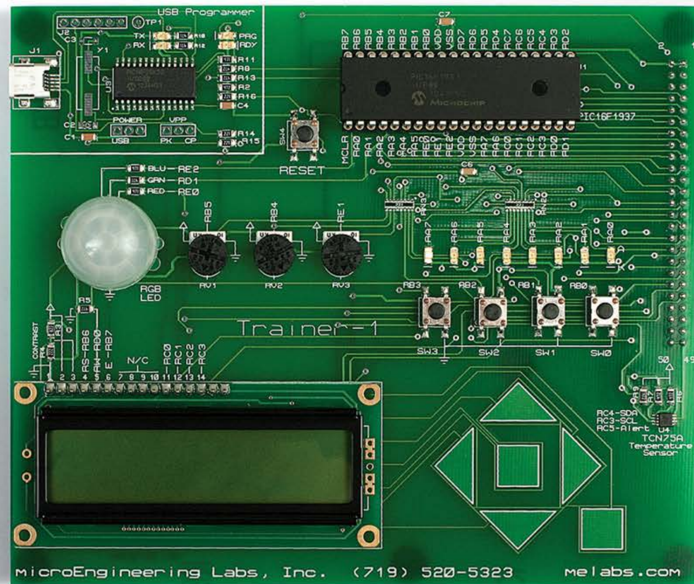
microEngineering Labs, Inc.
www.melabs.com

microEngineering Labs Trainer



Contents

Introduction 7
Chapter 1: Installation and System Test 10
1.1 Goals: 10
1.2 The Tools 10
1.3 Getting Started 10
1.4 Compile and Program 11
1.4.1 MicroCode Studio 11
1.4.2 Compilation with PICBASIC PRO™ Compiler (PBP) 11
1.4.3 Programming with meProg 12
1.5 Summary 13
1.6 Exercise 13
1.7 Troubleshooting 13
Chapter 2: Blink 14
2.1 Goals 14
2.2 Program Listing 14
2.3 Include configuration code 15
2.4 Code: blink an LED 15
2.5 Summary 16
2.6 Exercises 16
Chapter 3: Read Inputs and Drive Outputs 17
3.1 Goals 17
3.2 Program Listing 17
3.3 Code: read inputs and drive outputs 17
3.4 Debugging 18
3.4.1 Select COM Port 19
3.4.2 Special Compile 19
3.4.3 Controlling the program 19
3.5 Summary 20
3.6 Exercises 20
Chapter 4: LCD 21
4.1 Goals 21
4.2 Program Listing 21
4.3 Code: LCDOUT 21
4.4 Code: formatting ASCII strings 22
4.5 Code: count on screen 22
4.6 Summary 23

4.7 Exercises 24
Chapter 5: Analog Conversion 25
5.1 Goals....25
5.2 New concept: peripheral modules and SFRs 25
5.3 Program Listing....26
5.4 Define the conversion 26
5.5 Code: perform the conversions 28
5.6 Code: display values on screen 29
5.7 Summary....30
5.8 Exercises 30
Chapter 6: PWM 31
6.1 Goals....31
6.2 What is PWM and why do we use it? 31
6.3 Program Listing....32
6.4 HPWM....32
6.5 Code: create constants and variables 32
6.6 Code: control LEDs with POTs 34
6.7 Summary....34
6.8 Exercises 34
Chapter 7: Timers 36
7.1 Goals....36
7.2 Program Listing....36
7.3 Code: slowing things down 37
7.4 Code: declare Variables, Aliases, and Constants 38
7.5 Code: initialize the Timer1 module....38
7.6 Code: using Timer1 and its interrupt flag 39
7.7 Code: controlling the LEDs 40
7.8 Summary....42
7.9 Exercises 42
Chapter 8: Capacitive Touch....44
8.1 Goals....44
8.2 Program Listing....44
8.3 Code: declarations 45
8.4 Code: initializing registers 45
8.5 Code: the main loop....46
8.6 Code: checking for touch 46
8.7 Code: display capacitive sense readings with a nested subroutine 48

8.8 Summary....49
8.9 Exercises....49
Chapter 9: Timer-Driven Assembly Language Interrupt....50
9.1 Goals....50
9.2 Program Listing....50
9.3 Assembly Language and interrupts....51
9.4 Code: declarations and initialization....51
9.5 Code: jump over Assembly Language ISR....52
9.6 Code: the main loop....53
9.7 Summary....54
9.8 Exercises....54
Chapter 10: User Interface....56
10.1 Goals....56
10.2 Program Listing....56
10.3 Re-using program code....59
10.3.1 Reused Code: PWM output to drive the RGB LED....59
10.3.2 Reused Code: reading touch pad states....60
10.3.3 Reused Code: Timer1 interrupt....61
10.4 Code: startup....61
10.5 Code: the main loop....62
10.5.1 Code: check for user input....62
10.5.2 Parse user input and take actions....62
10.6 Code: LCD for feedback....65
10.7 Code: pace and auto-repeat....66
10.8 Summary....66
10.9 Exercises....67
Chapter 11: Binary Tricks of the Trade....68
11.1 Goals....68
11.2 Program Listing 1: Simple Binary Count....68
11.3 Code: bitwise AND as a "Mask"....69
11.4 Program Listing 2: Masked Counting....70
11.5 Program Listing 3: Restricted Count....71
11.6 Program Listing 4: Bit Toggle Rate....71
11.7 Summary....72
11.8 Exercises....73
Chapter 12: RS-232 Communication....74
12.1 Goals....74

12.2 Program Listing 74
12.3 Code: define hardware serial protocol 75
12.4 Code: array variable 75
12.5 Code: prompt with HSEROUT 75
12.6 Code: initialize array 76
12.7 Code: receive with HSERIN....76
12.8 Code: confirm with HSEROUT 76
12.9 Code: display input on LCD 77
12.10 Summary 77
12.11 Exercise:....77
Chapter 13: I2C Communication 78
13.1 Goals 78
13.2 Program Listing 78
13.3 The TCN75A and its Data Sheet 79
13.4 Code: define the I2C connection 79
13.5 Code: additional variables and aliases 79
13.6 Code: configure the temperature sensor 80
13.7 Code: query the sensor 80
13.8 Code: calculate values for displayed temperature 81
13.9 Code: displaying temperatures with 2 decimal places 82
13.10 Code: I2C error condition....83
13.11 Summary 83
13.12 Exercises: 83
Chapter 14: Device Configuration....84
14.1 Goals 84
14.2 Program Listing 84
14.3 Code: #CONFIG 85
14.4 Code: internal oscillator 86
14.5 Code: DEFINE OSC 86
14.6 I/O pins 86
14.6.1 Analog and Digital 87
14.6.2 Pull-ups for pushbuttons....87
14.6.3 Data Direction....88
14.6.4 Liquid-Crystal Display (LCD) 89
14.6.5 Initialization summary....90
Chapter 15: Appendixes 91
15.1 Bits, Bytes, and Words 91

15.2 The Microchip Data Sheet 92
15.2.1 Using registers (SFRs) 92
15.3 Schematic 95

microEngineering labs, inc. (the company) disclaims all warranties, express or implied, including without limitation the implied warranty of fitness for a particular purpose, the implied warranty of merchantability, and the implied warranty of the accuracy of the information presented in this document. In no event shall the company or its employees, agents, suppliers or contractors be liable for any incidental, indirect, special or consequential damages arising out of or in connection with the use of the products described herein, including without limitation, lost profits, downtime, goodwill, damage to or replacement of equipment or property, or any costs for recovering, reprogramming or reproducing any data used with the products.

