Development Kit For the PIC® MCU

Exercise Book

Capacitive Touch

March 2010

Custom Computer Services, Inc.
Brookfield, Wisconsin, USA
262-522-6500

Recognized Microchip Third-Party Tool Provider

Custom Computer Services, Inc. proudly supports the Microchip brand with highly optimized C compilers and embedded software development tools.

1 UNPACKING AND INSTALLATION

Inventory

- ❑ Use of this kit requires a PC with Windows 95, 98, ME, NT, 2000 or XP. The PC must have a spare 9-Pin Serial or USB port, a CD-ROM drive and 75 MB of disk space.
- ❑ The diagram on the following page shows each component in the Capacitive Touch kit. Ensure every item is present.

Software

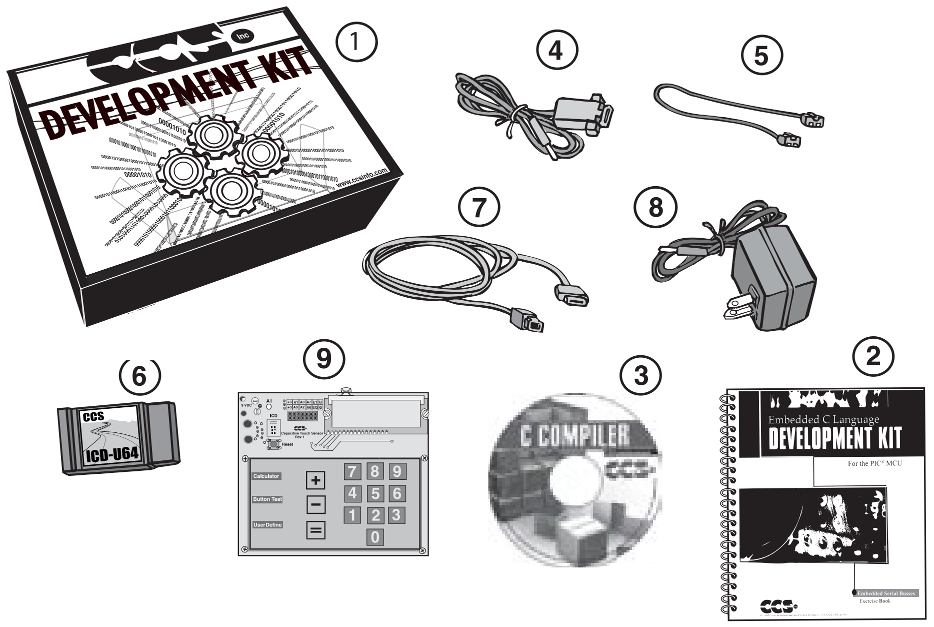
- ❑ Insert the CD into the computer and wait for the installation program to start. If your computer is not set up to auto-run CDs, then select **My Computer** and double click on the CD drive.
- ❑ Click on **Install** and use the default settings for all subsequent prompts by clicking NEXT, OK, CONTINUE…as required.
- ❑ Identify a directory to be used for the programs in this booklet. The install program will have created an empty directory **c:\program files\picc\projects** that may be used for this purpose.
- ❑ Select the compiler icon on the desktop. In the PCW IDE, click **Help>About** and verify a version number is shown for the IDE and PCM to ensure the software was installed properly. Exit the software.

Hardware

- ❑ Connect the PC to the ICD(6) using the USB cable.(1) Connect the prototyping board (10) to the ICD using the modular cable. Plug in the AC adaptor (9) to the power socket and plug it into the prototyping board (10). The first time the ICD-U is connected to the PC, Windows will detect new hardware. Install the USB driver from the CD or website using the new hardware wizard. The driver needs to be installed properly before the device can be used.
- ❑ The LED should be red(2) on the ICD-U to indicate the unit is connected properly.
- ❑ Run the Programmer Control Software by clicking on the CCSLOAD icon on the desktop. Use CCSLOAD Help File for assistance.
- ❑ The software will auto-detect the programmer and target board and the LED should be illuminated green. If any errors are detected, go to Diagnostic tab. If all tests pass, the hardware is installed properly.
- ❑ Disconnect the hardware until you are ready for Chapter 3. Always disconnect the power to the Prototyping board before connecting/disconnecting the ICD or changing the jumper wires to the Prototyping board.

(1) ICS-S40 can also be used in place of ICD-U. Connect it to an available serial port on the PC using the 9 pin serial cable. There is no driver required for S40.

(2) ICD-U40 units will be dimly illuminated green and may blink while connecting.



1. Storage box
2. Exercise booklet
3. CD-ROM of C compiler (optional)
4. Serial PC to Prototyping board cable
5. Modular ICD to Prototyping board cable
6. ICD unit for programming and debugging
7. USB (or Serial) PC to ICD cable
8. AC Adaptor (9VDC)
9. Prototyping board with a PIC16LF727 microcontroller
   (See inside front and back cover for details on the board layout and schematic)

2 USING THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Editor

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>Open>Source File**. Select the file: **c:\program files\picc\examples\ex_stwt.c**
- ❑ Scroll down to the bottom of this file. Notice the editor shows comments, preprocessor directives and C keywords in different colors.
- ❑ Move the cursor over the **Set_timer0** and click. Press the F1 key. Notice a Help file description for set_timer0 appears. The cursor may be placed on any keyword or built-in function and F1 will find help for the item.
- ❑ Review the editor special functions by clicking on **Edit**. The IDE allows various standard cut, paste and copy functions.
- ❑ Review the editor option settings by clicking on **Options>Editor Properties**. The IDE allows selection of the tab size, editor colors, fonts, and many more. Click on **Options>Toolbar** to select which icons appear on the toolbars.

