ScienceDirect

Mikroc Pro

Related terms:

Development Board, Waveform, Library Function, Microcontroller, Program Listing, Project Hardware, Target Microcontroller

mikroC Pro for PIC32 Built-in Library Functions

Dogan Ibrahim, in Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC, 2014

mikroC Pro for PIC32 language provides a large number of built-in library functions that can be called from anywhere in a program, and there is no need to include header files at the beginning of the program. Most of the library functions are known as hardware functions and they have been developed for peripheral devices. In addition, standard ANSI C and miscellaneous software libraries are provided. mikroC Pro for PIC32 user manual gives a detailed description of each built-in function. In this chapter, we shall be looking at the details of some commonly used important functions.

PIC32 Microcontroller Development Tools

Dogan Ibrahim, in Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC, 2014

C Programming for 32-Bit PIC Microcontrollers

Dogan Ibrahim, in Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC, 2014

3.3 PIC32 Microcontroller Specific Features

mikroC Pro for PIC32 language differs from the ANSI C standard in certain ways. The modifications have been implemented to make the language more specific to the PIC32 series of microcontrollers. In this section, we shall be looking at some of the important features of the mikroC Pro for PIC32 language which are specific to the architecture of the PIC32 microcontrollers.

mikroC Pro for PIC32 language includes the following keywords which have specifically included to support the PIC32 microcontrollers:

code: This memory type is used for allocating **constants** in program memory. An example is given below:

```
const code x, y, z;
```

data: This memory type is used for storing variables in data RAM. An example is given below:

```
data int x, y, z;
```

rx: This memory type is used to store variables in the working registers. An example is given below:

```
rx char x;
```

sfr: This memory type allows the programmer to access the SFRs. An example is given below:

```
char Temp at PORTB;
```

at: This is used as an alias to a variable. An example is given below:

```
char x at PORTA;
```

sbit: This data type is used to provide access to processors registers and SFR. An example is given below:

```
sbit LCD_RS at LATB2_bit;
```

bit: This data type is used to declare single bit variables. An example is given below:

```
bit flag;
```

iv: Used to inform the compiler that this is an ISR.

3.3.1 SFR Registers

All PIC32 microcontroller SFR registers are defined as global variables of type **volatile int**. Access to all these registers and their bits are available in our programs. Some examples are given below:

```
PORTB
```

```
PORTB.RB0
```

```
INTCONbits.TPC
```

3.3.2 Linker Directives

mikroC for PIC32 linker supports a number of useful directives as explained below.

Directive Absolute

Specifies the starting address of a variable in RAM, or a constant in ROM. Some examples are given below:

```
char x absolute 0xA0000000;    // x occupies a byte in
0xA0000000
```

```
int x absolute 0xA0000000;     // x occupies 2 bytes staring
from 0xA0000000
```

Directive org

This directive specifies the starting address of a routine in ROM. **Org** is appended to a function definition. An example is shown below:

```
void Func(void) org 0xBA000000    // Function starts at
0xBA000000
```

```
{
```

```
}
```

Directive orgall

This directive is used to place all routines, constants, etc. above a specified ROM address.

Directive funcorg

With this directive, we can specify the starting address of a routine in ROM using the routine name. An example is given below:

```
#pragma funcorg <function name> <starting address>
```

3.3.3 Built-in Utilities

mikroC Pro for PIC32 language provides a number of utilities that could be useful in programs. Some of the important utilities are summarized in this section. Further details can be obtained from the mikroC Pro for PIC32 User Guide.

Lo: Returns (or sets) the low byte of a number. For example,

```
If x = 0x12345678 then z = Lo(x) returns 0x78
```

or

```
Lo(x) = 0x11 sets x to x = 0x12345611
```

Hi: Returns (or sets) the high byte of a number. For example,

```
If x = 0x12345678 then z = Hi(x) returns 0x56
```

Higher: Returns (or sets) the higher byte of a number. For example,

```
If x = 0x12345678 then z = Higher(x) returns 0x34
```

Highest: Returns (or sets) the highest byte of a number. For example,

```
If x = 0x12345678 then z = Highest(x) returns 0x12
```

LoWord: Returns (or sets) the low word of a number. For example,

```
If x = 0x12345678 then z = LoWord(x) returns 0x5678
```

HiWord: Returns (or sets) the high word of a number. For example,

```
If x = 0x12345678 then z = HiWord(x) returns 0x1234
```

Delay_us: Creates delay in microseconds.