PIC, PICmicro, dsPIC, and MPLAB are registered trademarks of Microchip Technology Inc. in the USA and other countries. MPASM, PICkit, PICBASIC, PICBASIC PRO, PICPROTO, and EPIC are trademarks of Microchip Technology Inc. in the USA and other countries.

Introduction

Welcome!

Whether you've never written a single word of code or you've been doing it for years, we've designed this trainer package to give you a thorough introduction to microcontroller programming. The microcontroller used on the board is the PIC16F1937 from Microchip. The programming language is BASIC using the PICBASIC PRO Compiler from ME Labs. The programming skills you develop here will apply to any microcontroller in any programming language.

Each chapter will serve to introduce one or two new concepts and in many cases will also incorporate lessons from previous chapters. The exercises offered at the end of each chapter are designed to challenge you to put into practice the lessons from both that chapter as well as those from previous chapters. As with any language, proficiency comes from practice, and the more engaged you become in the exercises the more fluent you'll become with PICBASIC PRO.

Chapters 1-10 represent the main learning progression of the tutorial. Chapters 1-9 will each build upon and utilize the lessons of the previous chapters and introduce some new concepts. Chapter 10 serves as the culminating exercise asking you to bring together all the previous skills presented earlier. Chapters 11, 12 & 13 include some often-utilized tools and techniques that we thought might be worth exploring in greater detail. Finally, Chapter 14 goes into a more advanced explanation of the microcontroller's configuration, which is included in all the chapter exercises.

If you are not familiar with the process of writing code it is advised that you read each lesson carefully and thoroughly - there are lots of details to understand. While "skimming" the instructions is something we've all done, in this particular instance you may find that you need to re-read sections to develop a complete understanding of the concepts. Along with this tutorial you will find that the **PBP Reference Manual** will be an invaluable guide to help answer questions and provide necessary tools for many of the exercises at the end of each chapter.

Making the most of the Exercises…

At the end of each chapter, we've provided exercises in the form of short programming tasks. It will be through these that you will develop a fluency and a deeper understanding of just how things work.

Each exercise will be based on a program that we provide. Usually you will be asked to modify a program to work in a different way. In some of the exercises we will provide a program that doesn't work and you will be asked to fix it (debug it).

For each exercise, we've also provided a program example as a solution to the exercise. One of the great things about writing code is that there is no one "right" way to get a task accomplished. If the program behaves the way you intended – then it's correct. There are many ways to accomplish a given task and we've provided code for each exercise that offers *one solution* to that exercise.

You may find that you've accomplished the task successfully only to see that your code is different than the solution we've provided. That is actually the *best* possible outcome for a couple of reasons: First it means that you've been able to successfully accomplish the task. Next, it means that you now have an example of another way to do the same thing! You can look at the code and see the similarities and differences. If you don't understand something in the sample code, you can easily load it and use the ICD (In-Circuit Debugging) feature in MicroCode Studio (that will be explained in chapter 3) to step through each line of code and see exactly what it does.

File organization:

The example programs are found in the ME Labs Trainer EXAMPLES folder. They are named starting with a number for the chapter. (2-1_blink.pbp is associated with chapter-2 and 3-1_LED_Button.pbp is associated with chapter-3.)

The solutions to exercises are placed in a subfolder named EXERCISES. They too are named according the chapters they are association with. If the exercise is a debugging exercise, in which we provide broken code, the solution will described in comments at the bottom of the code listing.

We provided a subfolder called MY PROGRAMS in which we recommend you save any program that you write or modify. This keeps your programs separate and maintains the provided programs in their unchanged state.

The file 14-1_Init.pbp is copied to each of these folders so that it may be called with an 'INCLUDE' command from any of the other programs.

The easiest way to prepare a provided program for modification is to use the Save As function in MicroCode Studio to make a copy of the program within your MY PROGRAMS folder. For example, to modify the chapter-2 example (2-1_blink.pbp), you would:

Click File > Open, select the file 2-1_blink.pbp, and open it.

Click File > Save As, navigate to the MY PROGRAMS folder, rename the file, and save it.

You may use any name you want for your program files, but we recommend a bit of thought toward organization. It's a good idea to maintain some reference to the chapter number as we have in the examples and exercises. You will likely accumulate a lot of files. It helps if the file names are descriptive. Filenames can be any length that Windows will accept, but we recommend that you keep them fairly short in the interest of visibility in file dialog windows. Extra dots in filenames cause problems, so DON'T name your files in the form: my_program_version_1.1.pbp.

If you wish to restore all the provided files to their original state, you may use the included utility ***Setup Trainer for Current User***, which can be found in the ME Labs Trainer program group.

Thoughts on number systems… (Chapter 11)

"Do I really need to learn binary and/or hexadecimal number systems?" The short answer is "No, there are plenty of folks who can write code and do not understand the details of these systems." ***however,*** the time you take to develop a deeper understanding of these systems (particularly binary) will certainly make it easier to move through each lesson. Understanding just the basics of the binary number system will deepen your ability to write more effective and creative code by a big margin.

The binary (and hexadecimal) system is an integral aspects of all computational hardware and if you understand these you are much more likely to make the most of the immense power that microcontrollers have. Simply put, you don't need to, but if you do you will be really glad you did. For this reason we've added a lesson (Chapter 11) which serves to introduce you to some techniques using bitwise operators on binary numbers. You'll find that moving beyond familiarity and into fluency with the binary world will encourage a good deal more creativity in your code writing as well as in your problem solving skills in general.

Some final words before we start…

If you are completely new to programming and this is your first time seeing any of this information you may get to a point where you don't understand some (or all) of what's being described to you. That's to be expected. There will likely be new terms and usage that, initially, may not make sense. Remember that you are learning a new language, it may take some time to develop an "ear" for it. If you've ever been to a foreign country, you may remember that after a little while, accent's and language seemed to get easier to understand and this isn't much different.

Keep in mind that you are not expected to know the specific concepts, commands and phrases that we are introducing in each chapter. They are *supposed* to be unfamiliar to you – that's why you are here! You will likely find that after doing some of the exercises the terms and concepts will begin to make more sense. Also, keep in mind that there is nothing wrong with going back and looking at the code you've created in the previous chapters' exercises. Sometimes it is best to keep moving forward and trust that things will reveal themselves after a bit of

experimentation and experience. The best way to become fluent in any language is by using it and that's why we've provided exercises at the end of each chapter.