Compiler

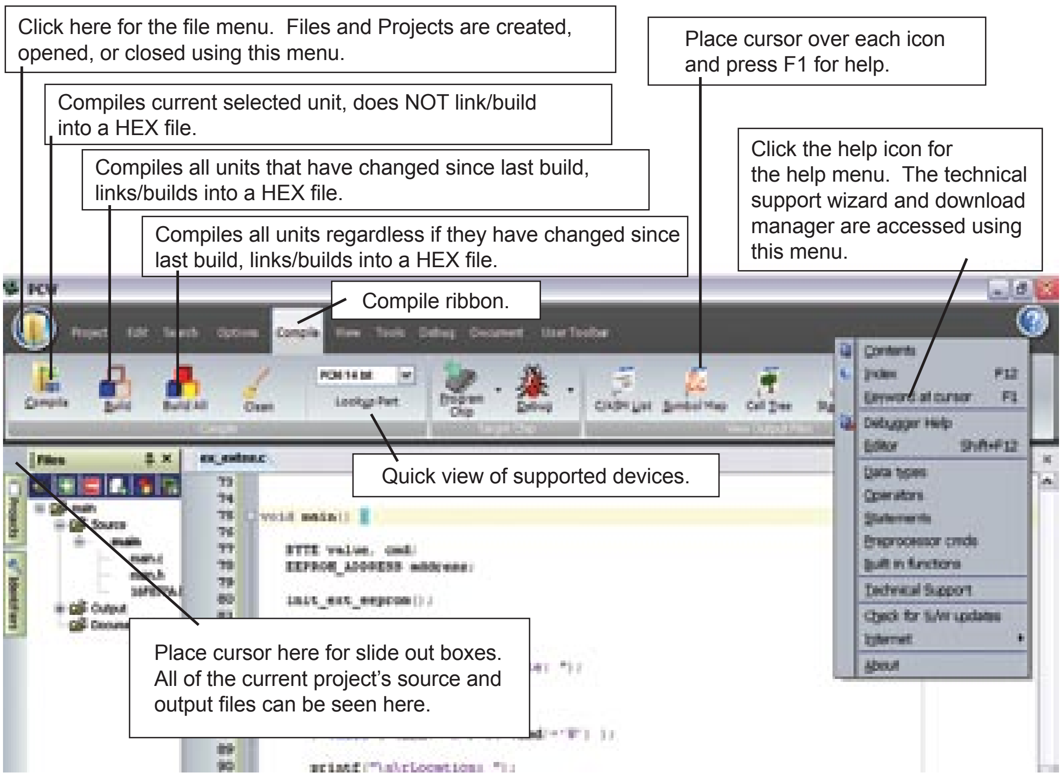
- ❑ Use the drop-down box under Compile to select the compiler. CCS offers different compilers for each family of Microchip parts. All the exercises in this booklet are for the PIC16LF727 chip, a 16-bit opcode part. Make sure **PCM 14-bit** is selected in the drop-down box under the **Compiler** tab.
- ❑ The main program compiled is always shown in the bottom of the IDE. If this is not the file you want to compile, then click on the tab of the file you want to compile. Right click into editor and select **Make file project**.
- ❑ Click **Options>Project Options>Include Files…** and review the list of directories the compiler uses to search for included files. The install program should have put two directories in this list: devices and drivers.
- ❑ Normally the file formats need not be changed and global defines are not used in these exercises. To review these settings, click **Options>Project Options>Output Files** and **Options>Project Options>Global Defines**.
- ❑ Click the compile icon to compile. Notice the compilation box shows the files created and the amount of ROM and RAM used by this program. Press any key to remove the compilation box.

Viewer

- ❑ Click **Compile>Symbol Map**. This file shows how the RAM in the microcontroller is used. Identifiers that start with @ are compiler generated variables. Notice some locations are used by more than one item. This is because those variables are not active at the same time.
- ❑ Click **Compile>C/ASM List.** This file shows the original C code and the assembly code generated for the C. Scroll down to the line:

```
int_count=INTS_PER_SECOND;
```

- ❑ Notice there are two assembly instructions generated. The first loads 4C into the W register. INTS_PER_SECOND is #defined in the file to 76. 4C hex is 76 decimal. The second instruction moves W into memory. Switch to the Symbol Map to find the memory location where int_count is located.
- ❑ Click **View>Data Sheet**, then **View.** This brings up the Microchip data sheet for the microprocessor being used in the current project.


Click here for the file menu. Files and Projects are created, opened, or closed using this menu.

Compiles current selected unit, does NOT link/build into a HEX file.

Compiles all units that have changed since last build, links/builds into a HEX file.

Compiles all units regardless if they have changed since last build, links/builds into a HEX file.

Compile ribbon.

Place cursor over each icon and press F1 for help.

Click the help icon for the help menu. The technical support wizard and download manager are accessed using this menu.

Quick view of supported devices.

Place cursor here for slide out boxes. All of the current project's source and output files can be seen here.

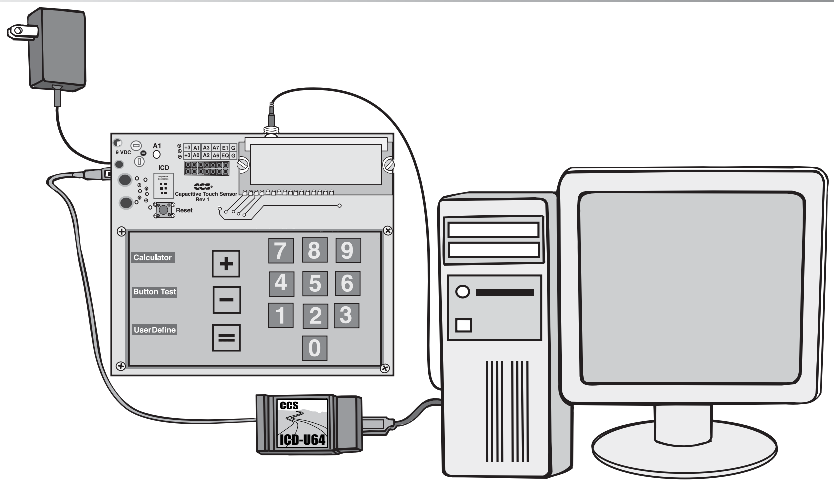
3 COMPILING AND RUNNING A PROGRAM

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>New>Source File** and enter the filename **EX3.C**
- ❑ Type in the following program and **Compile.**

```
#include <16lf727.h>
#device ICD=TRUE
#fuses HS,NOWDT
#use delay(INTERNAL=8mhz)

#define GREEN_LED PIN_A1

void main() {
    while (TRUE) {
       output_low (GREEN_LED);
       delay_ms (1000);
       output_high (GREEN_LED);
       delay_ms (1000);
    }
}
```

NOTES

- The first four lines of this program define the basic hardware environment. The chip being used is the PIC16LF727, running at 8MHz with the ICD debugger.
- The #define is used to enhance readability by referring to GREEN_LED in the program instead of PIN_A5.
- The “while (TRUE)” is a simple way to create a loop that never stops.
- Note that the “output_low” turns the LED on because the other end of the LED is +5V. This is done because the chip can tolerate more current when a pin is low than when it is high.
- The “delay_ms(1000)” is a one second delay (1000 milliseconds).