Delay_ms: Creates delay in milliseconds.

Delay_Cyc: Creates delay based on the clock frequency.

Clock_MHz: Returns the clock frequency in MHz.

EnableInterrupts: Enables interrupts.

DisableInterrupts: Disables interrupts.

3.3.4 Start-up Initialization

Upon reset the MCU jumps to address 0xBFC00000 where the **BootStartUp** function is located. Before the processor jumps to the main program, the following actions take place:

The **BootStartUp** function configures:

- CP0 coprocessor register
- SFR registers associated with the interrupt
- Stack pointer and global pointer.

The processor is configured by default as

- cache and prefetch are enabled;
- flash wait states are set;
- executable code is allocated in the KSEG0;
- data are allocated in the KSEG1.

3.3.5 Read–Modify–Write Cycles

There are many applications where we may want to read the value of a port pin, modify it, and then write it back. This is commonly known as the read–modify–write cycle. There are some issues that we should be careful about when performing read–modify–write cycles.

When data are read from a port pin, we read the actual physical state of that port pin. Now, the problem arises if the port pin is loaded by the external hardware such that the state of the port pin may be affected. For example, if the port pin is connected to a capacitive load, the state of the port pin may be affected if the capacitor is charged or discharged. Thus, what we are reading is not the true state of the port pin, but rather the physical state of the port pin.

The read–modify–write cycle problem can be avoided by using the **LATx** registers when writing to and reading from ports, rather than **PORTx** registers. Writing to a **LATx** register is equivalent to writing to a **PORTx** register, but reading from a **LATx** register returns the data held in the port latch register, regardless of the physical state of the port pin. Thus, reading will not be affected by the external hardware connected to the port pin. It is recommended to use the **LATx** port register instead of the **PORTx** register, especially while reading from port pins connected to external devices such as LCDs, GLCDs, capacitive and inductive loads.

3.3.6 Interrupts

Interrupts are an important and complex topic in microcontroller programming. The interrupt handling mechanisms of the mikroC Pro for PIC32 language are described in detail in the projects sections of this book.

Advanced PIC18 Projects

Dogan Ibrahim, in PIC Microcontroller Projects in C (Second Edition), 2014

mikroC Pro for PIC CAN Functions

mikroC Pro for the PIC language provides two sets of libraries for CAN bus applications: the library for PIC microcontrollers with built-in CAN modules, and the library based on the use of SPI bus for PIC microcontrollers having no built-in CAN modules. In this project, we shall only be looking at the library functions available for PIC microcontrollers with built-in CAN modules. Similar functions are available for PIC microcontrollers with no built-in CAN modules.

mikroC CAN functions are supported only by PIC18XXX8 microcontrollers with MCP2551 or similar CAN transceivers. Both standard (11 identifier bits) and extended format (29 identifier bits) messages are supported.

The following mikroC functions are provided:

- CANSetOperationMode,
- CANGetOperationMode,
- CANInitialize,
- CANSetBaudRAte,
- CANSetMask,
- CANSetFilter,
- CANRead,
- CANWrite.

These functions are described below.

1. **CANSetOperationMode**

This function is used to set the CAN operation mode. The function prototype is

void CANSetOperationMode(**char** mode, **char** wait_flag)

Parameter **wait_flag** is either 0 or 0xFF. If set to 0xFF, the function blocks and will not return until the requested mode is set. If 0, the function returns as a nonblocking call.

The mode can be one of the following:

- _CAN_MODE_NORMAL - normal mode of operation,
- _CAN_MODE_SLEEP - SLEEP mode of operation,
- _CAN_MODE_LOOP - Loop-back mode of operation,

- _CAN_MODE_LISTEN - Listen Only mode of operation,
- _CAN_MODE_CONFIG - Configuration mode of operation,

2. **CANGetOperationMode**

This function returns the current CAN operation mode. The function prototype is

char CANGetOperationMode(void)

3. **CAN_Initialize**

This function initializes the CAN module. All mask registers are cleared to 0 to allow all messages. Upon execution of this function, the Normal mode is set. The function prototype is

```
void CANInitialize(char SJW, char BRP, char PHSEG1,
                                   char PHSEG2,char PROPEG,
char CAN_CONFIG_FLAGS)
```

where

SJW is the Synchronization Jump Width,

BRP is the Baud Rate prescaler,

PHSEG1 is the Phase_Seg1 timing parameter,

PHSEG2 is the Phase_Seg2 timing parameter,

PROPSEG is the Prop_Seg.

