
PIC1000: Getting Started with Writing C-Code for PIC16 and PIC18

Introduction

Authors: Cristian Săbiuță, Cristina Ionescu, Microchip Technology Inc.

This technical brief provides the steps recommended to successfully program a PIC16 or PIC18 microcontroller and defines coding guidelines to help write more readable and reusable code.

High-level programming languages are a necessity due to imposed short development time and high-quality requirements. These languages make it easier to maintain and reuse code due to better portability and readability than the low-level instructions specific for each microcontroller architecture.

Programming language alone does not ensure high readability and reusability, but good coding style does. Therefore, the PIC® peripherals, header files and drivers are designed according to this presumption.

Since the most widely used high-level programming language for PIC microcontrollers is C, this document will focus on C programming. To ensure compatibility with most PIC C compilers, the code examples in this document are written using ANSI C coding standard.

Table of Contents

Introduction	1
1. Data Sheet Module Structure and Naming Conventions.....	3
1.1. How to Find the Data Sheet	3
1.2. Pin Description.....	3
1.3. Modules Description.....	4
1.4. Naming Conventions.....	6
1.5. PIC® Configuration Bits.....	8
2. Modules Representation in Header Files.....	9
2.1. Registers Representation in Header Files.....	9
2.2. Bit Masks and Bit Field Masks.....	10
2.3. Bit Positions.....	12
3. Writing Bare Metal C-Code for PIC.....	13
3.1. Set, Clear and Read Register Bits.....	13
3.2. Register Initialization.....	15
3.3. Change Register Bit Field Configurations.....	18
3.4. Setting Configuration Bits.....	19
4. Application Example Showing Alternative Ways of Writing Code.....	21
4.1. Turn ON an LED on a Button Press using Bit Unions.....	21
4.2. Turn ON an LED on a Button Press using Bit Masks.....	21
4.3. Turn ON an LED on a Button Press using Bit Positions.....	22
5. Further Steps.....	23
5.1. MPLAB® X and XC8 Compiler.....	23
5.2. Application Notes and Technical Briefs.....	23
6. Conclusion.....	24
7. References.....	25
The Microchip Website.....	26
Product Change Notification Service.....	26
Customer Support.....	26
Microchip Devices Code Protection Feature.....	26
Legal Notice.....	26
Trademarks.....	27
Quality Management System.....	27
Worldwide Sales and Service.....	28

1. Data Sheet Module Structure and Naming Conventions

The first step in writing C-code for a microcontroller is knowing and understanding the type of information found in the data sheet of the device used for programming. The data sheet contains information about the features, memories, core and peripheral modules of the microcontroller. Also included in the data sheet are the functional descriptions and base addresses of the peripheral modules, the names and addresses of the registers and other functional and electrical characteristics.

1.1 How to Find the Data Sheet

The data sheet for the PIC16F and PIC18F microcontroller families can be found at:

- [PIC16F1779](#)
- [PIC16F18846](#)
- [PIC16F18877](#)
- [PIC18F45Q43](#)
- [PIC18F47Q10](#)
- [PIC18F47K42](#)

Any documentation related to Microchip products can be found at:

- [Microchip Data Sheet](#) page

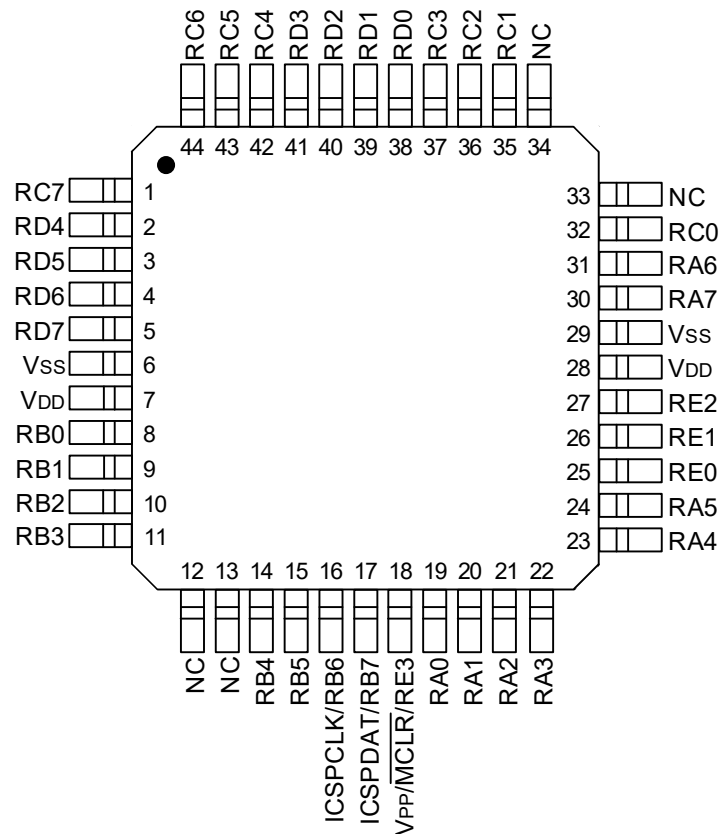
Or by searching the respective part number on:

- www.microchip.com

1.2 Pin Description

The pin description can be found in any device data sheet. The pinout is contained in the data sheet's *Pin Diagrams* chapter, which covers various devices depending on the pin or package specifications. The TQFP pinout of the PIC18-Q10 44-pin devices is presented in [Figure 1-1](#).

Figure 1-1. PIC18-Q10 44-Pin TQFP Pinout



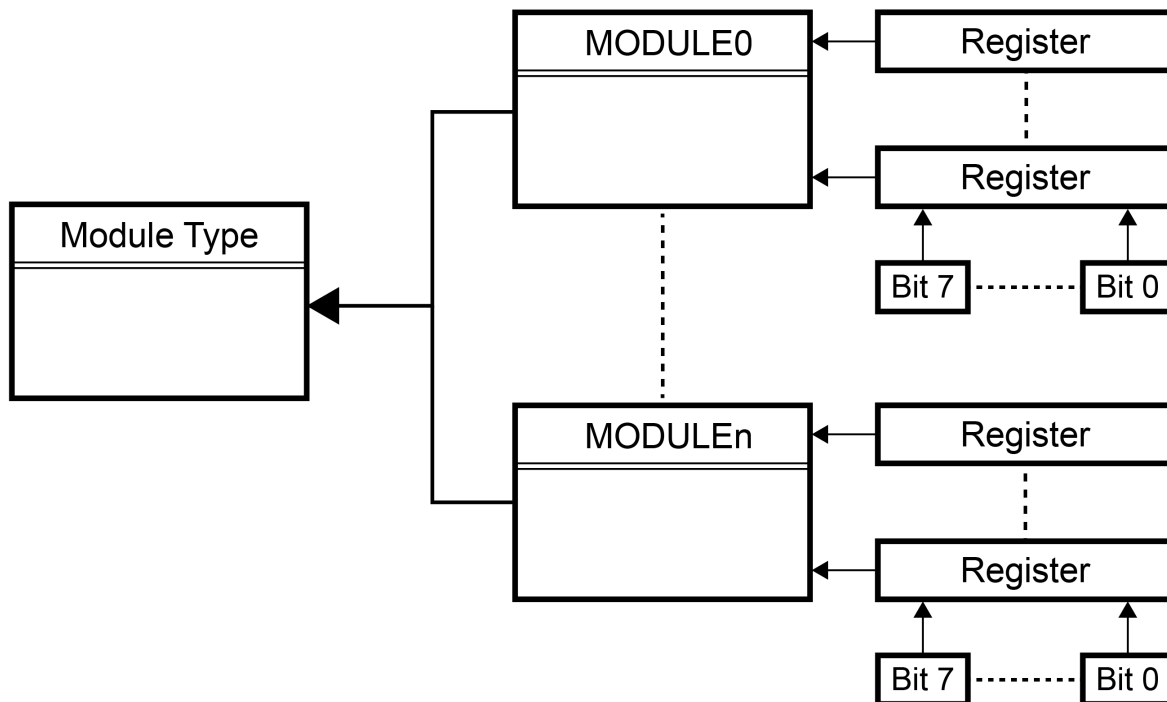
The *Pin Allocation Tables* chapter from the [PIC18F27/47Q10 data sheet](#) contains information about the pre-established pin functions. These functionalities that can be configured for each I/O pin are usually input or output of peripheral modules instances. This information varies depending on the device number of pins. If an evaluation board is used, such as the PIC18F47Q10 Curiosity Nano Development Platform, the routing of the pins on the specific board must be known. The information is available at [PIC18F47Q10 Curiosity Nano](#) page.