- ❑ Connect the ICD to the Prototyping board using the modular cable, and connect the ICD to the PC. Power up the Prototyping board.
- ❑ Click **Debug>Enable Debugger** and wait for the program to load.
- ❑ If you are using the ICD-U40 and the debugger cannot communicate to the ICD unit go to the debug configure tab and make sure ICD-USB from the list box is selected.
- ❑ Click the green go icon:
- ❑ Expect the debugger window status block to turn yellow indicating the program is running.
- ❑ The green LED on the Prototyping board should be flashing. One second on and one second off.
- ❑ The program can be stopped by clicking on the stop icon:

FURTHER STUDY

A *Modify the program to turn the green LED on for 5 seconds, then off for 5 seconds.*
B *Add to the program a #define macro called "delay_seconds" so the delay_ms(1000) can be replaced with : delay_seconds(1); and delay_ms(5000) can be: delay_seconds(5);.*

Note: *Name these new programs EX3A.c and EX3B.c and follow the same naming convention throughout this booklet.*

4 PROGRAM STRUCTURE

- ❑ It is good practice to put all the hardware definitions for a given design into a common file that can be reused by all programs for that board. Open **EX3.C** and drag the cursor over (highlight) the first 6 lines of the file. Click **Edit>Paste** to file and give it the name **prototype.h.**
- ❑ It is also helpful to collect a library of utility functions to use as needed for future programs. Note that just because a function is part of a program does not mean it takes up memory. The compiler deletes functions that are not used. Create a new file **utility.c** and the following new function to the file:

```
void blink_led_once() {
       output_low (GREEN_LED);
       delay_ms (1000);
       output_high (GREEN_LED);
       delay_ms (1000);
}
```

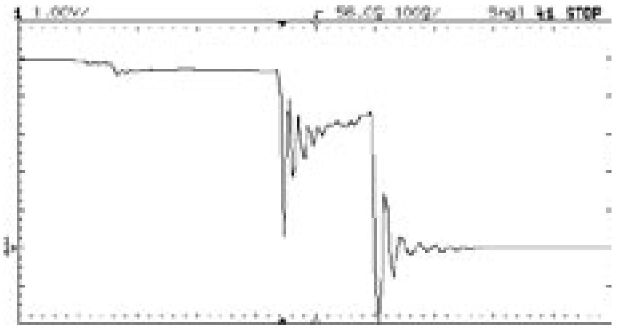
- ❑ Close all files and click **File>New>Source File** and enter the filename **EX4.c.**
- ❑ Type in the following program.

```
#include <prototype.h>
#include <utility.c>

void main() {
       while(TRUE) {
              blink_led_once();
       }
}
```

- ❑ **Click on Compile** and Run the program. Check that the LED blinks the same as in Chapter 3 program **EX3.c.**

NOTES

- The Prototyping board has one momentary pushbutton that may be used as an input to the program. The input pin is connected to a 10K pull-up resistor to +5V. The button, when pressed, shorts the input pin to ground. The pin is normally high while in this configuration, but it is low while the button is pressed.
- This program shows how to use simple C functions. The function wait_for_one_press() will first get stuck in a loop while the input pin is high (not pressed). It then waits in another loop while the pin is low. The function returns as soon as the pin goes high again. Note that the loops, since they do not do anything while waiting, do not look like much-they are a simple ; (do nothing).
- When the button is pressed once, it is common for several very quick connect disconnect cycles to occur. This can cause the LEDs to advance more than once for each press. A simple debounce algorithm can fix the problem. Add the following line between the two while loops: delay_ms(100); The following scope picture of a button press depicts the problem:



FURTHER STUDY

A *Modify the function in* **utility.c** *so that the LED is on for twice as long as it is off.*
B *Create your own blink routine function in* **utility.c** *and test on the prototyping board. Remember that "output_low" turns the LED on, not off.*

5 DEBUGGING

- ❑ Open **EX4.C** and start the debugger **Debug>Enable Debugger**.
- ❑ Click the reset icon to ensure the target is ready.
- ❑ Click the step-over icon twice. This is the step over command. Each click causes a line of C code to be executed. The highlighted line has not been executed, but the line about to be executed.
- ❑ Step over the `blink _ LED _ once();` line and notice that one click executed the entire function. This is the way step over works
- ❑ Step over until the call to `blink _ LED _ once();` is highlighted. This time, instead of step over, use the standard step icon several times and notice the debugger is now stepping into the function.
- ❑ Click the GO icon to allow the program to run and verify that the program is running normally. Click the stop icon to halt execution. Notice the C source line that the program stopped on.
- ❑ In the editor, click on `blink_LED_once();` to move the editor cursor to that line. Then click the **Breaks** tab and click the add icon to set a breakpoint. The debugger will now stop every time that line is reached in the code. Click the GO icon. The debugger should now stop on the breakpoint. Repeat this a couple of times to see how the breakpoint works. Note that the ICD with PIC16 chips only allow one breakpoint at a time.
- ❑ Click **Compile>C/ASM list**. Scroll down to the highlighted line. Notice that one assembly instruction was already executed for the next line. This is another side effect of the ICD debugger. Sometimes breakpoints slip by one ASM instruction.
- ❑ Click the step over icon a few times and note that when the list file is the selected window, the debugger has executed one assembly instruction per click instead of one entire C line.
- ❑ Close all files and click **File>New>Source File** and enter filename **EX5.c**.
- ❑ Type in the following program.

```
#include <prototype.h>
#include <utility.c>

void main() {
       int a,b,c;

       a=11;
       b=5;
       c=a+b;
       c=b-a;
       while(TRUE);
}
```

- ❑ **Compile** the program and step-over until the c=a+b line is executed. Click the **Watch** tab, then the add icon to add a watch. Enter **c** or choose **c** from the variables from list, then click **Add Watch**. The expected value is 16.
- ❑ Step-over the subtraction and notice the value of c. The int data type by default is not signed, so c cannot be the expected -6. The modular arithmetic works like a car odometer when the car is in reverse only in binary. For example, 00000001 minus 1 is 00000000, subtract another 1 and you get 11111111.
- ❑ Reset and again step up to the c=a+b. Click the Eval tab. This pane allows a one time expression evaluation. Type in a_b and click Eval to see the debugger and calculate the result. The complete expression may also be put in the watch pane as well. Now enter b=10 and click Eval. This expression will actually change the value of B if the “keep side effects” check box of the evaluation tab is checked. Check it and click Eval again. Step over the addition line and click the Watch tab to observe the c value was calculated with the new value of b.

FURTHER STUDY

A *Modify the program to include the following C operators to see how they work:*

* / % & ^

Then, with b=2 try these operators: >> <<
Finally, try the unary complement operator with c=~a;

A *Design a program to test the results of the relational operators*

< > == !=

by exercising them with b as 10, 11, and 12.
Then, try the logical operators || and && with the four combinations of a=0,1 and b=0,1. Finally, try the unary not operator with: c=!a; when a is 0 and 1.

6 INTRODUCTION TO THE LCD

The Prototyping board included in this Development Kit has an external Liquid Crystal Display (LCD) that can be used to display a large set of characters and numbers in order to assist in the debugging of software. The LCD can also act as an inexpensive program monitoring tool, if programmed correctly in software. This particular LCD uses seven data I/O pins in order to communicate with the microcontroller. The following is a simple program which outputs a traditional message on the LCD.