CAN_CONFIG_FLAGS can be one of the following configuration flags:

Value	Meaning
_CAN_CONFIG_DEFAULT	Default flags
_CAN_CONFIG_PHSEG2_PRG_ON	Use supplied PHSEG2 value
_CAN_CONFIG_PHSEG2_PRG_OFF	Use a maximum of PHSEG1 or information processing Time whichever is greater
_CAN_CONFIG_LINE_FILTER_ON	Use the CAN bus line filter for wake-up
_CAN_CONFIG_FILTER_OFF	Do not use a CAN bus line filter
_CAN_CONFIG_SAMPLE_ONCE	Sample bus once at the sample point
_CAN_CONFIG_SAMPLE_THRICE	Sample bus three times prior to the sample point

Value	Meaning
_CAN_CONFIG_STD_MSG	Accept only standard identifier messages
_CAN_CONFIG_XTD_MSG	Accept only extended identifier messages
_CAN_CONFIG_DBL_BUFFER_ON	Use double buffering to receive data
_CAN_CONFIG_DBL_BUFFER_OFF	Do not use double buffering
_CAN_CONFIG_ALL_MSG	Accept all messages including invalid ones
_CAN_CONFIG_VALID_XTD_MSG	Accept only valid extended identifier messages
_CAN_CONFIG_VALID_STD_MSG	Accept only valid standard identifier messages
_CAN_CONFIG_ALL_VALID_MSG	Accept all valid messages

The above configuration values can be bitwise AND'ed to form complex configuration values.

4. **CANSetBaudRate**

This function is used to set the CAN bus baud rate. The function prototype is

```
void CANSetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2,
                    char PROPSEG, char CAN_CONFIG_FLAGS)
```

the arguments of the function are as in function *CANInitialize*.

5. **CANSetMask**

This function sets the mask for filtering of messages. The function prototype is

```
void CANSetMask(char CAN_MASK, long value, char
                CAN_CONFIGFLAGS)
```

CAN_MASK can be one of the following:

- _CAN_MASK_B1—Receive Buffer 1 mask value,
- _CAN_MASK_B2—Receive Buffer 2 mask value.

value is the mask register value. CAN_CONFIG_FLAGS can be either _CAN_CONFIG_XTD (extended message), or _CAN_CONFIG_STD (standard message).

6. **CANSetFilter**

This function sets filter values. The function prototype is

```
void CANSetFilter(char CAN_FILTER, long value, char
                  CAN_CONFIG_FLAGS)
```

CAN_FILTER can be one of the following:

- _CAN_FILTER_B1_F1—Filter 1 for Buffer 1,
- _CAN_FILTER_B1_F2—Filter 2 for Buffer 1,
- _CAN_FILTER_B2_F1—Filter 1 for Buffer 2,
- _CAN_FILTER_B2_F2—Filter 2 for Buffer 2,
- _CAN_FILTER_B2_F3—Filter 3 for Buffer 2,
- _CAN_FILTER_B2_F4—Filter 4 for Buffer 2.

CAN_CONFIG_FLAGS can be either _CAN_CONFIG_XTD (extended message), or _CAN_CONFIG_STD (standard message).

7. **CANRead**

This function is used to read messages from the CAN bus. Zero is returned if no message is available. The function prototype is

```
char CANRead(long *id, char *data, char *datalen, char
             *CAN_RX_MSG_FLAGS)
```

id is the CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended). **data** is an array of bytes up to eight where the received data are stored. **datalen** is the length of the received data (1–8).