Keep in mind also that the PBP Reference Manual is available. This tutorial describes PBP commands in a brief, simplified manor. The reference manual offers much more in-depth information.

Learning any new skill requires a balance of patience and persistence. If you're feeling "stuck", you may find that taking a break to let things ferment a bit can work wonders and when you return things often seem clearer. We hope you enjoy the process!

Chapter 1: Installation and System Test

1.1 Goals:

- Installation of software
- Identifying and understanding the tools that are used
- Achieve a successful “first compile”, proving that the tools are properly installed

The goal of this chapter is to install all of the tools and run through a first-time compile/program process. When successful, the result will be a blinking LED on the Trainer.

The example program for this chapter is named “1-1_Test.PBP”.

1.2 The Tools

Installation Procedure:

- Install all software applications.
- Connect the Trainer to a USB port and allow Windows to install the driver.
- Run the MCSX (MicroCode Studio) application to begin.

The ME Labs Trainer package includes a CD with all the software that you need. Insert the CD in the drive and let it run. If necessary, browse the CD and run the setup program. In most cases, it is recommended not to change the default installation locations.

The ME Labs Trainer software provides the capability of uploading programs to the trainer board. It also includes the example programs and documentation that you will need to use the product. There are two more applications that must be installed in addition to the Trainer software:

PICBASIC PRO Compiler – The application that turns BASIC programs into machine language programs.

MicroCode Studio – The code editor and user interface. This is where you work.

These included applications will be offered as checkbox-options at the end of the Trainer installation process. Each will install separately using its own installer.

After the software is installed, connect the trainer board to a USB port. If asked for driver software, you may tell Windows to search automatically. It’s important that the driver is installed successfully. If you encounter unexpected errors, you may contact ME Labs for technical support.

1.3 Getting Started

There are three main aspects to this system that you will need to make sure are loaded properly.

- MicroCode Studio, which serves as the interface where you’ll be writing your programs.
- PICBASIC PRO™ Compiler, which works through MicroCode Studio and is the “translator” that converts your programs to machine-language that runs on the PIC Microcontroller.
- The drivers that allow the PC to program and communicate with the Trainer Board.

When these are loaded initially, it is important to make sure each piece is correctly installed. Once you’ve loaded everything on the CD, you’ll do the same thing a software engineer would do – check the installation with a simple blinking light program.

IMPORTANT NOTE: If the software was previously installed by a different user-account on the computer (I.T. personnel, instructor at school, etc.), you will need to set up your user folder before proceeding. To do so, make sure that MicroCode Studio is shut down and then click ***Start > All Programs > ME Labs Trainer > Setup Trainer for Current User***. This will create the "ME Labs Trainer" folder in your documents folder, placing the examples and

documentation there. You may also use this folder to restore (overwrite) the examples as well as save new programs that you write for the exercises at the end of each chapter.

Once installed, you should begin by running MicroCode Studio. Its icon is named MCSX and can be found on your desktop and in your Start menu.

MicroCode Studio should automatically open the example named: "1-1_Test.pbp". (If it doesn't, stop here and go back two paragraphs.) This is a very simple program that blinks an LED on the trainer board. In the engineering world, this is the standard first step when bringing up new hardware. We want to prove that the hardware is sound and that our software tools are working. Blinking an LED is a fundamental function that should return quick results.

In most new installations MicroCode Studio will open with the first example program already displayed. If it doesn't show the program ***1-1_Test.pbp***, click **File > Open** and find the file on your system. It should be located in your documents folder ("My Documents") under "**ME Labs Trainer\EXAMPLES**\".

Important consideration: Before editing or changing the existing program, click "Save As…" and create a new title (like "Button1") in the MY PROGRAMS folder. This helps keep the original files intact and allows you to revisit older files you created.

1.4 Compile and Program

1.4.1 MicroCode Studio

Please observe that at this point only one part of the system is running – MicroCode Studio. This is the application that manages your program files, allows you to create and edit code, and –when the time comes– launches the compiler and programming utility. MicroCode will also provide the debugging capability (more on this later).

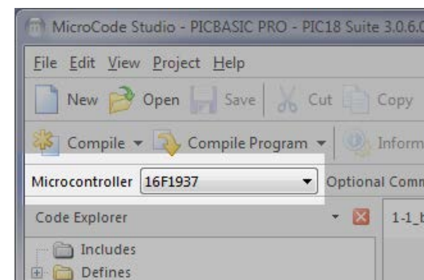


Figure 1-1: Select Microcontroller

It's critical that MicroCode knows the part number of the microcontroller that is installed on the trainer board. Find the dropdown labeled "Microcontroller" on the MicroCode toolbar make sure that the selection is "**16F1937**". If you look carefully at the 40-pin microcontroller chip on the trainer board, you will see that the full part number is "PIC16F1937-I/P".

At this point, with MicroCode running, the proper file should be open, and the proper target device (16F1937) should be selected.

1.4.2 Compilation with PICBASIC PRO™ Compiler (PBP)

The next step is to compile the example program. The compilation process changes the BASIC program that you see in the MicroCode Studio window into a machine language file that the microcontroller can understand. The tool that performs this step is the PICBASIC PRO™ Compiler (PBP). PBP is a BASIC compiler, meaning that it compiles BASIC-language programs. If you wished to use another language, you would acquire a different compiler for the language of choice.

To invoke PBP, click the "Compile" button on the MicroCode toolbar. (Note that there is also a button named "ICD Compile", which is something different that will be presented later.) You should see a popup dialog with a green progress bar that shows the brief compilation/assembly progress. Once it disappears, the status indicator at the lower-left corner of the MicroCode window should show "Success".

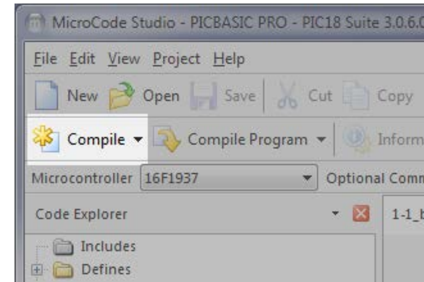


Figure 1-2: Click "Compile"

A successful compile indicates that each of the different parts of the system have been installed and configured correctly. If you encounter an error message at this point, you should resolve it before continuing. (See 1.6 Troubleshooting for clues.)

(If, upon first compile, the PBP Activation Manager pops up, enter the provided Activation Key for PBP along with your name and email. Once activated, this dialog should remain invisible.)

When you clicked on the Compile button, this is what actually happened:

- MicroCode Studio made sure that any edits to the program file were saved.
- MicroCode then called the compiler (PBP) and told it what device to compile for (16F1937) and what file to compile (*1-1_Test.pbp*).
- PBP read the file (1-1_Test.pbp) and performed its magic to create a file called blink.asm that holds the Assembly Language version of the blink program.
- PBP then called the Microchip MPASM assembler and told it what file to assemble (*1-1_Test*.asm). The green progress bar that you saw was actually MPASM running. (I only mention the MPASM assembler for full disclosure at this point. You need not be concerned with Assembly Language until much later.)
- MPASM read the file (*1-1_Test*.asm) and created a machine-language file (*1-1_Test*.hex). The .hex file is now ready to be programmed into the microcontroller.