- ❑ Close all files and click **File>New>Source File** and enter filename **EX6.c.**
- ❑ Type in the following program:

```
#include <prototype.h>
#include <utility.c>

#define LCD_ENABLE_PIN      PIN_C3
#define LCD_RS_PIN                PIN_C0
#define LCD_RW_PIN                PIN_C1
#define LCD_DATA4                 PIN_C4
#define LCD_DATA5                 PIN_C5
#define LCD_DATA6                 PIN_C6
#define LCD_DATA7                 PIN_C7

#include <lcd.c>

void main(void) {
      lcd_init();
      printf(lcd_putc,"Hello World");
}
```

- ❑ Click on Compile and Run the program.

- ❑ After verifying proper operation of the LCD, add the **#define** and **#include <lcd.c>** lines to the end of **prototype.h**.
- ❑ In the example above, the **printf** statement calls upon the function **lcd_putc** to print the string of characters. All normal rules regarding **printf** still apply when writing to the LCD using this method. Should you desire to print a single character to the LCD without using **printf**, you simply use the **lcd_putc** function. To illustrate this, add the following line after the **printf** statement, and compile and run the program:

```
lcd _ putc('8');
```

- ❑ Notice that the number 8 appears immediately after the “Hello World” statement. When using **lcd_putc**, be sure to use apostrophes (‘ ‘) around the character to be printed. Writing a simple decimal number with **lcd_putc** will result in the ASCII character for that number to printed, instead of the number itself. For example:

  `lcd_putc(52);` and `lcd_putc(‘4’);`

  will both cause the microcontroller to write the character ‘4’ to the LCD, since the ASCII value of the character ‘4’ is 52.

NOTES

- The LCD requires the function **lcd_init()** to be called in main before any writing to the LCD can begin.
- There are two functions in addition to **lcd_init()** and **lcd_putc** which the compiler can utilize. **lcd_gotoxy(x,y)** will set the write position of the LCD to the coordinates given as **x** and **y;** for example, **lcd_gotoxy(1,1)** will set the write position to the upper left-hand corner of the LCD. This function will not clear any of the characters already printed on the LCD, however. **lcd_getc(x,y)** will return the character printed on the LCD at coordinates **x,y;** for example, **lcd_getc(1,1)** will return the character printed at the upper left-hand corner of the LCD.
- While the LCD can be a valuable debugging tool, the required time to write a character to the LCD is rather long (up to 2 ms or longer). It is good programming practice to write to an LCD only when absolutely necessary, as opposed to every cycle through a **while(TRUE)** loop.

FURTHER STUDY

A *The compiler comes equipped with special characters for the LCD.* ***‘\f’*** *will cause the LCD to clear and set the write position to the upper-left hand corner again.* ***‘\n’*** *will set the write position to the beginning of the second line.* ***‘\b’*** *will set the write position one space back. Using these special characters, alter the program so that “Hello” prints on the top line of the LCD, and “World” prints on the bottom line. Then, clear the LCD and write “Hello World” on just the top line, then just the bottom line.*

7 TIMERS AND INTERRUPTS

- ❑ The PIC16LF727 has three built-in timers. Each timer has a different set of features. The following example will use TIMER0 to measure the time it takes to execute some C code.
- ❑ Close all files and click **File>New>Source File** and enter name **EX7.c.**

```
#include <prototype.h>

void main() {
       long time;
       long a,b,c;

       lcd_init();

       setup_timer_0(rtcc_INTERNAL|RTCC_DIV_1);
       set_timer0(0);
       a=b*c;
       time=get_timer0();
       printf(lcd_putc,"Time in ticks is\n%lu",time);
}
```

- ❑ Compile and Run the program.

  *Note that multiplication requires a significant amount of processing time to complete. The number displayed on the LCD is number of clock cycles the microcontroller used to complete the multiplication. This is number represents the same number of assembly instructions the microcontroller would have executed in the same amount of time.

- ❑ Close all files and click **File>New>Source File** and enter name **EX7_2.c**
- ❑ Type in the following program:

```
#include <prototype.h>

int16 overflow_count=0;

#int_timer0
void timer1_isr() {
       overflow_count++;
}

void main () {
       setup_timer_0(rtcc_INTERNAL | RTCC_DIV_256);
       enable_interrupts(int_timer0);
       enable_interrupts(global);
       while(TRUE) {
              printf(lcd_putc,"\fOverflows: %lu", overflow_count);
              delay_ms(10);
       }
}
```

❑ Compile and Run the program.

*Note the relative speed at which the timer increments.

NOTES

- All of the timers on the PIC16LF727 count up and when the maximum value is reached, the timer restarts at 0. The set_timer0(0) resets the timer to 0. If the interrupt for timer0 AND global interrupts are enabled, an interrupt request will be generated when the timer restarts to 0 after its maximum value.
- If using RTCC_DIV_256 instead of RTCC_DIV_1, then the timer would increment once for every 256 instruction clocks. This effectively makes the timer count 256 times slower than the nominal rate.
- The interrupt function is designated by preceding it with #INT_TIMER1. A number of interrupt functions can be specified by preceding each with the proper directive like #INT_EXT for the external interrupt pin (B0) or #INT_RDA for an incoming RS-232 character.
- An interrupt must be specifically enabled (via enable_interrupts(specific interrupt)) AND interrupts must be globally enabled (via enable_interrupts(GLOBAL)). The GLOBAL enable/disable controls whether any interrupts are serviced.
- If interrupts are globally disabled and an interrupt event happens, then the interrupt function will be called when interrupts are enabled. If multiple interrupt events of the same type happe n while interrupts are disabled, then the interrupt function is called only once when interrupts are enabled.

❑ The following is a summary of the timers on the PIC16LF727 chip:

#0	Input is Instruction Clock or external pin Range is 0-255 Input can be divided by 1,2,4,8,16,32,64,128,256 Can generate interrupt on each overflow
#1	Input is Instruction Clock or external pin Range is 0-65535 Input can be divided by 1,2,4,8 Can generate interrupt on each overflow
#2	Input is Instruction Clock only Range can be programmed from 0-1 to 0-255 Input can be divided by 1,4,16 Can generate interrupt on 1-16 overflows

8 PWM AND STAND-ALONE PROGRAMS

Buzzers are a popular user interface feature. Driving a buzzer requires changing the state of an individual pin on a regular basis. Implementing this with output_high() and output_low() commands, however, will cause the processor to be almost completely consumed to wait during the buzzing. A less processor-intensive method is to use the Capture/Compare/PWM Module. This kit will focus mainly on Pulse-Width Modulation (PWM) as a way to alter the main clock signal of the microcontroller, in order to drive external hardware which requires a modulated signal input. When using PWM, there are two ways to alter the main clock signal:

1. Divide the frequency of the clock (such as lowering oscillation frequency from 8MHz to 1Mhz)
2. Change the duty cycle of signal (discrete values between 0% and 100%)

The prototype board has a small buzzer driven by the pin connected to the PWM1 module. The next exercise will use TIMER2, which is driven by the main processor clock, to alter the signal being applied to the buzzer; which in turn will affecting the way it sounds.