1.3 Modules Description

A PIC microcontroller is comprised of several building blocks: a PIC CPU core, SRAM, Flash, EEPROM, and various peripheral modules called module types. Throughout this document, all peripheral modules will be referred to as modules.

Newer PIC families of microcontrollers can have one or more instances of a given module type. All instances of a module have the same features and functions. There can be some modules that are a subset of other module types and inherit some of their features. The inherited features are fully compatible with the respective module type. For example, the subset module for a timer can have fewer compare and capture channels than a full timer module.

Figure 1-2. Module Types, Instances, Registers and Bits



A module type can be the Enhanced Universal Synchronous Asynchronous Receiver Transmitter (EUSART), while the module instance is, for example, 'EUSART1', where the '1' suffix indicates that the instance is 'EUSART number 1'. For simplicity, a module instance will be referred to as a module throughout this document, unless there is a need to differentiate.

Each module has several registers that contain control or status bits. All modules of a given type contain the same set or subset of registers. All of these registers contain the same set or subset of control and status bits.

All of the registers corresponding to a module have a fixed address in the I/O memory map. This way, each register will be available at an absolute address specified by the data sheet.

Every module has a dedicated chapter that presents the features of the module, a functional description of the module and the specific signals and guidelines on how to configure a certain mode of operation with all the terminology explained. At the end of a module chapter, the *Register Definitions* subchapter contains the scope of every register, the reset values of the registers, and whether or not it is readable or writable. It also provides the position of every configurable/accessible bit of a register.

All the registers, their addresses, and the bit names and positions are described in the *Register Summary* section for each module. The register summary for the ADC module is presented in [Figure 1-3](#).

Figure 1-3. Register Summary for ADC Peripheral

Address	Name	Bit Pos.								
0x00 ... 0x0F50	Reserved									
0x0F51	ADACT	7:0								ADACT[4:0]
0x0F52	ADCLK	7:0								ADCS[5:0]
0x0F53	ADREF	7:0				ADNREF				ADPREF[1:0]
0x0F54	ADCON1	7:0	ADPPOL	ADIPEN	ADGPOL					ADDSSEN
0x0F55	ADCON2	7:0	ADPSIS		ADCRS[2:0]		ADACL			ADMMD[2:0]
0x0F56	ADCON3	7:0			ADCALC[2:0]		ADSOI			ADTMD[2:0]
0x0F57	ADACQ	7:0								ADACQ[7:0]
0x0F58	ADCAP	7:0								ADCAP[4:0]
0x0F59	ADPRE	7:0								ADPRE[7:0]
0x0F5A	ADPCH	7:0								ADPCH[5:0]
0x0F5B	ADCON0	7:0	ADON	ADCONT		ADCS		ADFM		ADGO
0x0F5C	ADPREV	7:0								ADPREVL[7:0]
		15:8								ADPREVH[7:0]
0x0F5E	ADRES	7:0								ADRESL[7:0]
		15:8								ADRESH[7:0]
0x0F60	ADSTAT	7:0	ADAOV	ADUTHR	ADLTHR	ADMATH				ADSTAT[2:0]
0x0F61	ADRPT	7:0								ADRPT[7:0]
0x0F62	ADCNT	7:0								ADCNT[7:0]
0x0F63	ADSTPT	7:0								ADSTPTL[7:0]
		15:8								ADSTPTH[7:0]
0x0F65	ADLTH	7:0								ADLTHL[7:0]
		15:8								ADLTHH[7:0]
0x0F67	ADUTH	7:0								ADUTHL[7:0]
		15:8								ADUTHH[7:0]
0x0F69	ADERR	7:0								ADERRL[7:0]
		15:8								ADERRH[7:0]
0x0F6B	ADACC	7:0								ADACCL[7:0]
		15:8								ADACCH[7:0]
0x0F6D	ADFLTR	7:0								ADFLTRL[7:0]
		15:8								ADFLTRH[7:0]

For examples on how to access the ADGO bit from the ADCON0 register, refer to section 2.1.1. Register Unions

1.4 Naming Conventions

This section describes the register and bit naming conventions that can be found in the device data sheet.

1.4.1 Register Naming Conventions

Registers are divided into Control (CON), Status (STA) and Data registers with their naming reflecting this. A general purpose control register of a module has the control identifier named CON. If multiple general purpose control registers exist in a module, they have a suffix number identifier. In this case, the control registers will be named CON0, CON1, CON2 and so on. For example, see ADCON0, ADCON1, ADCON2 and ADCON3 registers in [Figure 1-3](#).

When there are multiple instances of the same peripheral in a device, the name of the peripheral control registers will be depicted as the concatenation of the peripheral identifier, the peripheral instance number and the control identifier. Therefore, all the register names of PIC microcontrollers are unique. For example, in [Figure 1-4](#), observe the RC2STA (Receive Status and Control Register) for EUSART peripheral instance 2.

For registers that have a specific function, their name reflects this functionality. For example, BAUD2CON is the Baud Rate Control Register for the second instance of the EUSART peripheral.

Figure 1-4. Register Summary for EUSART Peripheral

Address	Name	Bit Pos.								
0x00 ... 0x0E93	Reserved									
0x0E94	RC2REG	7:0	RCREG[7:0]							
0x0E95	TX2REG	7:0	TXREG[7:0]							
0x0E96	SP2BRG	7:0	SPBRGL[7:0]							
		15:8	SPBRGH[7:0]							
0x0E98	RC2STA	7:0	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
0x0E99	TX2STA	7:0	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D
0x0E9A	BAUD2CON	7:0	ABDOVF	RCIDL		SCKP	BRG16		WUE	ABDEN
0x0E9B ... 0x0F97	Reserved									
0x0F98	RC1REG	7:0	RCREG[7:0]							
0x0F99	TX1REG	7:0	TXREG[7:0]							
0x0F9A	SP1BRG	7:0	SPBRGL[7:0]							
		15:8	SPBRGH[7:0]							
0x0F9C	RC1STA	7:0	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
0x0F9D	TX1STA	7:0	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D
0x0F9E	BAUD1CON	7:0	ABDOVF	RCIDL		SCKP	BRG16		WUE	ABDEN

Since the PIC data bus width is 8 bits, larger registers are implemented using several 8-bit registers. For a 16-bit register, the high and low bytes are accessed by appending 'H' and 'L' respectively to the register name. For example, the ADC Result Register is named ADRES and the two bytes are ADRESL and ADRESH.