If all this happened with no indication of an error, then your compile worked and you have proven the operation of MicroCode Studio, PBP (and MPASM). If there was an error, review the installation steps and look for something that may have been missed. If you're still stumped, check the ME Labs forum or contact technical support.

1.4.3 Programming with meProg

Now that you've compiled successfully… well, you're going to do it again. This time, though, instead of the "Compile" button, click the "Compile Program" button. The process of compiling and process of programming are actually two distinct steps. The compile step serves as the translation and success compiling means that the translation was completed, but we don't know if the program itself will do what we intended. The "Compile Program" button does the translation and then sends the (translated) program to the microprocessor for it to execute the instructions.

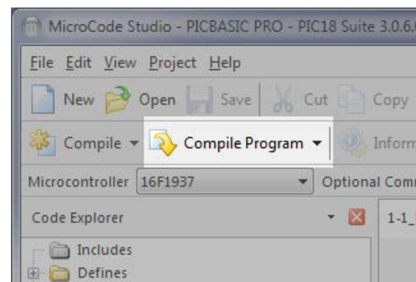


Figure 1-3: Click "Compile Program"

You'll see the same process repeat, but the programming progress bar will appear after the compile. The progress bar is part of the programming software (meProg) that was installed as part of the Trainer package. It's important to understand that this is separate piece of software that sends the machine-language code to the microcontroller.

Here's what happened this time:

- The compile/assembly process executed as detailed previously.
- MicroCode then launched the programmer software (meProg) and told it what device is on the trainer board (16F1937) and the name of the newly-created hex file (_1-1_Test_.hex).
- When the programmer finished, the microcontroller automatically began executing the program code on the Trainer board.

If everything went as we intended and all the parts were loaded and are functioning you should see a blinking light on the board. Success! You have just programmed a microcontroller. If you did run into some problems, have a look at the Troubleshooting section at the end of the chapter and that should help you sort things out.

1.5 Summary

You now have a proven, working system for microcontroller development. You've tested the software and the hardware to achieve the goal of blinking an LED.

1.6 Exercise

1. Open the program exercise file (1-1_Factory_Test.pbp), program the Trainer and follow the instructions on the LCD. This is how we test and prepare the boards before shipment.

1.7 Troubleshooting

Here are some common issues that you may encounter upon first installation and use.

"{ERROR} 1-1_test.pbp (15) : Syntax error (or Symbol not previously defined) – : When you open a new file in MicroCode Studio, ALWAYS check to make sure that MicroCode's target device selection is set to "16F1937". Most likely you're compiling a file that doesn't match the selected target device. This is a VERY common scenario. As you get more practice, you'll regularly check the Microcontroller to make sure that it is the correct one.

You can always unplug and the re-plug in the board if you find something isn't working properly. This will not harm the board in any way.

Cannot execute MPASMWIN (MPASMX) – Something went wrong in the installation, causing PBP to get disconnected from MPASM. Go to Start > All Programs > PBP3 from ME Labs and run the MPLAB-MPASM Setup utility. Use the browse button to locate a folder called MPASM Suite or MPASMX. This folder is usually found in C:\Program Files\Microchip or C:\Program Files(x86)\Microchip. Restart MicroCode Studio after finishing the setup utility.

Problem launching PBP – MicroCode has lost track of the compiler location. This indicates an installation problem that is probably related to access permissions on your computer. Contact support for this one.

Software froze at the programming step – Make sure the trainer board is connected at all times when compiling and programming. meProg needs to find the hardware immediately. If frozen, use your task manager to kill the meProg.exe application, connect the hardware, and try again.

Configuration programming error at 0001 / WARNING: Unable to open INCLUDE file 14-1_Init.pbp - When opening files in MicroCode Studio to use on the trainer board be sure that the "14-1_Init.pbp" file is in the same folder. All of the programs referenced in the tutorial utilize that file (`"14-1_Init.pbp)` and as long as it is in the same folder MicroCode Studio will be able to utilize it. If you create a new folder/location for files that you want to try to run, be sure that this file is in that same location.

Chapter 10: User Interface

10.1 Goals

The example program for this chapter is named "10-1_User_Interface.pbp". The program demonstrates how a real-world user interface is created and how it accepts control inputs and provides visual feedback. User-input is received from the capacitive touch pads while the LCD provides visual feedback to the user. The interface allows the user to control the color and brightness of the RGB LED.

Open the program in MicroCode Studio now, as the rest of this chapter will reference its component blocks of code. Program the Trainer to see it in action.

In this example, the following techniques are demonstrated:

- Previously written code snippets are copied from earlier programs.
- An array variable is used to store multiple settings that are selected with a "pointer" variable.
- Bit-aliases are used to give individual bits meaningful names.
- ARRAYWRITE command is used to fill multiple locations in an array variable.
- User-input (pad touches) are parsed and individual action taken for each pad.
- Multiple modes of display are used and controlled by a BIT variable.
- SELECT CASE is used to format display based on a numeric value.

