=	address (Jump)
If OVERFLOW	=	1;
PC	=	address (HERE + 2)

BNZ Branch if Not Zero

Syntax:	BNZ n
Operands:	-128 ≤ n ≤ 127
Operation:	if ZERO bit is ‘0’ (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	`1110` `0001` `nnnn` `nnnn`
Description:	If the ZERO bit is ‘0’, then the program will branch. The 2’s complement number ‘2n’ is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a 2-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal ‘n’	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal ‘n’	Process Data	No operation

Example: `HERE      BNZ  Jump`

Before Instruction

PC	=	address (HERE)

After Instruction

If ZERO	=	0;
PC	=	address (Jump)
If ZERO	=	1;
PC	=	address (HERE + 2)

36.2 Extended Instruction Set

In addition to the standard 75 instructions of the PIC18 instruction set, PIC18(L)F6xK40 devices also provide an optional extension to the core CPU functionality. The added features include eight additional instructions that augment indirect and indexed addressing operations and the implementation of Indexed Literal Offset Addressing mode for many of the standard PIC18 instructions.

The additional features of the extended instruction set are disabled by default. To enable them, users must set the XINST Configuration bit.

The instructions in the extended set can all be classified as literal operations, which either manipulate the File Select Registers, or use them for indexed addressing. Two of the instructions, `ADDFSR` and `SUBFSR`, each have an additional special instantiation for using FSR2. These versions (`ADDULNK` and `SUBULNK`) allow for automatic return after execution.

The extended instructions are specifically implemented to optimize re-entrant program code (that is, code that is recursive or that uses a software stack) written in high-level languages, particularly C. Among other things, they allow users working in high-level languages to perform certain operations on data structures more efficiently. These include:

- dynamic allocation and deallocation of software stack space when entering and leaving subroutines
- function pointer invocation
- software Stack Pointer manipulation
- manipulation of variables located in a software stack

A summary of the instructions in the extended instruction set is provided in Table 36-3. Detailed descriptions are provided in **Section 36.2.2 "Extended Instruction Set"**. The opcode field descriptions in Table 36-1 apply to both the standard and extended PIC18 instruction sets.

Note:	The instruction set extension and the Indexed Literal Offset Addressing mode were designed for optimizing applications written in C; the user may likely never use these instructions directly in assembler. The syntax for these commands is provided as a reference for users who may be reviewing code that has been generated by a compiler.

36.2.1 EXTENDED INSTRUCTION SYNTAX

Most of the extended instructions use indexed arguments, using one of the File Select Registers and some offset to specify a source or destination register. When an argument for an instruction serves as part of indexed addressing, it is enclosed in square brackets ("[ ]"). This is done to indicate that the argument is used as an index or offset. MPASM™ Assembler will flag an error if it determines that an index or offset value is not bracketed.

When the extended instruction set is enabled, brackets are also used to indicate index arguments in byte-oriented and bit-oriented instructions. This is in addition to other changes in their syntax. For more details, see **Section 36.2.3.1 "Extended Instruction Syntax with Standard PIC18 Commands"**.

Note:	In the past, square brackets have been used to denote optional arguments in the PIC18 and earlier instruction sets. In this text and going forward, optional arguments are denoted by braces ("{ }").

TABLE 36-3: EXTENSIONS TO THE PIC18 INSTRUCTION SET

Mnemonic, Operands		Description		Cycles	16-Bit Instruction Word				Status Affected
					MSb			LSb	
ADDFSR	f, k	Add literal to FSR		1	1110	1000	ffkk	kkkk	None
ADDULNK	k	Add literal to FSR2 and return		2	1110	1000	11kk	kkkk	None
CALLW		Call subroutine using WREG		2	0000	0000	0001	0100	None
MOVSF	z_s, f_d	Move z_s (source) to	1st word	2	1110	1011	0zzz	zzzz	None
		f_d (destination)	2nd word		1111	ffff	ffff	ffff	
MOVSS	z_s, z_d	Move z_s (source) to	1st word	2	1110	1011	1zzz	zzzz	None
		z_d (destination)	2nd word		1111	xxxx	xzzz	zzzz	
PUSHL	k	Store literal at FSR2, decrement FSR2		1	1110	1010	kkkk	kkkk	None
SUBFSR	f, k	Subtract literal from FSR		1	1110	1001	ffkk	kkkk	None
SUBULNK	k	Subtract literal from FSR2 and return		2	1110	1001	11kk	kkkk	None