After the *Register Summary* section in the device data sheet, each register has a *Register Definition* section, which fully describes the functionality of each bit and bit field in the register. The *Register Definitions* section shows one instance of all the register names with an 'x' in place of the peripheral instance number. An example is presented in [Figure 1-5](#).

1.4.2 Bit Naming Conventions

Register bits in the data sheet are named by combining a bit function abbreviation with the peripheral abbreviation. This format, also called a **Long Bit Name**, is used in both the register summary and register definition sections of the data sheet. For example, SPEN is the name for the Serial Port Enable bit, as shown in [Figure 1-5](#).

Figure 1-5. RCxSTA – Receive Status and Control Register

Name: RCxSTA
Address: 0xF9C,0xE98

Receive Status and Control Register

Bit	7	6	5	4	3	2	1	0
	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
Access	R/W	R/W	R/W	R/W	R/W	RO	R/HC	R/HC
Reset	0	0	0	0	0	0	0	0

Bit 7 – SPEN Serial Port Enable bit

Value	Description
1	Serial port enabled
0	Serial port disabled (held in Reset)

Since the prefix for the peripheral module type is unique, each bit name described in the data sheet is unique.

The device header file offers some register bits a **Short Bit Name** alternative consisting of only the bit function abbreviation. Since this is defined in the context of a bit union for a specific register, the bit access remains unique. For further details on how to access a bit using the short or long naming convention, refer to [2.1.1 Register Unions](#).

1.4.3 Register and Bit Naming Exceptions

1.4.3.1 Status, Interrupt, and Mirror Bits

Status, Interrupt Enable, Interrupt Flag, and Mirror bits are contained in registers that span across more than one peripheral. In these cases, the bit name shown is unique, and there is no prefix or short name variant.

1.4.3.2 Legacy Peripherals

There are some peripherals that do not strictly adhere to this naming convention. These are the peripherals that have existed for many years and are present in almost every device. These exceptions were necessary to limit the adverse impact of the new conventions on legacy code. Peripherals that do adhere to the new convention will include a table in the register section indicating the long name prefix for each peripheral instance. Peripherals that fall into the exception category will not have this table. These peripherals include, but are not limited to the following:

- Enhanced Universal Asynchronous Receiver Transceiver (EUSART)
- Master Synchronous Serial Port (MSSP)

1.5 PIC® Configuration Bits

Configuration bits are a collection of specialized bits that can only be modified at program time. Configuration bits are read during reset and enable or disable hardware features in the microcontroller. The features controlled by the configuration bits include, but are not limited to, the clock source, the Watchdog Timer (WDT), the Brown-Out Detector (BOD), and the Memory Read protection. Configuration bits are not executable code. Essentially, they are fuses located in the program memory space.

Each PIC microcontroller has its own set of configuration bits. The *Device Configuration* section of the individual data sheets contains the definition for each of the bits. See below an example from the [PIC16F18446 data sheet](#).

Figure 1-6. Register Summary for Configuration Words

Offset	Name	Bit Pos.							
0x8007	CONFIG1	7:0			RSTOSC[2:0]				FEXTOSC[2:0]
		13:8			FCMEN		CSWEN		CLKOUTEN
0x8008	CONFIG2	7:0		BOREN	LPBOREN			PWRTS[1:0]	MCLRE
		13:8			DEBUG	STVREN	PPS1WAY	ZCDDIS	BORV
0x8009	CONFIG3	7:0			WDTE[1:0]				WDTCP[4:0]
		13:8				WDTCCS[2:0]			WDTCWS[2:0]
0x800A	CONFIG4	7:0	WRTAPP			SAFEN	BBEN		BBSIZE[2:0]
		13:8			LVP		WRTSAF	WRTD	WRTC
0x800B	CONFIG5	7:0							CP
		13:8							

For further details on how to set the Configuration Bits consult the [3.4 Setting Configuration Bits](#) section.

2. Modules Representation in Header Files

A dedicated header file is available for each PIC device. If the target device is specified in the project settings, the MPLAB® XC8 Compiler will automatically include the correct header file if the device file is included as shown below:

```
#include <xc.h>
```

All of the required register macro definitions can be found in the header file along with bit masks, bit field masks, bit positions and union definitions for the registers. The macros and struct definitions which are already defined in the device specific header file can be used instead of using a register's address.

This is useful for devices that contain the same module and where the header file definitions for that module are identical.

2.1 Registers Representation in Header Files

The I/O memory map is laid out so that all registers for a given peripheral module are placed in one continuous memory block. Registers belonging to different modules are not mixed up, where the registers macros are defined as below:

```
#define LATA LATA
extern volatile unsigned char          LATA          __at(0xF82);
#define LATB LATB
extern volatile unsigned char          LATB          __at(0xF83);
#define LATC LATC
extern volatile unsigned char          LATC          __at(0xF84);
```

2.1.1 Register Unions

Each register has a union declared in the header file for the individual bits in that register. This allows access to an individual bit/bit field from the register using the union declaration.

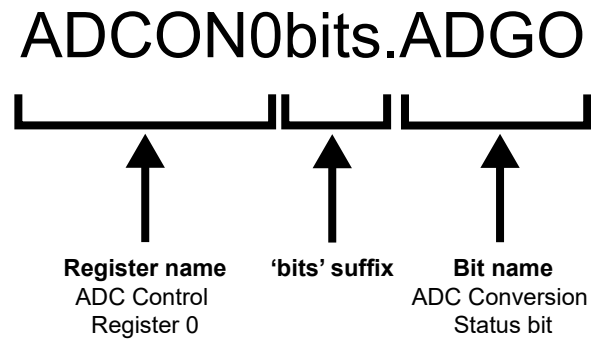
```
typedef union {
    struct {
        unsigned ADGO          :1;
        unsigned ADFM          :1;
        unsigned ADCS          :1;
        unsigned ADCONT        :1;
        unsigned ADON          :1;
    };
    struct {
        unsigned GO            :1;
        unsigned ADFM0        :1;
    };
} ADCON0bits_t;
extern volatile ADCON0bits_t ADCON0bits __at(0xF5B);
```

The union declaration of the ADCON0 register is shown in the code listing above. This register can be accessed as a whole using the macro declaration or as an individual bit/bit field from the register using the union declaration. Here is an example:

```
ADCON0 = 0x01;      /* using macro declaration */
ADCON0bits.GO = 1; /* using bit unions with short bit name convention */
ADCON0bits.ADGO = 1; /* using bit unions with long bit name convention */
```

The convention used when accessing a bit or a bit field from a register using the union declaration of register is presented in [Figure 2-1](#).