10.2 Program Listing

Program: 10-2_User_Interface.pbp

```
' Variables used to control the user-interface screens
pointer VAR BYTE    ' numeric with range 0-2, tracks parameter under edit.
mode    VAR BIT     ' bit variable can be 0 or 1, indicates display mode.


''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Constants and variables for PWM

frequency  CON  7820    ' All channels will run at 7820 Hertz

red        CON  3     ' Red LED is connected to CCP3
green      CON  4     ' Green LED is connected to CCP4
blue       CON  5     ' Blue LED is connected to CCP5

duty       VAR  BYTE[3]    ' Create an array with 3 byte locations for duty values
red_duty   VAR  duty[0]    ' Alias the location of the Red duty-cycle
green_duty VAR  duty[1]    ' Alias the location of the Green duty-cycle
blue_duty  VAR  duty[2]    ' Alias the location of the Blue duty-cycle

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Constants, Variables, and settings for Capacitive Touch Sensor
touch_threshold CON 100         ' reading less than this value indicates a touch

channel         VAR BYTE        ' Specifies the virtual channel number
touch_states    VAR BYTE        ' Holds the state of each touch pad as a bit
CPS_value       VAR BYTE        ' Stores the timer value from each pad in turn

CPSCON0     = %10001101            ' Enable cap sense using Timer0
OPTION_REG = %11100011           ' Set Timer0 to clock source, prescaler 1:16

' Assign meaningful names to the individual touch pads
left            VAR touch_states.5
right           VAR touch_states.1
up              VAR touch_states.4
down            VAR touch_states.2
enter           VAR touch_states.3
```

```
escape          VAR touch_states.0
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

' Variables and settings for Timer-driven interrupt
DEFINE INTHAND  ticker  ' Declare the name of the Assembly Language ISR
tick1    VAR BYTE BANK0 ' Variable to hold the timer in

T1CON = %00110001     ' Enable Timer1 with prescale value 1:8
PIE1 = %00000001      ' Timer1 interrupt enabled
INTCON = %11000000    ' global & peripheral interrupts enabled


GOTO startup    ' Jump over Interrupt Service Routine (ISR) to startup label

ASM
ticker                  ; Begin the Interrupt Service Routine
    clrf    BSR         ; Point to BANK0
    bcf     T1CON, 0    ; Stop Timer1
    movlw   0xA7        ; Load W with 0xA7
    addwf   TMR1L, F    ; Add preset value to TMR1 LSB
    btfsc   STATUS, 2   ; If no byte overflow, skip next instruction
    incf    TMR1H, F    ; If byte overflow, increment MSB of TMR1
    movlw   0x3C        ; Load W with 0x3C
    addwf   TMR1H, F    ; Add preset value to TMR1 MSB
    bsf     T1CON, 0    ; Start Timer1
    bcf     PIR1, TMR1IF    ; clear Timer1 interrupt flag
    incf    _tick1, F    ; increment tick variable
    retfie  ; Return (using automatic context-restore)
ENDASM

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' The startup section is executed only once when the program starts.  It serves
' to initialize everything for a consistent start.
startup:
    PAUSE 200
    LCDOUT $FE, 1 ' Clear the LCD

    pointer = 0   ' To avoid erratic behavior, this value must be less than 3

    ' Don't continue until all the touch pads report an "untouched" state
    GOSUB wait_for_release

    ARRAYWRITE duty, [0,0,0]    ' Set RGB values to 0 (RGB LED off)
    GOSUB set_color             ' Update the state of the RGB LED
    mode = 1                    ' Set the display mode

main_loop:

    GOSUB ReadCPS   ' Update the touch_states variable, and in so doing update
                    ' the bits named left, right, up, down, enter, and escape.

    ' For each bit representing a touchpad, perform the appropriate action.
    ' Only one action will be performed, even if multiple pads are touched.

    IF left = 1 THEN            ' If the left pad is touched
        pointer = pointer - 1   ' decrement the pointer value
        ' If the value underflows to 255, force it to 2.  This yields
        ' a "circular" selection of the parameter to be edited.
        IF pointer = 255 THEN pointer = 2

    ELSEIF right = 1 THEN       ' If the right pad is touched
        pointer = (pointer + 1) ' Increment the pointer value
        ' When the pointer value reaches 3, reset it to zero. This yields
        ' a "circular" selection of the parameter to be edited.
        IF pointer > 2 THEN pointer = 0

    ELSEIF (up = 1) AND (mode = 0) THEN ' If the up pad is touched when editing
        ' Increment the value held in duty[pointer]
```

```
        duty[pointer] = duty[pointer] + 1
        GOSUB set_color     ' Update the PWM signals that drive the LED

    ELSEIF (down = 1) AND (mode = 0) THEN ' down pad is touched when editing
        ' Decrement the value held in duty[pointer]
        duty[pointer] = duty[pointer] - 1
        GOSUB set_color     ' Update the PWM signals that drive the LED

    ELSEIF enter = 1 THEN   ' If the enter (center) pad is touched
        LCDOUT $FE, 1       ' Clear the LCD
        mode = mode + 1     ' Toggle the mode flag

    ELSEIF escape = 1 THEN  ' If the escape pad is touched
        GOTO startup        ' Reset all and start over

    ELSE                    ' If no pads are touched
        tick1 = 0           ' Inhibit tick1 from counting up
    ENDIF

    ' At this point, the mode, pointer, and duty values are used to affect
    ' changes on the LCD.  The screen is completely redrawn, regardless of
    ' changes.

    IF mode = 1 THEN    ' If in display mode
        ' Print the names of colors on top line
        LCDOUT $FE, $80,   " RED GREEN BLUE"
        ' Print the duty-cycle values underneath each color's name
        LCDOUT $FE, $C1, DEC3 duty[0], "  ", DEC3 duty[1], "   ", DEC3 duty[2]

    ELSE                ' If in edit mode
        ' Print 1 of 3 headers on the first line of the LCD
        SELECT CASE pointer     ' Use pointer value to select the header
            CASE 0      ' If pointer = 0
                LCDOUT $FE, $80, "B <   RED   > G"
            CASE 1      ' If pointer = 1
                LCDOUT $FE, $80, "R <  GREEN  > B"
            CASE 2      ' If pointer = 2
                LCDOUT $FE, $80, "G <   BLUE  > R"
        END SELECT
        ' The value of duty[pointer] correspondes to the selected header above.
        ' Print the current value on line-2 of the display
        LCDOUT $FE, $C6, DEC3 duty[pointer]
    ENDIF

    ' Wait here until all pads report untouched.  This makes it easier on
    ' the user because the directional changes will only change once per touch.
    ' The "wait_for_release' subroutine, however, will allow a repeated value
    ' change if the touch is maintainted for a second or so...
    GOSUB wait_for_release

GOTO main_loop      ' Do it forever


'''''''''''''' Subroutines ''''''''''''''''''''''''''''''''''''''''''''''''''
DISABLE DEBUG   ' Debugging this routine wrecks the timing. Disable debugging
                ' in this section of code.  Debugging will resume when the
                ' program returns to the main loop.
ReadCPS:
touch_states = %00000000    ' clear states of touch pads in result variable
    FOR channel = 0 TO 5    ' loop through 6 channels
        ' Use LOOKUP to convert channel number to hardware (actual) channel.
        ' Place the actual channel-selection number in register CPSCON1
        LOOKUP channel, [14, 10, 13, 15, 12, 11], CPSCON1
        TMR0 = 0            ' Clear the Timer0 count value
        PAUSE 10            ' Count for 10ms
        CPS_value = TMR0    ' Record the Timer0 count value
        IF CPS_value < touch_threshold THEN  ' If this channel's reading is less than
                            threshold...
            touch_states.0[channel] = 1    ' Set this channel's state bit to indicate touch
```

```
        ENDIF
'        GOSUB display_readings  ' Call a sub to display actual readings.  Used to set
                                   threshold.
    NEXT channel
RETURN

ENABLE DEBUG

wait_for_release:   ' Used to control auto-repeat on the touch pads
    ' Wait in the following loop until all touchpads report untouched
    DO UNTIL touch_states = %00000000
        IF tick1 > 12 THEN  ' If the tick1 variable exceeds 12
            tick1 = 13      ' Don't let it count beyond 13
            EXIT            ' Skip the rest of the DO..LOOP
        ENDIF
        GOSUB ReadCPS       ' Check the pads for touch
    LOOP
RETURN

set_color:  ' Update the PWM channels that drive the LEDs
  HPWM red, red_duty, frequency         ' Output PWM for Red LED
  HPWM green, green_duty, frequency     ' Output PWM for Green LED
  HPWM blue, blue_duty, frequency       ' Output PWM for Blue LED
RETURN
```

10.3 Re-using program code

Most software engineers try to use previously-written code whenever possible. It saves time and yields fewer opportunities for error. The following code has been explained in-detail in the chapters where it was originally presented, so I will only cover the changes and new concepts here.