- ❑ Close all files and click **File>New>Source File** and enter filename **EX8.c**. Type in the following program.

```
#include <prototype.h>
#include <utility.c>

void main() {

       SETUP_TIMER_2(T2_DIV_BY_16,200,1);

       while(TRUE) {
              SETUP_CCP1(CCP_PWM);
              SET_PWM1_DUTY(402);
              delay_ms(100);
              SETUP_CCP1(CCP_OFF);
              delay_ms(1000);
       }
}
```

- ❑ Compile and Run the program.
- ❑ The buzzer make a short sound every 1.1 seconds. The buzzer will sounds because a voltage sensitive plate moves up or down inside the buzzer every time the output pin of the microcontroller changes. The frequency of the sound generated is exactly equal to the frequency at which the output pin changes.

  The frequency of the sound from the buzzer can be predicted by looking at our SETUP_TIMER_2 call. Each prescaler and count instruction divides the internal clock frequency which drives the TIMER2. Notice that TIMER2 uses a prescaler of 16, and counts to 200 before changing the PWM output pin's state. The instruction clock also requires 4 ticks to execute one instruction. Since the instruction clock is operating at 8 MHz (see the #use delay line in prototype.h), the frequency out of the PWM (and into the buzzer) is

  (8,000,000 Hz) / (16*200*4) = 625 Hz
  (Internal Clock) / (Buzzer Frequency * 4) = Prescaler * TopCount

The relative amount of time the output pin of the PWM is high versus low is known as the duty cycle of the PWM. This is usually given as a percentage of the time the output pin is high: 0% being always low, 100% being always high. When programming, the **SET_PWM1_DUTY(x)** call allows you to set the duty cycle parameter to a 10-bit value, x. To calculate the duty cycle, divide the parameter **x** by **[4 * (TopCount + 1)]**, which is the relative length of one cycle. In our code:

(x / [4 * (TopCount +1)]) = (402 / [4*(200+1)]) = (402 / 804) = 50%

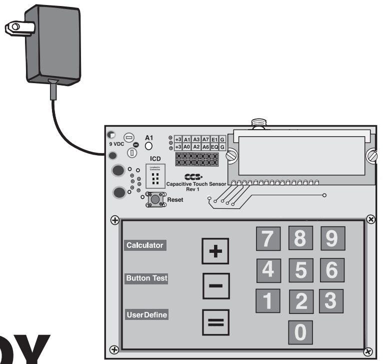
Therefore, the output pin is high exactly the same amount of time that it is low, and a uniform square wave is generated. For buzzer purposes, the duty cycle will not have much impact on how the sound is heard, but other hardware applications, including motor control, may require specific duty cycles to overcome problems such as noise or large inductance. NOTE: any duty cycle greater than or equal to 100% will keep the PWM output pin permanently in the HIGH state.

- ❑ Close all files and open the prototype.h file. Copy this file and remove the line:

```
#device ICD=TRUE
```

  Name the file ‘protoalone.h’.

- ❑ Modify **EX8.c** to use **protoalone.h**.
- ❑ Compile the program, then click **Tools>ICD** to load the program onto the prototyping board.
- ❑ Disconnect the power from the prototyping board, then disconnect the ICD from the prototyping board.
- ❑ Power up only the prototyping board and verify the program runs correctly.



FURTHER STUDY

A *Modify the program so that the LED turns on whenever the buzzer is sounding.*

B *Make a version of the program that plays a variety of notes and outputs the current frequency in Hz on the LCD. Humans can detect frequencies from around 20-20000 Hz,so using a prescaler of 16 will drop the frequencies into normal hearing range.*

9 CAPACITIVE SENSING DESIGN THEORY

Theory

The PIC16LF727 contains a Capacitive Sensing Module (CSM) which, with proper hardware and software configuration, allow detection of changes in the capacitance of an external pad. It is important to understand the basic theory behind the CSM and its operation in order to use in future projects.

The CSM when activated, turns on an oscillator consisting of a constant current source and a constant current sink. This results in a triangular voltage waveform when put through a capacitor. When the external capacitor is connected with the internal resistor, a simple relaxation oscillation circuit is created. From basic electric circuit theory, the relative frequency of relaxation oscillator with fixed inputs and a fixed resistance is based completely on the capacitor in the circuit. For example, by changing the capacitance of the circuit (externally), in a change in the frequency of the oscillator will occur.

The human finger has dielectric properties. When a finger is brought near the external capacitive pad, the dielectric properties of the finger interact with the electric field of the capacitor; causing the capacitor in the circuit to appear to have more capacitance than usual. Thus, resulting a change in the frequency in the oscillator circuit.

One way to quantitatively measure the frequency of a digital circuit is to use the built-in Timer peripherals. The CSM is equipped with the option of internally connecting the output of the oscillation circuit with the input of TIMER1. Each rising edge of the oscillating voltage will cause the counter to increment by 1. TIMER0 can be used to simultaneously “watch” the TIMER1 peripheral for a fixed amount of time to determine a change in frequency. Compare the number of ticks in TIMER1 after a fixed period of time to the number of ticks in another fixed period of time.

Implementation

The following is a software model for getting fast, reliable results from the timers based on the previously described theory. The duties that the software must perform to efficiently use this peripheral include:

- Setup timers 0 and 1
- Initialize the CSM (start the circuit oscillating)
- Upon each interrupt of timer0, do the following:
    - Read the value of timer1
    - Compare with a nominal threshold value
    - If less, store a key press in memory; if greater, ignore cycle
    - Monitor the next key of interest (multiplex)
    - Reset timer0 and timer1 and return
- If a key has been pressed, effect some kind of external change (LCD, RS232, LED, etc.)

The above algorithm gets more complex as more buttons are monitored. A significant amount of memory is required in order to compare each key to its nominal value, since the baseline reading of key presses can change due to alterations in size and location to the capacitive pads. As such, multiple data arrays will be required. In programs of this complexity, there is more room for errors and improper coding. The next chapter will focus on simply displaying the value of timer1 on the LCD in order to appreciate the sensitivity of the capacitive pads to a finger press.

NOTES

- This software algorithm requires exclusive use of the TIMER1 peripheral and therefore, TIMER1 cannot be used by any other function or subroutine without causing errors in the CSM.
- While TIMER0 is used to clock the TIMER1 peripheral, it is still possible to simultaneously use the TIMER0 ISR for a different purpose in the main program. In this case, the CCS compiler will automatically prioritize the CSM ahead of the user's commands in the ISR. If choosing to not use the CCS library with the CSM, it is recommended that the user functions in the TIMER0 ISR should be called after the necessary CSM operations are executed.
- For more information on the CSM of this particular chip, reference the PIC16F72X/PIC16LF72X data sheet, section 14.