Figure 2-1. Access Register Unions Convention



For further details on unions, consult [Microchip Developer - Unions](#).

2.1.2 Multibyte Registers

Some registers are used in conjunction with other registers to represent 16-bit values. These registers can be accessed as a whole using the register macro or by accessing the low/high bytes using the 'L'/'H' suffixes attached to the register macro. For example, the 16-bit ADC Result register has the following declaration in the header file:

```
#define ADRES ADRES
extern volatile unsigned short      ADRES          __at(0xF5E);
#define ADRESL ADRESL
extern volatile unsigned char       ADRESL         __at(0xF5E);
#define ADRESH ADRESH
extern volatile unsigned char       ADRESH         __at(0xF5F);
```

2.2 Bit Masks and Bit Field Masks

Register bits can be manipulated using predefined masks, or bit positions. The predefined bit masks from the header file are either related to individual bits, called a bit mask, or related to a bit field, called a bit field mask.

A bit mask is used both when setting and clearing individual bits. A bit field mask is mainly used when clearing multiple bits in a bit field.

For the ADCON2 register, the bit fields, bit names, bit positions, and bit masks of this register, see the table below.

Table 2-1. Bit Fields, Bit Names, Bit Positions, and Bit Masks in the ADCON2 Register

Bit Field	-	ADCRS			-	ADMD		
Bit Name	ADPSIS	ADCRS2	ADCRS1	ADCRS0	ADACL	ADMD2	ADMD1	ADMD0
Bit Position	7	6	5	4	3	2	1	0

.....continued								
Bit Field	-	ADCRS			-	ADMD		
Bit Mask	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

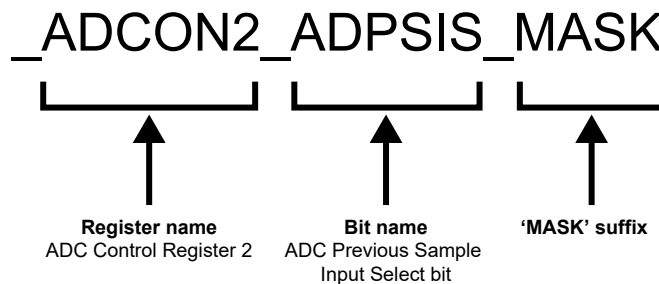
2.2.1 Bit Masks

The bit masks are predefined in the device header file. For example, the bit mask for the ADPSIS bit is defined as below.

```
#define _ADCON2_ADPSIS_MASK 0x80
```

The naming convention adopted for the predefined bit masks in the header file is presented in [Figure 2-2](#) with an example for the ADPSIS bit in the ADCON2 register.

Figure 2-2. Naming Convention of Bit Masks



Note: The register name, bit name, bit field name, the 'MASK' suffix are separated with '_' and the entire macro name begins with '_'.

2.2.2 Bit Field Masks

Many functions are controlled by a bit field. For example, ADCRS[2:0] and ADMD[2:0] in the ADCON2 register are grouped bits. The value of the bits in a field selects a specific configuration.

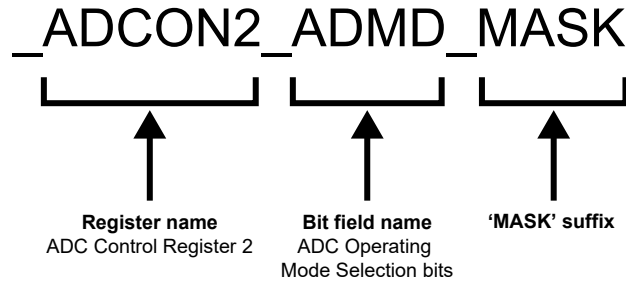
When changing bits in a bit field, it is not enough to set the bits for the desired configuration. It is also required to clear the bits from the old configuration before assigning a new value. To facilitate this, a bit field mask is defined.

The field masks are predefined in the device header file. For example, the field mask for the ADMD bit field is defined below.

```
#define _ADCON2_ADMD_MASK 0x7
```

The naming convention adopted for the predefined bit field masks in the header file is presented in the [Figure 2-3](#) with an example for the ADMD bit field in the ADCON2 register.

Figure 2-3. Naming Convention of Bit Field Masks



The bits from a bit field can be accessed as individual bits. To differentiate between these bits, a suffix (index of each bit in the bit field) is appended to the bit field name. The masks for the bits in a bit field are defined below.

```
#define _ADCON2_ADMD0_MASK          0x1
#define _ADCON2_ADMD1_MASK          0x2
#define _ADCON2_ADMD2_MASK          0x4
```

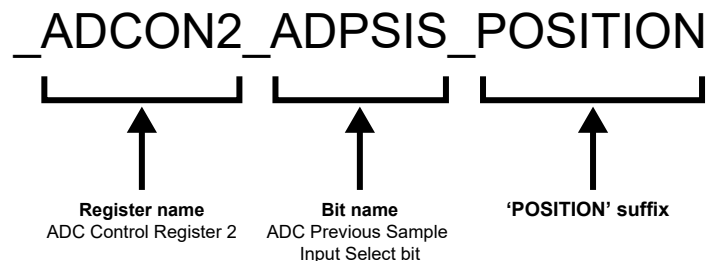
For further details on bit fields, consult [Microchip Developer - Bit Fields](#).

2.3 Bit Positions

It is possible to use bit positions as an alternative to set or clear bits. A bit position within a register is defined using the same naming convention used at the bit masks, with the '_POSITION' suffix instead of '_MASK'. There is also another definition for the bit positions in the header file. This bit position definition has the same functionality, but the suffix is '_POSN'. This is implemented for compatibility reasons.

The naming convention adopted for the predefined bit positions in the header file is presented in the [Figure 2-4](#) with an example for the ADPSIS bit position in the ADCON2 register.

Figure 2-4. Naming Convention of Bit Positions



The bit position definition for the ADPSIS bit from the header file is shown below.

```
#define _ADCON2_ADPSIS_POSN          0x7
#define _ADCON2_ADPSIS_POSITION      0x7
```

The bit positions are included for compatibility reasons. They are also needed when programming in assembly for instructions that use a bit number.

3. Writing Bare Metal C-Code for PIC

The following section focuses on how to write C-code for the PIC16 and PIC18 microcontroller families. The examples describe how to make the code highly readable and portable between different PIC16 and PIC18 devices. The examples can also be used as a guideline on how to write code that is easy to verify and maintain.

PIC registers are located in dedicated and continuous blocks in the memory space and can be seen as encapsulated units. This reflects on the way that the registers are accessed when coding in C. Registers are encapsulated using C unions, in which all the bits and bit fields are encapsulated using C structs. A register can be accessed as a whole using the macro declaration of the register or it can be accessed using the union declaration of the register. A bit field can be accessed similarly to the register.

This document introduces a naming convention and register access method that is compliant with the PIC header files. This provides readability and portability to the codes written in C-code.

3.1 Set, Clear and Read Register Bits

Setting and clearing register bits are fundamental operations used in embedded programming. Applications are based on this technique.

The Read-Modify-Write (RMW) operations are a class of atomic operations that both read a memory location and write a new value to it simultaneously, either with a completely new value or some part of the previous value.