CAN_RX_MSG_FLAGS can be one of the following:

- _CAN_RX_FILTER_1—Receive Buffer Filter 1 accepted this message.
- _CAN_RX_FILTER_2—Receive Buffer Filter 2 accepted this message.
- _CAN_RX_FILTER_3—Receive Buffer Filter 3 accepted this message.
- _CAN_RX_FILTER_4—Receive Buffer Filter 4 accepted this message.
- _CAN_RX_FILTER_5—Receive Buffer Filter 5 accepted this message.
- _CAN_RX_FILTER_6—Receive Buffer Filter 6 accepted this message.
- _CAN_RX_OVERFLOW—Receive buffer overflow occurred.
- _CAN_RX_INVALID_MSG—Invalid message received.
- _CAN_RX_XTD_FRAME—Extended Identifier message received.
- _CAN_RX_RTR_FRAME—RTR frame message received.

- _CAN_RX_DBL_BUFFERED—This message was double-buffered.

The above flags can be bitwise AND'ed if desired.

8. **CANWrite**

This function is used to send a message to the CAN bus. A zero is returned if the message cannot be queued (buffer full). The function prototype is

```
char CANWrite(long id, char *data, char datalen, char
        CAN_TX_MSG_FLAGS)
```

id is the CAN message identifier. Only 11 or 29 bits may be used depending on the message type (standard or extended). **data** is an array of bytes up to eight where the data to be sent are stored. **datalen** is the length of the data (1–8).

CAN_TX_MSG_FLAGS can be one of the following:

- _CAN_TX_PRIORITY_0 - Transmit priority 0
- _CAN_TX_PRIORITY_1 - Transmit priority 1
- _CAN_TX_PRIORITY_2 - Transmit priority 2
- _CAN_TX_PRIORITY_3 - Transmit priority 3
- _CAN_TX_STD_FRAME - Standard Identifier message
- _CAN_TX_XTD_FRAME - Extended Identifier message
- _CAN_TX_NO_RTR_FRAME - Non-RTR message,
- _CAN_TX_RTR_FRAME - RTR message.

The above flags can be bitwise AND'ed if desired.

Simple PIC18 Projects

Dogan Ibrahim, in PIC Microcontroller Projects in C (Second Edition), 2014

mikroC Pro for PIC

The mikroC Pro for PIC program is named MIKROC-LED1.C, and the program listing is given in Figure 5.5. At the beginning of the program, PORTC pins are configured as outputs by setting TRISC = 0. Then, an endless **for** loop is formed, and the LEDs are turned ON alternately in an anticlockwise manner to give the chasing effect. The program checks continuously so that when LED 7 is turned ON the next LED to be turned ON is LED 0.

```
/*************************************************************************
                              CHASING LEDS
                              ============

In this project 8 LEDs are connected to PORTC of a PIC18F45K22 microcontroller and the
microcontroller is operated from an 8 MHz crystal.

The program turns ON the LEDs in an anticlockwise manner with 1 s delay between
each output. The net result is that LEDs seem to be chasing each other.


Author:         Dogan Ibrahim
Date:           August 2013
File:           MIKROC-LED1.C
*************************************************************************/

void main()
{
   unsigned char J = 1;

   ANSELC = 0;                          // Configure PORTC as digital
   TRISC = 0;                           // Configure PORTC as output
   for(;;)                              // Endless loop
   {
       PORTC = J;                       // Send J to PORTC
       Delay_ms(1000);                  // Delay 1 s
       J = J << 1;                      // Shift left J
       if(J == 0) J = 1;                // If last LED, move to first one
   }
}
```

Figure 5.5. mikroC Pro for PIC Program Listing.

The program is compiled using the mikroC compiler. Project settings should be configured to an 8-MHz clock, XT crystal mode, and WDT OFF. The HEX file (MIKROC-LED1.HEX) should be loaded to the PIC18F45K22 microcontroller using an in-circuit debugger, a programming device, or the EasyPIC V7 development board.

When using the mikroC Pro for PIC compiler, the configuration fuses can be modified from the Edit Project window that is entered by clicking Project → Edit Project.