10.3.1 Reused Code: PWM output to drive the RGB LED

In Chapter 6 we learned how to drive the individual RGB LEDs to achieve a variable color output. The user interface in this chapter will also drive a variable color output, so I've copied parts of the code that was presented in Chapter 6.

```
frequency   CON  7820    ' All channels will run at 7820 Hertz

red         CON  3     ' Red LED is connected to CCP3
green       CON  4     ' Green LED is connected to CCP4
blue        CON  5     ' Blue LED is connected to CCP5

duty        VAR  BYTE[3]    ' Create an array with 3 byte locations for duty values
red_duty    VAR  duty[0]    ' Alias the location of the Red duty-cycle
green_duty  VAR  duty[1]    ' Alias the location of the Green duty-cycle
blue_duty   VAR  duty[2]    ' Alias the location of the Blue duty-cycle
```

In the declarations above, the type of declaration for the duty variables has changed from the original chapter-6 example. Instead of the individual duty variables from Chapter 6, we've created an array variable called "duty" with 3 elements, then created aliases for each of the locations in the array variable. This allows the previous variable names (red_duty, green_duty, and blue_duty) to be used as before, but It also allows the program to easily select between these variables based on the index value 0,1, or 2. (duty[0] is Red, duty[1] is Green, and duty[2] is Blue.)

The output to the RGB LED is updated using the following subroutine. The commands were copied from Chapter 6, then placed in a subroutine.

```
set_color:  ' Update the PWM channels that drive the LEDs
```

```
  HPWM red, red_duty, frequency         ' Output PWM for Red LED
  HPWM green, green_duty, frequency     ' Output PWM for Green LED
  HPWM blue, blue_duty, frequency       ' Output PWM for Blue LED
RETURN
```

The set_color subroutine will be called whenever the output color needs to be updated with new values.

10.3.2 Reused Code: reading touch pad states

Since the user interface of this chapter needs to read the touch pads, the variable declarations and initialization are copied from the example program in Chapter 8:

```
touch_threshold CON 100         ' reading less than this value indicates a touch

channel         VAR BYTE        ' Specifies the virtual channel number
touch_states    VAR BYTE        ' Holds the state of each touch pad as a bit
CPS_value       VAR BYTE        ' Stores the timer value from each pad in turn

CPSCON0    = %10001101             ' Enable cap sense using Timer0
OPTION_REG = %11100011           ' Set Timer0 to clock source, prescaler 1:16

' Assign meaningful names to the individual touch pads
left            VAR touch_states.5
right           VAR touch_states.1
up              VAR touch_states.4
down            VAR touch_states.2
enter           VAR touch_states.3
escape          VAR touch_states.0
```

Aliases were added to give each touchpad a meaningful name. This will help when writing the code that acts upon the user's touch inputs.

I've also copied the subroutine from Chapter 8 that reads the touchpads. This will appear at the end of our user interface example:

```
ReadCPS:
touch_states = %00000000    ' clear states of touch pads in result variable
    FOR channel = 0 TO 5    ' loop through 6 channels
        ' Use LOOKUP to convert channel number to hardware (actual) channel.
        ' Place the actual channel-selection number in register CPSCON1
        LOOKUP channel, [14, 10, 13, 15, 12, 11], CPSCON1
        TMR0 = 0              ' Clear the Timer0 count value
        PAUSE 10              ' Count for 10ms
        CPS_value = TMR0      ' Record the Timer0 count value
        IF CPS_value < touch_threshold THEN  ' If this channel's reading is less than
                              threshold...
            touch_states.0[channel] = 1    ' Set this channel's state bit to indicate touch
        ENDIF
'        GOSUB display_readings  ' Call a sub to display actual readings.  Used to set
                              threshold.
    NEXT channel
RETURN
```

The ReadCPS subroutine will be called whenever the touchpads need to be checked for touch. Note that the GOSUB that called the routine display_readings is commented because this function won't be required in this example. Commenting code instead of deleting it can be advantageous if you find you need the code later.

10.3.3 Reused Code: Timer1 interrupt

Chapter 9 presented a timer-based, Assembly-language interrupt that was used to track how long a button was pressed. The user interface in this chapter will use the same timer function to allow an auto-repeat function on the touchpads. The following code, described in Chapter 9, is added to this chapter's example:

```
' Variables and settings for Timer-driven interrupt
DEFINE INTHAND  ticker  ' Declare the name of the Assembly Language ISR
tick1    VAR BYTE BANK0 ' Variable to hold the timer in

T1CON = %00110001      ' Enable Timer1 with prescale value 1:8
PIE1 = %00000001       ' Timer1 interrupt enabled
INTCON = %11000000     ' global & peripheral interrupts enabled


GOTO startup    ' Jump over Interrupt Service Routine (ISR) to startup label

ASM
ticker                  ; Begin the Interrupt Service Routine
    clrf    BSR         ; Point to BANK0
    bcf     T1CON, 0    ; Stop Timer1
    movlw   0xA7        ; Load W with 0xA7
    addwf   TMR1L, F    ; Add preset value to TMR1 LSB
    btfsc   STATUS, 2   ; If no byte overflow, skip next instruction
    incf    TMR1H, F    ; If byte overflow, increment MSB of TMR1
    movlw   0x3C        ; Load W with 0x3C
    addwf   TMR1H, F    ; Add preset value to TMR1 MSB
    bsf     T1CON, 0    ; Start Timer1
    bcf     PIR1, TMR1IF    ; clear Timer1 interrupt flag
    incf    _tick1, F    ; increment tick variable
    retfie  ; Return (using automatic context-restore)
ENDASM
```

The code above functions in a self-contained fashion. It will maintain the value in the "tick1" variable without our manual intervention. Note that instead of jumping to the main_loop label to get over the ISR, the code jumps to the startup label.

10.4 Code: startup

To insure that the program behaves properly when started, it needs some code that only executes once and is not repeated within the main loop.

```
' The startup section is executed only once when the program starts.  It serves
' to initialize everything for a consistent start.
startup:
    PAUSE 200
    LCDOUT $FE, 1 ' Clear the LCD

    pointer = 0   ' To avoid erratic behavior, this value must be less than 3

    ' Don't continue until all the touch pads report an "untouched" state
    GOSUB wait_for_release

    ARRAYWRITE duty, [0,0,0]     ' Set RGB values to 0 (RGB LED off)
    GOSUB set_color              ' Update the state of the RGB LED
    mode = 1                     ' Set the display mode
```

The 200mS pause gives the LCD time to initialize before the program clears it.

The variable "pointer" will determine which color (red, green, or blue) is being edited or written to the LCD. Since there are only three colors, any value in pointer that is not 0, 1, or 2 will cause unexpected results. Therefore, this variable must be initialized when the program starts.

Mostly as a precaution, the program will wait until the touchpads are idle (untouched) before execution is allowed to continue.