Since it has a wide applicability, reading the value of a bit is mainly used in conditional expressions (e.g. `if` statement) and as a condition in loop expressions (e.g. `while` statement). A common use case of this technique is polling on an interrupt flag (reading the value of the bit and executing a set of instructions if the bit is set/clear).

Note: For further details on binary arithmetic see [Bitwise Operators](#).

3.1.1 Set, Clear and Read Register Bits using Bit Unions

The recommended coding style to set or clear a specific bit in a register is to use the union declaration of the register from the header file.

The example below shows how to set and clear the Enable bit from the ADCON0 register using the recommended coding style.

```
ADCON0bits.ADON = 1; /* the ADC Enable bit is set */
ADCON0bits.ADON = 0; /* the ADC Enable bit is cleared */
```

The value of a register bit can be tested as follows. The code listing below shows how to poll the ADGO bit, waiting until it is cleared:

```
/* wait while the ADGO bit is set */
while (ADCON0bits.ADGO)
{
    /* wait */
}
```

Note: Setting the ADGO bit in the ADC's ADCON0 register starts an ADC conversion. This bit is then cleared by hardware when the conversion is complete.

The code listing below shows how to read the value of a PORT pin using bit unions and execute a set of instructions if that pin is low.

```
/* if pin 0 of the PORTA is clear execute a set of instructions */
if (!PORTAbits.RA0)
{
```

```

    /* set of instructions */
}

```

3.1.2 Set, Clear and Read Register Bits using Bit Masks

There are alternative ways to set and clear register bits by using bit masks or bit positions.

In order to set a bit from a register using bit masks, the binary OR operator will be applied between the register and the bit mask. Using the binary OR operator will ensure that the other bit settings made inside the register will remain the same and unaffected by this operation.

```

ADCON0 |= _ADCON0_ADON_MASK; /* the ADC Enable bit is set */

```

In order to clear a bit from a register using bit masks, the binary AND operator will be applied between the register and the negated value of the bit mask. This operation also keeps the other bit settings unchanged.

```

ADCON0 &= ~_ADCON0_ADON_MASK; /* the ADC Enable bit is cleared */

```

The bit mask for the ADC Enable bit (ADON) has the following declaration in the header file.

```

#define _ADCON0_ADON_MASK                0x80

```

The code listing below shows how to read the value of a PORT pin using bit masks and execute a set of instructions if that pin is low.

```

if(PORTA & _PORTA_RA0_MASK)
{
    /* set of instructions */
}

```

3.1.3 Set, Clear and Read Register Bits using Bit Positions

In order to set a bit from a register using bit positions, the binary OR operator will be applied between the register and the value resulting from shifting '1' with the value of the bit position. To clear a bit using bit positions the binary AND operator is used with the negated value of the shifting result.

```

ADCON0 |= (1 << _ADCON0_ADON_POSITION); /* the ADC Enable bit is set */
ADCON0 &= ~(1 << _ADCON0_ADON_POSITION); /* the ADC Enable bit is cleared */

```

Note: The bit position for the ADC Enable bit (ADON) has the following declaration in the header file.

```

#define _ADCON0_ADON_POSITION            0x7

```

The code listing below shows how to read the value of a PORT pin using bit positions and execute a set of instructions if that pin is low.

```

if(PORTA & (1<< _PORTA_RA0_POSITION))
{
    /* set of instructions */
}

```

3.2 Register Initialization

In order to initialize a register, the user must find the desired configuration of the register to achieve the expected functionality by consulting the device data sheet and setting or clearing register bits, so that the value in the register matches the desired configuration.

Register initialization is most often performed as part of device initialization after reset, when the register is in a known state of '0'. This is a special use-case, since:

- The register value may be 0x00
- Various bits and bit fields need to be configured at once

Read-modify-write operations are not needed, when working with bit masks or bit positions, if the reset value of the register is 0x00 and the register configures in a single line.

Note: For most PIC registers, the reset value for all bits and bit fields is '0', but there are exceptions. For example, the Peripheral Interrupt Priority Register 3 has several bits with the reset value '1'. In this case, the user has to explicitly set the desired configuration without relying on the fact that usually bits reset values are 0. The reset value for all bits and bit fields in a register are shown in the figure below.

Figure 3-1. Peripheral Interrupt Priority Register 3 – Reset Value

Name: IPR3
Address: 0xEB8

Peripheral Interrupt Priority Register 3

Bit	7	6	5	4	3	2	1	0
	RC2IP	TX2IP	RC1IP	TX1IP	BCL2IP	SSP2IP	BCL1IP	SSP1IP
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	1	1	0	0	1	1

The following example will apply the various methods of configuring a register (T0CON0 – Timer 0 Control Register 0), shown in the figure below.

Note: For the register in this example (T0CON0), all bits and bit fields are '0' after reset.

Figure 3-2. Timer 0 Control Register 0 – Configuration

Name: T0CON0
Address: 0xFD4

Timer0 Control Register 0

Bit	7	6	5	4	3	2	1	0
	T0EN		T0OUT	T016BIT	T0OUTPS[3:0]			
Access	R/W		R	R/W	R/W	R/W	R/W	R/W
Reset	0		0	0	0	0	0	0

Bit 7 – T0EN TMR0 Enable

Value	Description
1	The module is enabled and operating
0	The module is disabled

Bit 5 – T0OUT TMR0 Output

Bit 4 – T016BIT TMR0 Operating as 16-Bit Timer Select

Value	Description
1	TMR0 is a 16-bit timer
0	TMR0 is an 8-bit timer

Bits 3:0 – T0OUTPS[3:0] TMR0 Output Postscaler (Divider) Select

Value	Description
1111	1:16 Postscaler
1110	1:15 Postscaler
1101	1:14 Postscaler
1100	1:13 Postscaler
1011	1:12 Postscaler
1010	1:11 Postscaler
1001	1:10 Postscaler
1000	1:9 Postscaler
0111	1:8 Postscaler
0110	1:7 Postscaler
0101	1:6 Postscaler
0100	1:5 Postscaler
0011	1:4 Postscaler
0010	1:3 Postscaler
0001	1:2 Postscaler
0000	1:1 Postscaler

Here is the desired configuration:

- Enable the timer - T0EN = 1
- Select 8-bit mode - T016BIT = 0 (default)
- Select a 1:10 postscaler - T0OUTPS = 1001 (changed in [3.3 Change Register Bit Field Configuration](#) to 0111)

The resulting value can be directly written to the register:

```
T0CON0 = 0b1000 1001;    /* binary */
T0CON0 = 0x89;           /* hexadecimal */
T0CON0 = 137;            /* decimal */
```

However, to improve the readability (and potentially the portability) of the code, it is recommended to use the device defines, which are shown in the upcoming sections.

3.2.1 Register Initialization using Bit Unions

Register initialization using bit and bit field unions will always need to be done in several lines of code, if configuring more than one bit or bit field.

The example below shows the recommended way of initializing a register, by using the union declaration of the register from the header file.

```
TOCON0bits.TOEN = 1;      /* Enable TMR0 */
TOCON0bits.T016BIT = 0;  /* Select 8-bit operation mode */
TOCON0bits.T0OUTPS = 0x9; /* Select 1:10 postscaler */
```

3.2.2 Register Initialization using Bit Masks

Read-modify-write operations are not needed, when working with bit masks or bit positions, if the reset value of the register is 0x00 and the register configures in a single line.