Intermediate PIC18 Projects

Dogan Ibrahim, in PIC Microcontroller Projects in C (Second Edition), 2014

mikroC Pro for PIC

The mikroC Pro for the PIC program listing is given in Figure 6.57 (MIKROC-MUSIC.C). The program is very simple. The notes are stored in an array called Notes. The program initializes the keypad library and the sound library. Then, an endless loop is formed, and inside this loop, the code of the pressed key is determined, and this is used as an index to the Notes array to play the note corresponding to the pressed key. Notes are played for a minimum of 100 ms when a key is pressed. Valid key codes are from 1 to 13, and keys with codes >13 are not played.

```
/*************************************************************************
                        MINI ELECTRONIC ORGAN
                        =====================

In this project a 4 x 4 keypad is connected to PORTC of a PIC18F45K22 microcontroller. Also a
buzzer is connected to port pin RE1. The keypad is organized such that pressing a key plays a
musical note. The notes on the keypad are organized as follows:

C4    D4   E4   F4
G4    A4   B4   C5
C4#  D4#  F4#  G4#
A4#

Author:   Dogan Ibrahim
Date:     September 2013
File:     MIKROC–MUSIC.C
*************************************************************************/
// Keypad module connections
char keypadPort at PORTC;
// End of keypad module connections

//
// Start of MAIN program
//
void main()
{
   unsigned char MyKey;
   unsigned Notes[] = {0,262,294,330,349,392,440,494,524,277,311,370,415,466};

   ANSELE = 0;                                  // Configure PORTE as digital
   ANSELC = 0;                                  // Configure PORTC as digital
   TRISC = 0xF0;                                // RC4–RC7 are inputs

   Keypad_Init();                               // Initialize keypad library
   Sound_Init(&PORTE, 1);                       // Initialize sound library
//
// Program loop
//
   for(;;)                                      // Endless loop
   {
      do
         MyKey = Keypad_Key_Press();            // Get code of pressed key
      while(!MyKey);

      if(MyKey <= 13)Sound_Play(Notes[MyKey], 100);  // Play the note
   }
}
```

Figure 6.57. mikroC Pro for PIC Program Listing.

mikroC Pro for PIC Programming Language

Dogan Ibrahim, in PIC Microcontroller Projects in C (Second Edition), 2014

2.9 Summary

This chapter presented an introduction to the mikroC Pro for PIC language. A C program may contain a number of functions and variables and a main program. The beginning of the main program is indicated by the statement **void main()**.

A variable stores a value used during the computation. All variables in C must be declared before they are used. A variable can be an 8-bit character, a 16-bit integer, a 32-bit long, or a floating point number. Constants are stored in the flash program memory of PIC microcontrollers and thus using them saves valuable and limited RAM.

Various flow control and iteration statements such as **if**, **switch**, **while**, **do**, **break**, and so on have been described in the chapter with examples.

Pointers are used to store the addresses of variables. As we shall see in the next chapter, pointers can be used to pass information back and forth between a function and its calling point. For example, pointers can be used to pass variables between a main program and a function.

Library functions simplify programmers' tasks by providing ready and tested routines that can be called and used in our programs. Examples are also given on how to use various library functions in our main programs.

MPLAB X IDE and MPLAB XC8 C Programming Language

Dogan Ibrahim, in PIC Microcontroller Projects in C (Second Edition), 2014

3.2 MPLAB X IDE

The MPLAB X IDE is the integrated development environment (just like the mikroC Pro for PIC IDE) that enables the user to create a source file, edit, compile, simulate, debug, and send the generated code to the target microcontroller with the help of a compatible programmer/debugger device. The Microchip Inc. web site (http://www.microchip.com/pagehandler/en-us/family/mplabx/) contains

detailed information on the features and the use of the MPLAB X IDE. In this chapter, we will be looking at the program development steps using this IDE.

MPLAB X IDE is available at the Microchip Inc. web site (http://www.microchip.com/pagehandler/en-us/family/mplabx/#downloads), and it must be installed on your PC before it can be used. At the time of writing this book, the latest version of the MPLAB X IDE was v1.85.

Advanced PIC32 Projects

Dogan Ibrahim, in Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC, 2014

8.9 Project 8.9—Calculating Timing in Digital Signal Processing

8.9.1 Project Description

DSP algorithms require MAC operations using fixed-point or floating point arithmetic. In this project, we will calculate the time required to carry out floating point MAC operations using the mikroC Pro for PIC32 compiler, and the PIC32MX460F512L microcontroller with 80 MHz clock. Various MAC operations are performed and the time it takes to carry out these operations is displayed on an LCD.