CALLW — Subroutine Call Using WREG

Syntax:	CALLW
Operands:	None
Operation:	(PC + 2) $\rightarrow$ TOS, (W) $\rightarrow$ PCL, (PCLATH) $\rightarrow$ PCH, (PCLATU) $\rightarrow$ PCU
Status Affected:	None
Encoding:	`0000` `0000` `0001` `0100`
Description	First, the return address (PC + 2) is pushed onto the return stack. Next, the contents of W are written to PCL; the existing value is discarded. Then, the contents of PCLATH and PCLATU are latched into PCH and PCU, respectively. The second cycle is executed as a `NOP` instruction while the new next instruction is fetched. Unlike `CALL`, there is no option to update W, Status or BSR.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read WREG	PUSH PC to stack	No operation
No operation	No operation	No operation	No operation

Example: `HERE CALLW`

Before Instruction

PC	=	address `(HERE)`
PCLATH	=	10h
PCLATU	=	00h
W	=	06h

After Instruction

PC	=	001006h
TOS	=	address `(HERE + 2)`
PCLATH	=	10h
PCLATU	=	00h
W	=	06h

MOVSF — Move Indexed to f

Syntax:	MOVSF [z_s], f_d
Operands:	$0 \le z_s \le 127$ $0 \le f_d \le 4095$
Operation:	$((FSR2) + z_s) \rightarrow f_d$
Status Affected:	None

Encoding:

1st word (source)	`1110`	`1011`	`0zzz`	`zzzz`$_s$
2nd word (destin.)	`1111`	`ffff`	`ffff`	`ffff`$_d$

Description:	The contents of the source register are moved to destination register 'f_d'. The actual address of the source register is determined by adding the 7-bit literal offset 'z_s' in the first word to the value of FSR2. The address of the destination register is specified by the 12-bit literal 'f_d' in the second word. Both addresses can be anywhere in the 4096-byte data space (000h to FFFh). The `MOVSF` instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register. If the resultant source address points to an indirect addressing register, the value returned will be 00h.
Words:	2
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Determine source addr	Determine source addr	Read source reg
Decode	No operation No dummy read	No operation	Write register 'f' (dest)

Example: `MOVSF [05h], REG2`

Before Instruction

FSR2	=	80h
Contents of 85h	=	33h
REG2	=	11h

After Instruction

FSR2	=	80h
Contents of 85h	=	33h
REG2	=	33h

37.2 MPLAB XC Compilers

The MPLAB XC Compilers are complete ANSI C compilers for all of Microchip's 8, 16, and 32-bit MCU and DSC devices. These compilers provide powerful integration capabilities, superior code optimization and ease of use. MPLAB XC Compilers run on Windows, Linux or MAC OS X.

For easy source level debugging, the compilers provide debug information that is optimized to the MPLAB X IDE.

The free MPLAB XC Compiler editions support all devices and commands, with no time or memory restrictions, and offer sufficient code optimization for most applications.

MPLAB XC Compilers include an assembler, linker and utilities. The assembler generates relocatable object files that can then be archived or linked with other relocatable object files and archives to create an executable file. MPLAB XC Compiler uses the assembler to produce its object file. Notable features of the assembler include:

- Support for the entire device instruction set
- Support for fixed-point and floating-point data
- Command-line interface
- Rich directive set
- Flexible macro language
- MPLAB X IDE compatibility

37.3 MPASM Assembler

The MPASM Assembler is a full-featured, universal macro assembler for PIC10/12/16/18 MCUs.

The MPASM Assembler generates relocatable object files for the MPLINK Object Linker, Intel® standard HEX files, MAP files to detail memory usage and symbol reference, absolute LST files that contain source lines and generated machine code, and COFF files for debugging.

The MPASM Assembler features include:

- Integration into MPLAB X IDE projects
- User-defined macros to streamline assembly code
- Conditional assembly for multipurpose source files
- Directives that allow complete control over the assembly process

37.4 MPLINK Object Linker/ MPLIB Object Librarian

The MPLINK Object Linker combines relocatable objects created by the MPASM Assembler. It can link relocatable objects from precompiled libraries, using directives from a linker script.

The MPLIB Object Librarian manages the creation and modification of library files of precompiled code. When a routine from a library is called from a source file, only the modules that contain that routine will be linked in with the application. This allows large libraries to be used efficiently in many different applications.

The object linker/library features include:

- Efficient linking of single libraries instead of many smaller files
- Enhanced code maintainability by grouping related modules together
- Flexible creation of libraries with easy module listing, replacement, deletion and extraction

37.5 MPLAB Assembler, Linker and Librarian for Various Device Families

MPLAB Assembler produces relocatable machine code from symbolic assembly language for PIC24, PIC32 and dsPIC DSC devices. MPLAB XC Compiler uses the assembler to produce its object file. The assembler generates relocatable object files that can then be archived or linked with other relocatable object files and archives to create an executable file. Notable features of the assembler include:

- Support for the entire device instruction set
- Support for fixed-point and floating-point data
- Command-line interface
- Rich directive set
- Flexible macro language
- MPLAB X IDE compatibility