The example below shows how to achieve the same configuration using bit masks.

```
TOCON0 = _TOCON0_TOEN_MASK      /* Enable TMR0 */
| _TOCON0_T0OUTPS0_MASK        /* Select 1:10 postscaler */
| _TOCON0_T0OUTPS3_MASK;      /* 8-bit operation mode selected by default */
```

Note: The bit wise OR ('|') operator on the register side of the assignment is left out. In most cases, device and peripheral initialization routines are written in this way.



The above initialization of the register must be done in a single line of C code. Writing as follows, the bit mask used in the second and third line would clear the bits in the previous lines.

```
/* incorrect initialization of the TOCON0 register */
TOCON0 = _TOCON0_TOEN_MASK;
TOCON0 = _TOCON0_T0OUTPS0_MASK;
TOCON0 = _TOCON0_T0OUTPS3_MASK;
```

Note: Bit Masks can only set bits in a single line of code, so configurations which require bits to be cleared are left out since they are correctly configured by their reset value.

In this example, no mask is used to explicitly configure the timer in the 8-bit mode. This is possible because the reset value of the T016BIT is '0' which corresponds to the 8-bit mode. To emphasize the configuration of this bit as 0, the user could explicitly select the desired 8-bit mode by using a read-modify-write operation. However, this would need to be a separate line of code and would leave the register unchanged:

```
TOCON0 = _TOCON0_TOEN_MASK      /* Enable TMR0 */
| _TOCON0_T0OUTPS0_MASK        /* Select 1:10 postscaler */
| _TOCON0_T0OUTPS3_MASK;
TOCON0 &= ~_TOCON0_T016BIT_MASK; /* Select 8-bit operation mode explicitly */
```

3.2.3 Register Initialization using Bit Positions

Read-modify-write operations are not needed, when working with bit masks or bit positions, if the reset value of the register is '0' and the register configures in a single line.

The code listing below shows how to initialize a register using bit positions.

```
TOCON0 = (1 << _TOCON0_TOEN_POSITION) /* Enable TMR0 */
| (0 << _TOCON0_T016BIT_POSITION) /* A placeholder to select 16-bit mode*/
```

```
| (1 << _T0CON0_T0OUTPS0_POSITION)
| (1 << _T0CON0_T0OUTPS3_POSITION); /* Select 1:10 postscaler */
```

Note: The `(0 << _T0CON0_T016BIT_POSITION)` line does nothing here, but it can be used as a placeholder to quickly change bit settings.

3.3 Change Register Bit Field Configurations

This section covers considerations when updating a register bit field using various header file defines. The following bit field will be used as an example, where an update is needed, compared to the initialization configuration covered in [3.2. Register Initialization](#) section.

T0OUTPS[3:0] TMR0 Output Postscaler Select:

- Select a 1:10 postscaler - T0OUTPS: 1001 (previous setting)
- Select a 1:8 postscaler - T0OUTPS: 0111 (new setting)

3.3.1 Change Register Bit Field Configurations using Bit Unions

The union declaration of the registers offers access to register bits and bit fields without affecting the rest of the register.

```
/* using a hex value */
T0CON0bits.T0OUTPS = 0x7; /* Select 1:10 postscaler */
/* using a binary value */
T0CON0bits.T0OUTPS = 0b0111; /* Select 1:10 postscaler */
```

This is the recommended way of updating bit field register configurations, which is simpler than the alternative options.

3.3.2 Change Register Bit Field Configurations using Bit Masks

When updating only a bit field in a register, a read-modify-write must be used, to keep the other settings unaffected. Therefore, in order to change the configuration of a register bit field, it is recommended to first clear the bit field and then set a new configuration. However, in order to avoid putting the register in an unintended state between the clear and setting the new configuration, this should be done in a single line of code. For simplicity, the steps are first covered individually.

The bit field masks can be used to clear a bit field before assigning a new configuration. In the example, the T0OUTPS bit field mask is used to clear bit field.

```
/* The T0OUTPS bit field is cleared (Selecting a postscaler of 1:1 (T0OUTPS = 0)) */
T0CON0 &= ~_T0CON0_T0OUTPS_MASK;
/* Selecting new configuration (0b0111) of the T0OUTPS bit field */
T0CON0 |= _T0CON0_T0OUTPS2_MASK | _T0CON0_T0OUTPS1_MASK | _T0CON0_T0OUTPS0_MASK;
```

The bit field mask for the TMR0 Output Postscaler Select (T0OUTPS) has the following declaration in the header file.

```
#define _T0CON0_T0OUTPS_MASK 0xF
```

These steps must be implemented in a single line to avoid putting the microcontroller in an unintended state.

```
T0CON0 = (T0CON0 & ~_T0CON0_T0OUTPS_MASK) | _T0CON0_T0OUTPS2_MASK
| _T0CON0_T0OUTPS1_MASK
| _T0CON0_T0OUTPS0_MASK;
```

Note: If the code is implemented as two separate lines, the first line of code will select for a short time, a postscaler of 1:1 (T0OUTPS = 0).



Even though it may seem easier to split the code into two separate lines of code, one to clear the old configuration and another to set the desired configuration. It is recommended to use a single line to achieve this, as presented in the code listing.

3.3.3 Change Register Bit Field Configurations using Bit Positions

The example below shows how to update a register bit field using bit positions to set the new configuration. Similar to the process of updating a register configuration using bit masks, the current configuration must be cleared and the new configuration set, in a single line of code.

```
/* Changing a bit field configuration with bit positions */
T0CON0 = (T0CON0 & ~_T0CON0_T0OUTPS_MASK) | (0 << _T0CON0_T0OUTPS3_POSITION)
| (1 << _T0CON0_T0OUTPS2_POSITION)
| (1 << _T0CON0_T0OUTPS1_POSITION)
| (1 << _T0CON0_T0OUTPS0_POSITION);
```

Note: The (0 << _T0CON0_T0OUTPS3_POSITION) line is added simply for readability, but it can be removed.

```
T0CON0 = (T0CON0 & ~_T0CON0_T0OUTPS_MASK) | (1 << _T0CON0_T0OUTPS2_POSITION)
| (1 << _T0CON0_T0OUTPS1_POSITION)
| (1 << _T0CON0_T0OUTPS0_POSITION);
```

3.4 Setting Configuration Bits

It is unlikely that a new C program will run properly on the device, even though the program is valid. All Microchip 8-bit devices must be configured to ensure correct operation. Some configuration settings affect fundamental operation of the device, such as those for the instruction clock.