8.9.2 Project Hardware

The project hardware is same as shown in Figure 7.17. An LCD is connected to PORT B of the microcontroller and the microcontroller is configured to operate at 80 MHz clock rate using the built-in PLL module.

In this project, the LV-32MX V6 development board is used. The following jumpers should be configured on the board to enable the on-board LCD:

```
DIP switch SW20 positions 1–6, set to ON
```

8.9.3 Project PDL

The PDL of the project is shown in Figure 8.58. Timer 1 is configured to count up with a prescaler of 1 (i.e. with a clock of 80 MHz, the count rate is 0.0125 μs). The counter is enabled before the MAC operations. At the end of the MAC operations, the timer count is multiplied by 0.0125 μs in order to calculate the elapsed time.

```
BEGIN
    Configure LCD connections to the microcontroller
    Disable Timer 1
    Configure Timer 1 to count at a rate of 0.0125 μs
    Start Timer 1
    Perform MAC operations
    Stop Timer 1
    Calculate elapsed time
    Display elapsed time on the LCD
END
```

Figure 8.58. Project PDL.

8.9.4 Project Program

The program listing is shown in Figure 8.59 (TIMING.C). At the beginning of the program, the LCD connection to the microcontroller is defined. Floating point variables *y*, *a* and *b* are then declared where *a* and *b* are loaded with some floating point numbers. Up to 500 locations are reserved for each variable.

```
/*************************************************************************************
                DIGITAL SIGNAL PROCESSING MAC TIMING
                ====================================

This project shows how to calculate the time it takes to carry out different number
of MAC operations.

An LCD is connected to PORT B of the PIC32MX460F512L microcontroller, operating at the
clock rate of 80 MHz (using the internal PLL). The program uses Timer 1 to calculate
the timing and then displayes the result on the LCD.


Author:         Dogan Ibrahim
Date:           September 2012
File:           TIMING.C
*************************************************************************************/

// LCD module connections
sbit LCD_RS at LATB2_bit;
sbit LCD_EN at LATB3_bit;
sbit LCD_D4 at LATB4_bit;
sbit LCD_D5 at LATB5_bit;
sbit LCD_D6 at LATB6_bit;
sbit LCD_D7 at LATB7_bit;

sbit LCD_RS_Direction at TRISB2_bit;
sbit LCD_EN_Direction at TRISB3_bit;
sbit LCD_D4_Direction at TRISB4_bit;
sbit LCD_D5_Direction at TRISB5_bit;
sbit LCD_D6_Direction at TRISB6_bit;
sbit LCD_D7_Direction at TRISB7_bit;
// End LCD module connections


//
// Start of man program
//
void main()
{
    float y, Tim, a[500], b[500];
    unsigned char Txt[14];
    unsigned int i;
//
// Load some floating point numbers to a and b
//
   for(i = 0; i < 500; i++)
   {
    a[i] = 2.85*i;
```

```
    b[i] = 3.57-i;
  }
//

// Configure Timer 1
//
  T1CONbits.ON = 0;                    // Disable Timer 1
  TMR1 = 0;                            // Clear TMR1 register
  T1CONbits.TCKPS = 0;                 // Select prescaler = 1
  T1CONbits.TCS = 0;                   // Select internal clock
  PR1 = 0xFFFF;                        // Load period register

  LCD_Init();                          // Initialize LCD
  LCD_Out(1,1, "TIME (us)");
  y = 0.0;

  T1CONbits.ON = 1;                    // Enable Timer 1
  for(i = 0; i < 10; i++)              // Start of MAC loop ( 10 operations here)
  {
    y = y + a[i] * b[i];
  }                                    // End of MAC loop
  T1CONbits.ON = 0;                    // Disable Timer 1

  Tim = 0.0125*TMR1;                   // Calculate elapsed time in us
  FloatToStr(Tim, Txt);                // Convert to string
  Lcd_Out(2,1, Txt);                   // Display elapsed time
}
```

Figure 8.59. Program listing for 10 MAC operations.