The ARRAYWRITE command sets the Red, Green, and Blue levels to values of zero. This command is used to write a list of values into sequential elements of the "duty" array variable. The line "ARRAYWRITE duty, [0,0,0]" finds the array variable named "duty" and writes 0 to the first element, 0 to the second, and 0 to the third. This could also be written in three lines:

```
duty[0] = 0
duty[1] = 0
duty[2] = 0
```

After the values for color levels are written to the duty array, the subroutine "set_color" must be called to update the PWM output signals.

The variable "mode" controls the information on the display. Mode-1 is a read-only display showing all three color values at once. Touching the center pad changes to mode-0, which allows the user to select one of three values and adjust it.

10.5 Code: the main loop

10.5.1 Code: check for user input

The first step is to read the touch pads and record which pads are being touched.

```
main_loop:

    GOSUB ReadCPS    ' Update the touch_states variable, and in so doing update
                     ' the bits named left, right, up, down, enter, and escape.
```

10.5.2 Parse user input and take actions

At this point, the program knows what pad(s) the user is touching because the touch_states variable is updated by the ReadCPS subroutine. The goal of this section of code is to take action according to the user's input. All actions take the form of changes to variables and calling the subroutine that updates the PWM outputs. The display will be updated in the next section of the program. Handling the actions and the display separately help simplify the logic of the program.

Here is where the alias names for each touchpad come into play. Because the line "left VAR touch_states.5" already exists in the program, I can write things like "IF left = 1 THEN". Without the alias, I would have to write something like "IF touch_states.5 = 1 THEN". The alias names make it much easier to see, at a glance, which touchpad the code is responding to.

Note that only one touchpad will get a response for each trip through the following code. This means that if two pads are touched at the same time, one will be ignored. Each IF or ELSEIF checks for a touch on an individual pad. The first one that is found to be touched is the only one that gets a response. In this way, the priority of response is determined by which pads we check first. We've chosen –quite arbitrarily –to give the left/right pads the highest priority.

```
    IF left = 1 THEN             ' If the left pad is touched
        pointer = pointer - 1    ' decrement the pointer value
        ' If the value underflows to 255, force it to 2.  This yields
        ' a "circular" selection of the parameter to be edited.
        IF pointer = 255 THEN pointer = 2

    ELSEIF right = 1 THEN        ' If the right pad is touched
        pointer = (pointer + 1) ' Increment the pointer value
        ' When the pointer value reaches 3, reset it to zero. This yields
        ' a "circular" selection of the parameter to be edited.
        IF pointer > 2 THEN pointer = 0
```

The left and right pads change the variable "pointer". A touch to the left pad decreases the pointer value. A touch to the right increases the pointer value. The value of the pointer will determine which color is displayed and edited when in mode-0 in the code to follow.

In each case, an additional line of code is added to give the left/right selection a "circular" behavior. That is to say, the user may repeatedly press the left or the right pad alone and scroll continuously through all the colors available. In order to accomplish this, the pointer variable is forced to its lowest value when it exceeds its highest, and forced to its highest value when it drops below its lowest. While it is imperative that the value isn't allowed to stay outside of its valid range (0-2), it is an arbitrary choice whether the user sees it stop at the end of its range or loop around to the other end of its range.

The limiting code in the action for the right pad is straightforward:

```
        IF pointer > 2 THEN pointer = 0
```

If the value exceeds 2, force it back to 0. When the user repeatedly touches right, the pointer follows the sequence 0, 1, 2, 0, 1, 2, 0, … until the user stops touching.

The limiting code for the left action does the same thing in reverse.

```
        IF pointer = 255 THEN pointer = 2
```

Note that that pointer is a byte variable and when it is at 0 and 1 is subtracted, the result is 255. This is because PBP (in the compile mode we are using) does not recognize signed numbers in comparisons. (Writing "If pointer < 0" doesn't work.)

Notice that in the first case, a greater-than comparison is used in "IF pointer > 2" and in the second an equal-to comparison is used in "IF pointer = 255". Why write "IF pointer > 2" instead of "IF pointer = 3"? Both statements produce the same result, but the greater-than comparison provides a self-correcting function. Precautions have been taken to restrict the value of pointer to a range of 0-2. But, what if something unforeseen happens and it arrives at this point in the code with a value of 5?

The answer lies in human behavior when interacting with machines. If anything unexpected is encountered by the user, the first response is to push more buttons. I can assure you that if the user sees a scrambled display when he presses left, he will press right to try to "fix" it. When right is touched, any aberrant value will immediately be replaced with a value of 0. It is unlikely that this will be needed, but why pass up the chance to handle the unexpected? Practices like this make for reliable products and demonstrate higher level programming.

> Always try to gracefully handle unexpected conditions, especially when your program must interact with a human.

Next in the code, the up/down touches are handled.

```
ELSEIF (up = 1) AND (mode = 0) THEN ' If the up pad is touched when editing
    'Increment the value held in duty[pointer]
    duty[pointer] = duty[pointer] + 1
    GOSUB set_color     ' Update the PWM signals that drive the LED

ELSEIF (down = 1) AND (mode = 0) THEN ' down pad is touched when editing
    ' Decrement the value held in duty[pointer]
    duty[pointer] = duty[pointer] - 1
    GOSUB set_color     ' Update the PWM signals that drive the LED
```

Note that these actions are dependent on two comparisons instead of one "(up = 1) AND (mode = 0)". Not only do the up or down pads need to be touched, but the mode must be 0 also. Since the up/down touches adjust the values of individual colors, their action is limited to mode-0 so they won't work unless the user has selected the edit-mode of operation. This is not strictly necessary, but it makes for more predictable operation. Were it not written this way, the colors could be selected and adjusted without any indication on the LCD. You are welcome to modify the code and try it for the sake of entertainment.

The actions are simple for these pads. Touch up and the duty value is incremented. Touch down and the duty value is decremented. The PWM outputs are updated each time the value is changed. There are no limits placed on the range of values held in the duty variable. If it exceeds 255, it is allowed to roll over to 0. This is the nature of a BYTE variable.

The new concept here is the use of "duty[pointer]". The variable "duty" is declared as a byte-array with three elements. In essence, this creates three duty variables that can be accessed separately with a number. In this case, the element of "duty" that is used is determined by the value of "pointer". If pointer=0, then duty[0] is used, if pointer=1, then duty[1] is used, and so forth.

This method greatly simplifies the written code. It allows the left/right pads to change the variable that is edited by the up/down pads, without the necessity of comparisons and multiple-case redundant code. The value of the pointer variable becomes essential as the decision maker when adjusting values and deciding what the display should show.