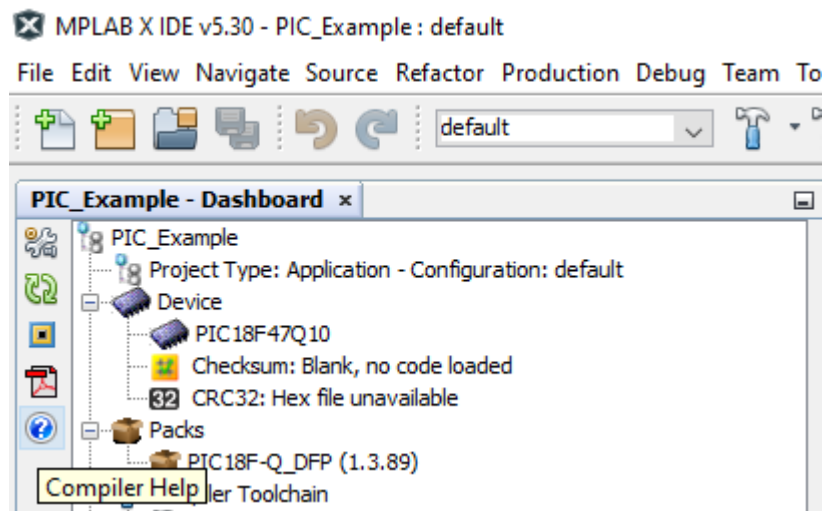


- Not setting the Configuration Bits can prevent even blinking an LED. Even if the TRIS register is set up and a value is written to the port, there are several things that can prevent such seemingly simple program from working.
- Ensure that the device's Configuration registers are set up correctly. Also, the user must make sure that every bit in these registers is explicitly specified, not leaving them in their default state. All the configuration features are described in the device data sheet. If the Configuration Bits that specify the oscillator source are wrong, for example, the device clock cannot be running.
- For more information, refer to the [MPLAB XC8 C Compiler User's Guide](#).

3.4.1 Accessing XC8 Configuration Bits Examples

To configure the device using MPLAB X Integrated Development Environment (IDE), the user must use configuration pragmas. More information about the compiler and the configuration bits of the desired device can be found by accessing the Compiler Help, the blue question mark from the MPLAB X IDE project dashboard, as presented in the figure below.

Figure 3-3. Accessing Compiler Help



Example configurations can be found for specific devices under the *Configuration Settings Reference* section. An example for the PIC16F18446 is shown below.

Figure 3-4. PIC16F18446 Example Configuration

For example:

```
// External Oscillator mode selection bits : EC above 8MHz; PFM set to high power
// Power-up default value for COSC bits : EXTOSC operating per FEXTOSC bits
// Clock Out Enable bit : CLKOUT function is disabled; i/o or oscillator function on OSC2
// Clock Switch Enable bit : Writing to NOSC and NDIV is allowed
// Fail-Safe Clock Monitor Enable bit : FSCM timer enabled
#pragma config FEXTOSC = ECH, RSTOSC = EXT1X, CLKOUTEN = OFF, CSWEN = ON, FCMEN = ON
```

3.4.2 MPLAB® X IDE Support for Setting Configuration Bits

The easiest way to complete the pragmas that are required to configure the device is to use the Configuration Bits Window, a feature of MPLAB X IDE.

Follow these steps to get the information to complete the pragmas:

- Open the Configuration Bits Window (*Window > Target Memory Views > Configuration Bits* or *Production > Set Configuration Bits*).
- Review every setting in the Configuration Bits Window.
- Generate the pragma derivatives that implement the chosen settings by clicking the **Generate Source Code to Output** button.
- Copy the generated code from this window to a source file.

For more details, see the following references:

- Consult the [MPLAB XC8 Getting Started Guide](#), Specifying *Device Configuration Bits* section.
- Microchip Developer Help: [View and Set Configuration Bits](#).
- Consult the video [MPLAB X IDE Advanced Debugging - Event Breakpoints](#).

4. Application Example Showing Alternative Ways of Writing Code

The example below demonstrates how to configure the microcontroller to turn on an LED when a user button is pressed. To achieve this, the user needs to identify the pins of the microcontroller routed to the user LED and to the user button. This example is developed for the [PIC18F47Q10 Curiosity Nano development board](#). The user LED is routed to the pin 0 of the PORTE. The user button is routed to the pin 2 of the PORTE.

4.1 Turn ON an LED on a Button Press using Bit Unions

The code below provides the described functionality following the recommended coding style by using the union declaration of the registers from the data sheet.

```
#include <xc.h>

void main(void)
{
    /* setting pin RE0 as output (LED) */
    TRISEbits.TRISE0 = 0;
    /* setting pin RE2 as input (button) */
    TRISEbits.TRISE2 = 1;
    /* enable digital input buffer for pin RE2 (button) */
    ANSELEbits.ANSELE2 = 0;
    /* enable internal pull-up for pin RE2 (button) */
    WPUEbits.WPUE2 = 1;

    /* main program loop */
    while(1)
    {
        /* if button is pressed (pin RE2 high) */
        if(PORTEbits.RE2)
        {
            /* turn on the LED (pin RE0 high) */
            LATEbits.LATE0 = 1;
        }
        else
        {
            /* turn off the LED (pin RE0 low) */
            LATEbits.LATE0 = 0;
        }
    }
}
```

4.2 Turn ON an LED on a Button Press using Bit Masks

The code below provides the same functionality using bit masks.

```
#include <xc.h>

void main(void)
{
    /* setting pin RE0 as output (LED) */
    TRISE &= ~_TRISE_TRISE0_MASK;
    /* setting pin RE2 as input (button) */
    TRISE |= _TRISE_TRISE2_MASK;
    /* enable digital input buffer for pin RE2 (button) */
    ANSELE &= ~_ANSELE_ANSELE2_MASK;
    /* enable internal pull-up for pin RE2 (button) */
    WPUE |= _WPUE_WPUE2_MASK;

    /* main program loop */
    while(1)
    {
        /* if button is pressed (pin RE2 high) */
        if(PORTE & _PORTE_RE2_MASK)
```

```

    {
        /* turn on the LED (pin RE0 high) */
        LATE |= _LATE_LATE0_MASK;
    }
    else
    {
        /* turn off the LED (pin RE0 low) */
        LATE &= ~_LATE_LATE0_MASK;
    }
}
}
}

```

4.3 Turn ON an LED on a Button Press using Bit Positions

The code below provides the same functionality using bit positions.

```

#include <xc.h>

void main(void)
{
    /* setting pin RE0 as output (LED) */
    TRISE &= ~(1 << _TRISE_TRISE0_POSITION);
    /* setting pin RE2 as input (button) */
    TRISE |= (1 << _TRISE_TRISE2_POSITION);
    /* enable digital input buffer for pin RE2 (button) */
    ANSELE &= ~(1 << _ANSELE_ANSELE2_POSITION);
    /* enable internal pull-up for pin RE2 (button) */
    WPUE |= (1 << _WPUE_WPUE2_POSITION);

    /* main program loop */
    while(1)
    {
        /* if button is pressed (pin RE2 high) */
        if(PORTE & (1 << _PORTE_RE2_POSITION))
        {
            /* turn on the LED (pin RE0 high) */
            LATE |= (1 << _LATE_LATE0_POSITION);
        }
        else
        {
            /* turn off the LED (pin RE0 low) */
            LATE &= ~(1 << _LATE_LATE0_POSITION);
        }
    }
}
}

```

5. Further Steps

The purpose of this section is to direct the user to the IDE installation guides and tutorials, the available application notes and the technical briefs.