Timer 1 is then disabled and configured to count with a prescaler of 1, i.e. at a rate of 0.0125 μs. Timer 1 is enabled, a **for** loop is used to carry out a number of different MAC operations, and at the end, the elapsed time is calculated by multiplying the Timer 1 count with the clock rate. The time is then displayed on the LCD.

Table 8.5 shows results of the tests with various number of MAC operations.

Table 8.5. Timing Calculations

No of MAC Operations	Time (μs)
10	1.05

No of MAC Operations	Time (μs)
20	1.93
30	2.8
40	3.67
50	4.55
100	8.93
200	17.68
400	35.18
500	43.93

A graph of the time it takes to perform various number of MAC operations is shown in Figure 8.60.

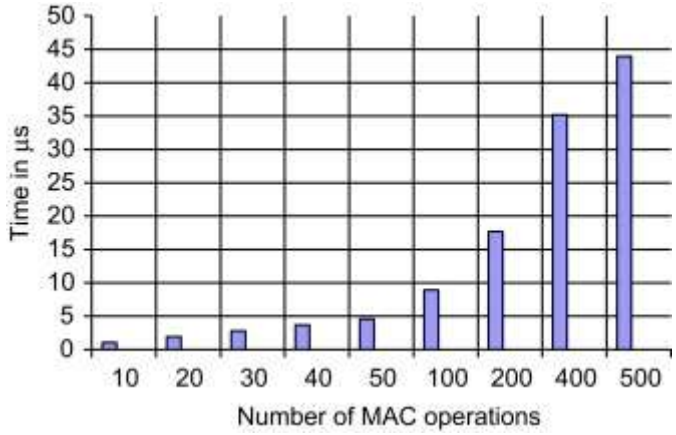


Figure 8.60. MAC timing. (For color version of this figure, the reader is referred to the online version of this book.)

PIC32 Microcontroller Series

Dogan Ibrahim, in Designing Embedded Systems with 32-Bit PIC Microcontrollers and MikroC, 2014

Configuring the Operating Clocks

The clock configuration bits shown in Figure 2.5 can be selected by programming the SFR registers (e.g. OSCCON, OSCTUN, OSCCONCLR and so on) or the device configuration registers (e.g. DEVCFG1 and DEVCFG2) during run time.

Alternatively, the operating clocks can be selected during the programming of the microcontroller chip. Most programming devices give options to users to select the operating clocks by modifying the device configuration registers just before the chip is programmed. The mikroC Pro for PIC32 compiler allows the configuration registers to be modified by selecting *Project → Edit Project* from the drop-down menu of the IDE.

Figure 2.6 shows the clocks that will be used in all the projects in this book (these are the default settings provided by mikroC Pro for PIC32 compiler). The selection is summarized below:

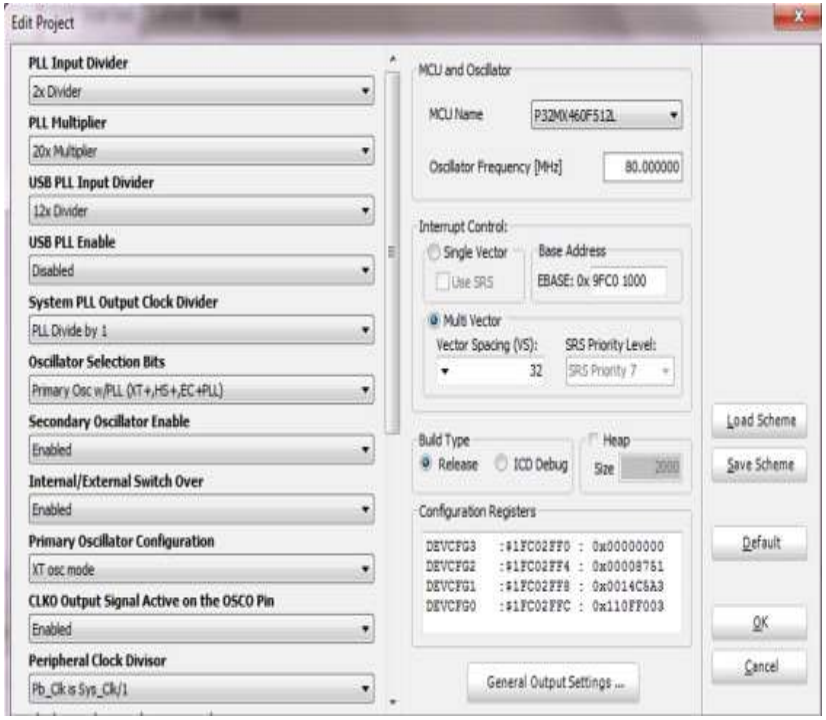


Figure 2.6. Edit project window. (For color version of this figure, the reader is referred to the online version of this book.)