10 CAPACITIVE SENSING WITH LIBRARY FUNCTIONS

The CCS compiler includes built-in functions which greatly simplify the capacitive sensing process. The following program, allows the microcontroller to detect presses on pads ‘0’, ‘3’, and ‘6’, and prints the key pressed to LCD.

- ❑ Close all files and click **File>New>Source File** and enter filename **EX11.c** and type in the following program.

```
#include <prototype.h>
#include <utility.c>

#use touchpad(scantime=32ms,threshold=6,pin_b1=’0’,pin_d0=’3’,pin_
d1=’6’)

void main(void) {
   INT i;
   lcd_init();
   enable_interrupts( GLOBAL);

   while(TRUE) {
                 i=touchpad_getc();
                 lcd_putc(i);
   }
}
```

- ❑ Compile and Run the program. Press the pads labeled as ‘0’, ‘3’, or ‘6’ and verify these digits appears on the LCD.
- ❑ Reset the program and set a breakpoint on the **lcd_putc(i)** line. Click the GO button. Notice that the program does not reach the breakpoint until a key has been pressed. The function **touchpad_getc()** cannot return a null value and will wait until the program stores a key press value in memory before returning. To avoid this active waiting of the microcontroller, change the **while(TRUE)** loop in the program as shown below and save the program as **EX11_2.**
- ❑ Next, change the **while(TRUE)** loop in the program to avoid the active waiting of the microcontroller.
- ❑ Modify the program as shown below and save the file as **EX11_2.c.**

```
while(TRUE) {
     if( touchpad_hit() ){
                                i=touchpad_getc();
                                lcd_putc(‘\f’);
       lcd_putc(i);
       delay_ms(1000);
     }
     else {
       printf(lcd_putc,”\fNo Press”);
       delay_ms(100);
     }
   }
```

- ❑ The touchpad_hit() function will return true only if there is a key press in memory that has not been called by touchpad_getc() yet. Using this function allows the microcontroller to do other operations while waiting for a key to be pressed.
- ❑ The compiler has de-bouncing software, such that a single press only causes the LCD to print a character once, but the threshold value in the **#use touchpad** directive may need adjusting specific to the of hardware.

NOTES

- In **#use touchpad(options)**, the **scantime=XXms** and **threshold=X** commands are optional. If they are not explicitly declared, the compiler defaults scantime to 32ms and the threshold value to 6% below nominal reading.
- The **scantime** parameter of **#use touchpad** is a per key measurement. That is, if the scan time is set to 32ms, the microcontroller will clock TIMER1 for 32ms for each key declared in **#use touchpad**. In the previous example, there are three keys being monitored, so it takes (32ms * 3) = 96ms for the microcontroller to scan all of the keys only once. If all 16 possible pads are being monitored, it would take (32ms * 16) = 512ms for the microcontroller to scan all of the keys one time.
- When declaring pins in **#use touchpad(options)**, take care to differentiate between characters and integer types. Recall that the **char** value of '0' has an **int** value of 48. Also, recall that the LCD prints **char** values, not **int** values.
- The touchpad_getc() function returns a char value. If the receiving memory location is declared as an int, the receiving memory location will receive the ASCII value of the key press.

FURTHER STUDY

A *Add pins to the **#use touchpad** directive such that the keys '0'-'9' are active, and note the delay between when a key is pressed and when it appears. Then, alter the **scantime** and **threshold** parameters so the reaction time is increased without sacrificing accuracy.*

B *Add conditional statements in the program that will reset the LCD when the '=' key is pressed, and will have the LCD write to the bottom line when the '+' key is pressed.*

11 ADVANCED CAPACITIVE SENSING AND ARRAYS

The following program uses the timer0 interrupt to shift key presses into a global array and continuously displays the entire array on the LCD.

- ❑ Close all files and click **File>New>Source File** and enter filename **EX12.c**. Type in the following program.

```
#include <prototype.h>
#include <utility.c>

#use touchpad(pin_b1='A',pin_d0='B',pin_d1='C')

#define num 8

int i;
char A[num];

#int_timer0
void timer0_isr() {
   if( TOUCHPAD_HIT() )
      if(!(TOUCHPADSTATUS&0x0F00)){
         for(i=(num-1);i>0;i--)
            A[i]=A[i-1];
         A[0]=touchpad_getc();
   }
}

void main(void) {
   int j;
   for(j=0;j<num;j++)
      A[j]='0';

   lcd_init();
   enable_interrupts( GLOBAL);

   while(TRUE){
      lcd_putc('\f');
      for(j=0;j<num;j++)
         lcd_putc(A[j]);
      delay_ms(100);
   }
}
```

- ❑ Compile and Run the program. Verify that the LCD displays contents of the character array A and that a new key press on pads ‘0’, ‘3’, and ‘6’ shifts the entire array to the left. Note that the pads now return the letters ‘A’, ‘B’, and ‘C’ respectively. This can be changed in the **#use touchpad** directive.

NOTES

- A C array declared as **A[8]** means the valid subscripts are **A[0]** through **A[7]**.
- Using the **#define num** line near the beginning of the program allows for better readability and easier revisions. In order to change the size of the array, only change the number that **num** is defined to be. Without this line, changing the size of the array would require making many replacements elsewhere in the program.
- There is a 16-bit C variable called **TOUCHPADSTATUS** that the compiler declares whenever the **#use touchpad** directive is used. The compiler uses this variable for calibrating, flagging, debouncing, and storing key presses with the CSM. The table below shows how each bit of **TOUCHPADSTATUS** is used by the compiler. As shown, the second nibble of **TOUCHPADSTATUS** is used for calibrating the CSM on start-up. Until the CSM is calibrated, the following line is needed in the TIMER0 ISR. To avoid writing to the character array while the CSM is calibrating:

```
        if(!(TOUCHPADSTATUS&0x0F00))
```

- In addition to **TOUCHPADSTATUS**, there is also a large data array created and declared by the compiler called **TOUCHDATA** which stores the threshold timer values for each key. While normally hidden in the background by the compiler, it is possible to read and write to this array, if desired.

Bit#	15	14	13	12	11	10	9	8
Name	TEST	PRESS	RES1	RES0	CSMC3	CSMC2	CSMC1	CSMC0

Bit#	7	6	5	4	3	2	1	0
Name	KEY7	KEY6	KEY5	KEY4	KEY3	KEY2	KEY1	KEY0

TEST
1 : Test mode
0 : Normal Running mode

PRESS
1 : Key is currently being pressed
0 : Key is currently not being pressed

RES1 : 0
Reserved bits, should be left as '0'

CSMC3 : 0
CSM calibration bits. Four bit number represents number of keys left to calibrate.

KEY7 : 0
ASCII value of the most recent key pressed. Value will be passed to **touchpad_getc( )** on call. After **touchpad_getc( )** is executed, this buffer is cleared.