5.1 MPLAB® X and XC8 Compiler

The XC compilers are comprehensive solutions for the project's software development. The MPLAB XC8 compiler supports all 8-bit PIC microcontrollers and is available as a free and unrestricted use download. When combined with the MPLAB X IDE, the front end provides editing errors and breakpoints that match corresponding lines in the source code. Single stepping through C source code inspects variables and structures at critical points.

Step-by-step instructions on how to set up a bare metal project for PIC is found in the [MPLAB XC8 Getting Started Guide](#). For further details on the XC8 compiler, consult [MPLAB XC8 C Compiler User's Guide](#).

An introduction in the MPLAB X environment and an installation guide is found in the [Getting Started - MPLAB X IDE Essentials - 01: Installation and Ecosystem](#). More information on Microchip's MPLAB X IDE is found at [MPLAB X IDE User's Guide](#).

5.2 Application Notes and Technical Briefs

Additionally, there are many application notes and technical briefs available online that describe module functioning or introduce important features of a module. For example, the [5-Bit Digital-to-Analog Converter Technical Brief](#) describes the DAC peripheral as relevant for the PIC16F and PIC18F microcontroller families.

Another example is the [PIC16/PIC18 ADC2 Technical Brief](#). Other application notes and technical briefs can be found at [Browse Application Notes - Microchip](#).

6. Conclusion

The main purpose of this technical brief is to introduce the user to a preferred coding style for programming the PIC microcontrollers. After reviewing this document, users will understand the type of information the data sheet is providing, macro definitions, variable declarations and data type definitions provided by the header files. The goals are to use an easily maintainable, portable and readable coding style; to become familiar with the naming conventions for the PIC registers and bits; and to prepare for further steps in developing a project using these microcontrollers.

This document provides information on specific data sheets, naming conventions, guidance on how to write C-code for PIC microcontrollers and further steps in developing a project.

While the C-code writing methods suggested here are not mandatory, one can consider the numerous advantages. The larger the project and the more features the device has, the greater the benefit of C-code utilization.

7. References

1. [PIC18F27/47Q10 Data Sheet](#)
2. [MPLAB XC8 Getting Started Guide](#)
3. [MPLAB XC8 C Compiler User's Guide](#)
4. [Microchip Developer - Fundamentals of the C Programming Language](#)

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip design partner program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Microchip Devices Code Protection Feature

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

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

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

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with

your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TempTrackr, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, Vite, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, INICnet, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, Serial Quad I/O, SMART-I.S., SQI, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2020, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-5224-6166-1

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
<p>Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Tel: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com</p> <p>Atlanta Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455</p> <p>Austin, TX Tel: 512-257-3370</p> <p>Boston Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088</p> <p>Chicago Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075</p> <p>Dallas Addison, TX Tel: 972-818-7423 Fax: 972-818-2924</p> <p>Detroit Novi, MI Tel: 248-848-4000</p> <p>Houston, TX Tel: 281-894-5983</p> <p>Indianapolis Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380</p> <p>Los Angeles Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800</p> <p>Raleigh, NC Tel: 919-844-7510</p> <p>New York, NY Tel: 631-435-6000</p> <p>San Jose, CA Tel: 408-735-9110 Tel: 408-436-4270</p> <p>Canada - Toronto Tel: 905-695-1980 Fax: 905-695-2078</p>	<p>Australia - Sydney Tel: 61-2-9868-6733</p> <p>China - Beijing Tel: 86-10-8569-7000</p> <p>China - Chengdu Tel: 86-28-8665-5511</p> <p>China - Chongqing Tel: 86-23-8980-9588</p> <p>China - Dongguan Tel: 86-769-8702-9880</p> <p>China - Guangzhou Tel: 86-20-8755-8029</p> <p>China - Hangzhou Tel: 86-571-8792-8115</p> <p>China - Hong Kong SAR Tel: 852-2943-5100</p> <p>China - Nanjing Tel: 86-25-8473-2460</p> <p>China - Qingdao Tel: 86-532-8502-7355</p> <p>China - Shanghai Tel: 86-21-3326-8000</p> <p>China - Shenyang Tel: 86-24-2334-2829</p> <p>China - Shenzhen Tel: 86-755-8864-2200</p> <p>China - Suzhou Tel: 86-186-6233-1526</p> <p>China - Wuhan Tel: 86-27-5980-5300</p> <p>China - Xian Tel: 86-29-8833-7252</p> <p>China - Xiamen Tel: 86-592-2388138</p> <p>China - Zhuhai Tel: 86-756-3210040</p>	<p>India - Bangalore Tel: 91-80-3090-4444</p> <p>India - New Delhi Tel: 91-11-4160-8631</p> <p>India - Pune Tel: 91-20-4121-0141</p> <p>Japan - Osaka Tel: 81-6-6152-7160</p> <p>Japan - Tokyo Tel: 81-3-6880-3770</p> <p>Korea - Daegu Tel: 82-53-744-4301</p> <p>Korea - Seoul Tel: 82-2-554-7200</p> <p>Malaysia - Kuala Lumpur Tel: 60-3-7651-7906</p> <p>Malaysia - Penang Tel: 60-4-227-8870</p> <p>Philippines - Manila Tel: 63-2-634-9065</p> <p>Singapore Tel: 65-6334-8870</p> <p>Taiwan - Hsin Chu Tel: 886-3-577-8366</p> <p>Taiwan - Kaohsiung Tel: 886-7-213-7830</p> <p>Taiwan - Taipei Tel: 886-2-2508-8600</p> <p>Thailand - Bangkok Tel: 66-2-694-1351</p> <p>Vietnam - Ho Chi Minh Tel: 84-28-5448-2100</p>	<p>Austria - Wels Tel: 43-7242-2244-39 Fax: 43-7242-2244-393</p> <p>Denmark - Copenhagen Tel: 45-4450-2828 Fax: 45-4485-2829</p> <p>Finland - Espoo Tel: 358-9-4520-820</p> <p>France - Paris Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79</p> <p>Germany - Garching Tel: 49-8931-9700</p> <p>Germany - Haan Tel: 49-2129-3766400</p> <p>Germany - Heilbronn Tel: 49-7131-72400</p> <p>Germany - Karlsruhe Tel: 49-721-625370</p> <p>Germany - Munich Tel: 49-89-627-144-0 Fax: 49-89-627-144-44</p> <p>Germany - Rosenheim Tel: 49-8031-354-560</p> <p>Israel - Ra'anana Tel: 972-9-744-7705</p> <p>Italy - Milan Tel: 39-0331-742611 Fax: 39-0331-466781</p> <p>Italy - Padova Tel: 39-049-7625286</p> <p>Netherlands - Drunen Tel: 31-416-690399 Fax: 31-416-690340</p> <p>Norway - Trondheim Tel: 47-72884388</p> <p>Poland - Warsaw Tel: 48-22-3325737</p> <p>Romania - Bucharest Tel: 40-21-407-87-50</p> <p>Spain - Madrid Tel: 34-91-708-08-90 Fax: 34-91-708-08-91</p> <p>Sweden - Gothenberg Tel: 46-31-704-60-40</p> <p>Sweden - Stockholm Tel: 46-8-5090-4654</p> <p>UK - Wokingham Tel: 44-118-921-5800 Fax: 44-118-921-5820</p>