- External 8 MHz crystal connected to OSCI and OSCO pins

- PLL Input Divider: 2
- PLL Multiplier: 20
- USB PLL Input Divider: 12
- USB PLL Enable: Enabled
- System PLL Output Clock Divider: 1
- Peripheral Clock Divisor: 1.

The above selection gives (Figure 2.7) the following:

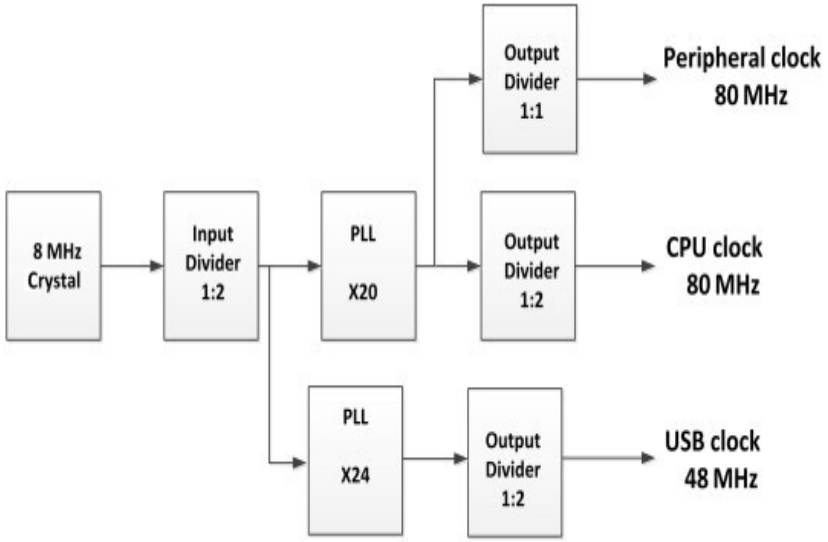
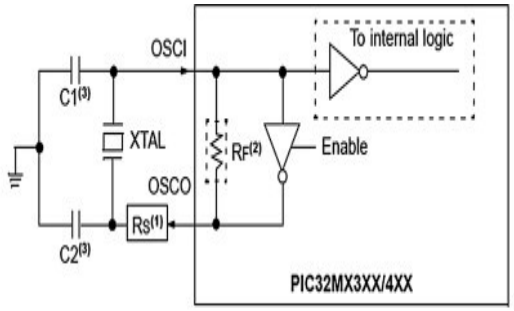


Figure 2.7. Clock selection for the projects.

CPU clock	80 MHz
USB clock	48 MHz
Peripheral clock	80 MHz

The crystal connections to OSCI and OSCO pins are shown in Figure 2.8 with two small capacitors. The connection of an external clock source in EC mode is shown in Figure 2.9. In this mode, the OSCO pin can be configured either as a clock output or as an I/O port.



Note 1: A series resistor, Rs, may be required for AT strip cut crystals.
2: The internal feedback resistor, RF, is typically in the range of 2 to 10 MΩ.
3: Refer to the "*PIC32MX Family Reference Manual*" (DS61132) for help determining the best oscillator components.

Figure 2.8. Crystal connection.

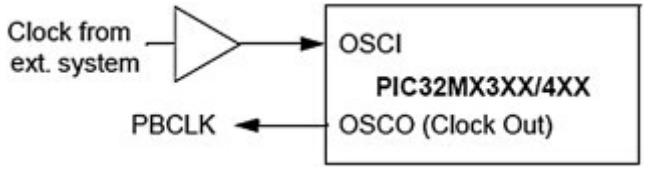


Figure 2.9. External clock connection.