FURTHER STUDY

A *Modify the program so that the green LED turns on when the character array is full.*

B *Define a new key such that when it is pressed during normal operation, the entire character array is reset to '0'.*

12 USING A RS-232 PORT

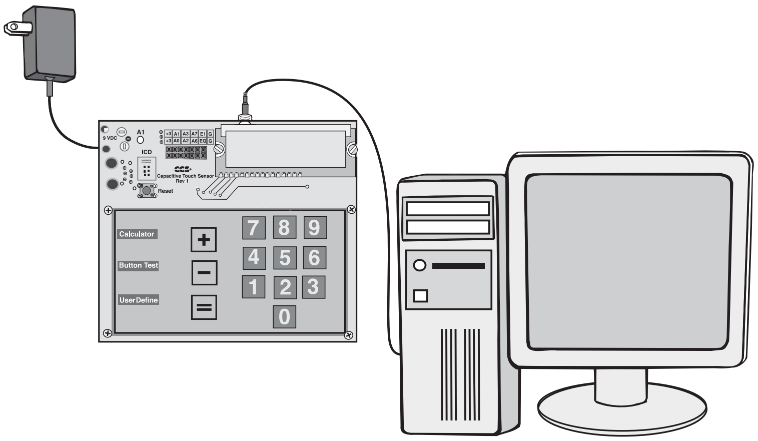
- ❑ RS-232 is a popular serial communications standard used on most PCs and many embedded systems. Two wires are used (in addition to ground), one for outgoing data and one for incoming data. The compiler will allow any pins to be used for RS-232, however, the prototyping board already has an RS-232 jack connected with pins E0 and E1. Add the following line to the end of the protoalone.h file:

  ```
  #use rs232 (baud=9600,  xmit=PIN_E1, rcv=PIN_E0)
  ```

- ❑ Close all files and click **File>New> Source File** and enter the filenameile **EX13.c.** Type in the following program:

```
#include <protoalone.h>
#include <stdlib.h>
#include <input.c>

void main() {
   long a,b,result;
   char opr;

   setup_timer_0(RTCC_INTERNAL);
   while(TRUE) {
      printf("\r\nEnter the first number: ");
      a=get_long();

      do {
         printf("\r\nEnter the operator (+-*/): ");
         opr=getc();
      } while(!isamong(opr,"+-*/"));

      printf("\r\nEnter the second number: ");
      b=get_long();

      switch(opr) {
         case '+' : result= a+b; break;
         case '-' : result= a-b; break;
         case '*' : result= a*b; break;
         case '/' : result= a/b; break;
      }

      printf("\r\nThe result is %lu ",result);
   }
}
```

- ❑ Compile and Run the program. Check the monitor tab to see the result.
- ❑ Connect the prototyping board to the PC as shown below.



- ❑ At the PC, close the debugger window and start the program Tools>Serial Port Monitor. Set the correct COMM port if necessary. Ensure that the monitor is set to view and send in ASCII mode, not HEX mode.
- ❑ Power up the prototyping board and a prompt at the PC should appear. Enter a number followed by the enter key, an operator (like +) and another number followed by enter. Verify the result is shown correctly.

NOTES

- The basic functions for RS-232 are putc() and getc(). printf calls putc() multiple times to output a whole string and format numbers if requested. get_long() is a function in input.c to read a long number by calling getc() many times. See input.c for other functions such as get_int() and get_string().
- The % in the printf indicates another parameter is included in the printf call and it should be formatted as requested. %lu indicates to format as an unsigned long.
- getc() will cause the program to stop and wait for a character to come in before it returns.

FURTHER STUDY

A *Modify to add the operators: % | & ^*

B *Modify to use float instead of long. You will need to do get_float() instead of get_long() and use the format specifier %9.4f to get 4 digits after the decimal place.*

13 CAPACITIVE SENSING AND RS-232

For many embedded systems applications, the processing power of the PIC16LF727 alone may not be enough, especially when 16 capacitive touch pads are being monitored. In these cases, it is logical to use the PIC16LF727 as a slave processor to a more powerful master controller.

Two steps are needed to set up a master-slave computer system: 1) create a way for the slave to interrupt the master, and 2)set up a method for communication between slave and master.

To interrupt a master microcontroller, have one port pin change states whenever a key has been pressed. Externally connect this pin to the external interrupt pin of the master microcontroller. If microcontroller also has communication based interrupts, those could be used as well to interrupt and transmit data simultaneously.

Transferring data between the master and slave can be more complicated. Fortunately, most microcontrollers and compilers (including the CCS compiler) allow for several communication possibilities, including AUSART (which RS-232 is derived from), SPI, I2C, and multiple user defined I/O pins which can be used to generate one's own communication protocol.

Chapter 13 experimented with RS-232, In which both the PC and microcontroller communicated, in order to accomplish the task of creating a simple calculator. Expanding on that method, by connecting two processing systems together.

- ❑ Close all files and click **File>New>Source File** and enter filename **EX14.c**. Type in the following program.

```
#include <protoalone.h>
#include <utility.c>
#include <stdlib.h>
#include <input.c>

#use touchpad(pin_b1='0',pin_d0='3',pin_d1='6')

void main() {
   int i;
   char check;

   enable_interrupts(GLOBAL);

   printf("\r\nConnected");

   while(TRUE) {
      i = touchpad_getc();
      do {
         printf("\r\nData ready, send '1' to receive: ");
         check=getc();
      } while(!isamong(check,"1"));

      printf("\r\n%c",i);
   }
}
```

- ❑ Compile the program.
- ❑ Load the program onto the chip and open the Serial Input/Output Monitor. Verify that pressing ‘0’, ‘3’, or ‘6’ causes a prompt to show up in the monitor, and consequently sending a ‘1’ from the PC will result in the pressed number to show in the monitor.

NOTES

- This program did not undertake how the PC should handle incoming data. When programming a master microcontroller, the send/receive process should mirror the send/receive process of the slave. That is, if one device is using putc(), the other device should be using getc() at the same time, and vice versa.
- This program contains a great deal of text being sent from the PIC16LF727 to the PC. In a real application where the master chip has no user interface and will automatically receive data, this text is not required, and the data transaction can occur at a much faster rate.
- The exercises in this kit were designed to wait for a ready signal (sending a ‘1’ to the microcontroller), to ensure that the receiving computer is ready for the data being transmitted. A PC is virtually always ready to send/receive data, but a master microcontroller may need to clear memory buffers, finish another instruction, etc., before being ready to receive. Using this method ensures that the key press is not lost simply because the master was not ready to receive.
- The prototyping board has RS-232 mapped to pins E0 and E1. The PIC16LF727 has available hardware and interrupts for RS-232, but only if the original pins C6 and C7 are used. As such, the prototyping board cannot be interrupted by RS-232 and must anticipate incoming data. When designinga circuit, using RS-232, pins C6 and C7 will allow access to RS-232 interrupts and buffering hardware.