We've chosen to call the center pad "enter".

```
ELSEIF enter = 1 THEN    ' If the enter (center) pad is touched
        LCDOUT $FE, 1        ' Clear the LCD
        mode = mode + 1      ' Toggle the mode flag
```

The action, when enter is touched, is to clear the LCD and toggle the mode. Since mode is a BIT variable, it can only store values 0 or 1. When the code adds one to mode, the result is always that mode becomes the opposite of what it was (0+1=1, 1+1=0). When the mode changes, the LCD is rewritten later in the program. To insure that there are no left-over characters on the display, the LCD is cleared here. The command to clear the display takes a bit of time, so it can be advantageious to do this as infrequently as possible. The LCDOUT commands found later in the code are repeated very rapidly, so a clear at that point would affect the apparent contrast of the display.

We've chosen to call the lower-right, square pad "escape".

```
ELSEIF escape = 1 THEN  ' If the escape pad is touched
    GOTO startup        ' Reset all and start over
```

For the escape touch, the program simply jumps back to the label "startup". To the user's eyes, this resets everything.

We've handled all the touches possible, but not the case when no pads are touched. The program uses the catch-all ELSE keyword to trigger this code, as opposed to a specific test for no pads touched. As mentioned before, the thinking is that this might catch an unexpected condition and make the program a bit more reliable.

```
    ELSE                    ' If no pads are touched
        tick1 = 0           ' Inhibit tick1 from counting up
    ENDIF
```

The only action if no pads are touched is to force the tick1 variable to a value of zero. What does this accomplish? Nothing, really, except that when any pad *is* touched the variable is *not* forced to 0. Consequentially, when any pad is touched continuously, the tick1 variable begins to count up. This allows for an auto-repeat function using a few lines of code inside the subroutine "wait_for_release".

10.6 Code: LCD for feedback

At this point in the program, the pointer and mode variables tell us what the LCD should show to the user. If the mode is 1, the LCD should show the user all three color values.

```
    IF mode = 1 THEN      ' If in display mode
        ' Print the names of colors on top line
        LCDOUT $FE, $80,   " RED GREEN BLUE"
        ' Print the duty-cycle values underneath each color's name
        LCDOUT $FE, $C1, DEC3 duty[0], "   ", DEC3 duty[1], "    ", DEC3 duty[2]
```

When the user changes the mode to 0, the pointer variable is used to determine which color value is being edited. The display will change appropriately.

```
ELSE                ' If in edit mode
        ' Print 1 of 3 headers on the first line of the LCD
        SELECT CASE pointer     ' Use pointer value to select the header
            CASE 0      ' If pointer = 0
                LCDOUT $FE, $80, "B <    RED    > G"
            CASE 1      ' If pointer = 1
                LCDOUT $FE, $80, "R <  GREEN    > B"
            CASE 2      ' If pointer = 2
                LCDOUT $FE, $80, "G <   BLUE    > R"
        END SELECT
        ' The value of duty[pointer] correspondes to the selected header above.
        ' Print the current value on line-2 of the display
        LCDOUT $FE, $C6, DEC3 duty[pointer]
    ENDIF
```

For the first line of the LCD, a SELECT CASE block is used to pick one of three display lines. When I write "SELECT CASE pointer", PBP sets up a multiple-choice environment that uses the value of pointer to execute one of many blocks of code. This is similar to the IF/ELSEIF syntax, except that SELECT CASE is intended to be based on the value of a single variable or expression.

The subsequent line "CASE 0" is equivalent to "ELSEIF pointer = 0" in an IF/ELSEIF scenario. The difference is that the CASE statement doesn't allow a compound-condition that checks more than one thing. It is invalid to write "CASE (pointer = 0) AND (mode = 0)". This limitation is exchanged for faster execution time and a bit more readability in the code.

In this SELECT CASE, one of three LCDOUT commands is executed based on the value of the pointer variable. For the color selected, its name is displayed in the center of line-1 on the display. To the right and left, single-letter clues are provided for the user so that he can anticipate what will happen when the left/right pads are touched.

On the second line of the display, the current value of the duty variable *for the selected color* is displayed. The pointer variable is used as the index value to specify which element of the duty variable is used ("duty[pointer]"). The LCD command string "$FE, $C6" locates the text on the sixth character of the second line.

10.7 Code: pace and auto-repeat

The last line of code before repeating the main loop calls the "wait_for_release" subroutine.

```
    GOSUB wait_for_release

GOTO main_loop      ' Do it forever
```

The main purpose of the subroutine is to hold program execution until the user lifts his finger from the touchpad. This stops actions from being repeated when the user's finger lingers on the pad. It would be inconsiderate to force the user to be agile and quick in order to use the interface. When we require that the user touch and remove a finger in order to make something happen, it assures that nothing will happen unintentionally.

To accomplish its fundamental function, the routine only needs to check the variable "touch_states", update it by calling the "ReadCPS" routine, and repeat this until no touches are detected. This was accomplished originally with a subroutine that looked like this:

```
wait_for_release:   ' Used to control auto-repeat on the touch pads
' Wait in the following loop until all touchpads report untouched
    DO UNTIL touch_states = %00000000
        GOSUB ReadCPS       ' Check the pads for touch
    LOOP
RETURN
```

Having written this, it became apparent that without this subroutine, all the pads repeated all the time. Since the subroutine existed to prevent this behavior, it could be selectively disabled to allow auto-repeat. To accomplish this, code was added to check the tick1 variable and abandon the DO..LOOP when tick1 reached a certain time value.

The final version of the routine looks like this:

```
' Wait in the following loop until all touchpads report untouched
    DO UNTIL touch_states = %00000000
        IF tick1 > 12 THEN  ' If the tick1 variable exceeds 12
            tick1 = 13      ' Don't let it count beyond 13
            EXIT            ' Skip the rest of the DO..LOOP
        ENDIF
        GOSUB ReadCPS       ' Check the pads for touch
    LOOP
RETURN
```

The auto-repeat code is embedded inside the DO loop. When the tick1 variable exceeds its threshold, the EXIT command is executed. EXIT causes the program to abandon the DO loop immediately. It then jumps to the line that follows LOOP, which causes it to return to the main program.

The result is that the subroutine is effectively disabled when the tick1 variable reaches the specified count. The program is then allowed to repeatedly process a continuous touch. The auto-repeat rate is simply the execution time of the main loop. Since the main loop calls the ReadCPS routine every time it executes, the auto-repeat happens once every 60mS or so.

10.8 Summary

- Sections of code are reused from Chapters 6, 8 and 9 to control the PWM output, read the touchpads, and keep time using an Interrupt.
- User input is parsed and actions are performed based on the pads that are touched.
- The LCD is manipulated based on changing variables to provide feedback to the user.

- An array variable is used to store parameters that can be selected with a number or a value in a pointer variable.

10.9 Exercises

1. Slow the existing auto-repeat rate. (Solution: 10-1_Slower_Autorepeat.pbp)
2. Change the left/right parameter selection from its "circular" operation so that it stops acting when the user reaches Blue (to the right) and Red (to the left). Force the user to reverse direction to return to Green (in the center). (Solution: 10-2_Non_Circular.pbp)
3. Instead of changing the output constantly as the RGB parameters are changed, only change the output when the user returns to mode-1 (presses ENTER). (Solution: 10-3_Delayed_Color.pbp)