FURTHER STUDY

A *In a similar fashion to EX12.C, buffer (store in order) the key presses on the PIC16LF727 in a data array. Once four key presses are stored in the array, announce a buffer overflow on RS-232 and transmit all four key presses upon a ready signal from the PC.*

B *Modify EX12A.C so that the green LED turns on when a key is pressed and the buzzer turns on when the buffer is full.*

14 ADVANCED DEBUGGING

- ❑ RS-232 printf statements can be a good tool to help debug a program. It does, however, require an extra hardware setup to use. If the ICD is being used as a debug tool, the compiler can direct putc() and getc() through the debugger interface to the debugger screen. Add the following line to the end of the **prototype.h** file:

```
#use rs232 (DEBUGGER)
```

- ❑ Modify **EX13.C** to create **EX15.C** by changing **protoalone.h** to **prototype.h."**
- ❑ Compile and load the program into the Prototyping board.
- ❑ Click GO, then click the **Monitor** tab.
- ❑ A prompt should appear. Enter some data to confirm that the program is working.
- ❑ Stop and reset the program.
- ❑ In PCW click **Project>Open all files** as an easy way to get all the project files open in IDE.
- ❑ Click the **stdlib.h** tab, and set a breakpoint in the atol() function on the line:

```
result = 10*result + (c - ‘0’);
```

- ❑ This function is called from get_long() to convert a string to a number. This line is executed for each character in the string.
- ❑ Click the debugger **Break Log** tab, check the LOG box, set the breakpoint as 1 and expression as **result**. Result is the value of the number being converted.
- ❑ Click GO, then click the **Monitor** tab and enter **1234 enter.**
- ❑ Click the **Log** tab and notice that each time the breakpoint was hit the value of the **result** variable was logged. In this case the breakpoint did not cause a full stop of the program, it just logged the value of the requested expression and kept on going.
- ❑ Stop the program.
- ❑ Delete the breakpoint by selecting the breakpoint and click on the 🟥 icon.
- ❑ Uncheck the LOG box under the log tab.
- ❑ Set a breakpoint on the last printf() in the program.
- ❑ Enter watches for **a, b** and **result**.
- ❑ Click GO and enter two numbers and +.
- ❑ When the break is reached click on the snapshot icon: 📷
- ❑ Check **Time** and **Watches**, uncheck everything else.
- ❑ If a printer is connected to the PC select **Printer**, otherwise select **Unique file**.
- ❑ Click on the **Now** button.
- ❑ Notice the requested data (time and watches) are either printed or written to a file as requested.

- ❑ Click on the snapshot icon again and this time select **Append to file**, put in a filename of EX15.TXT and check **After each single step**.
- ❑ Check **Last C line executed** in addition to the **Time** and **Watch** selected already and close the snapshot window.
- ❑ Reset and then Step Over until the final printf() is executed. Enter the data when requested.
- ❑ Use **File>Open>Any File** to find the file **EX15.TXT** (by default in the Debugger Profiles directory) after setting the file type to all files.
- ❑ Notice the log of what happened with each step over command.
- ❑ Uncheck the **After each single step** in the snapshot window.
- ❑ Clear the breakpoints and set a breakpoint on the **switch**.
- ❑ Click Reset then Go and enter the requested data using the + operator.
- ❑ When the break is reached click on the **Peripherals** tab and select Timer 0.
- ❑ Shown will be the registers associated with timer 0. Although this program does not use timer 0 the timer is always running so there is a value in the **TMR0** register. Write this value down.
- ❑ Clear the breakpoints and set a breakpoint on the final **printf()**.
- ❑ Click GO.
- ❑ Check the **TMR0** register again. If the new value is higher than the previous value then subtract the previous value from the current value. Otherwise, add 256 to the current value and then subtract the previous value (because the timer flipped over).
- ❑ The number we now have is the number of clock ticks it took to execute the switch and addition. A clock tick by default is 0.5ms. Multiply your number of ticks by 0.5 to find the time in ms. Note that the timers (and all peripherals) are frozen as soon as the program stops running.

FURTHER STUDY

A *The debugger **Eval** tab can be used to evaluate a C expression. This includes assignments. Set a break before the switch statement and use the Eval window to change the operator being used. For example, type a + but change it to a - before the switch.*

B *Set a break on the switch statement and when reached, change to the C/ASM view and single step through the switch statement. Look up the instructions executed in the PIC16LF727 data sheet to see how the switch statement is implemented. This implementation is dependent on the case items being close to each other. Change * to ~ and then see how the implementation changes.*

15 CAPACITIVE TOUCH PROJECT EXAMPLE

This exercise will combine almost all of the previous chapters into one working project to illustrate how using the built-in peripherals can result in a complex, but elegant, system.

❑ Add the following functions to utility.c:

```
void result_print(long number, long & a, long & b, char & opr, int1 &
calc)
{
   lcd_putc('\f');
   printf(lcd_putc,"\f%lu",number);
   a=number;
   b=0;
   opr='0';
   calc=0;
}

void send_data(long number)
{
   SETUP_CCP1(CCP_PWM);
   printf("\r\n%lu",number);
   delay_ms(50);
   SETUP_CCP1(CCP_OFF);
}
```

❑ Close all files and click **File>New>Source File** and enter filename **EX14.c.**

❑ Type in the following program. Note that #fuse touchpad directive should be all on one line.

```
#include <protoalone.h>
#include <utility.c>
#use touchpad(scantime=1ms,pin_a5='X',pin_b5='R',pin_b2='+',pin_b3='-
',pin_b4='=',pin_d3='8',
pin_d5='5',pin_d7='2',pin_d4='7',pin_d6='4',pin_b0='1',pin_
b1='0',pin_d0='3',pin_d1='6',pin_d2='9')

#int_timer0
void timer0_isr() {
   if( TOUCHPADSTATUS&0x4000 ) {
      output_low(GREEN_LED);
      output_low(PIN_C2);
   }
   else {
      output_high(GREEN_LED);
      output_high(PIN_C2);
   }
}
```

continued...

...continued

```
void main(void) {
   long i,inti,a=0,b=0,temp=0,result;
   char opr='0';
   int1 calc=0,opr_press=0,reset=0,send=0;

   lcd_init();
   SETUP_TIMER_2(T2_DIV_BY_16,255,1);
   enable_interrupts( GLOBAL);
   delay_ms(500);

   while(TRUE) {
      opr_press=0;
      while(!touchpad_hit());
      i=touchpad_getc();
      inti=i-48;

      switch(I){
         case '+' : opr='+'; opr_press=1; break;
         case '-' : opr='-'; opr_press=1; break;
         case 'R' : reset=1;
         case 'X' : send=1;
         case '=' : if(opr!='0'){calc=1;} break;
         default  : if(opr!='0'){temp=(b*10); b=temp+inti; break;}
                    else{temp=(a*10); a=temp+inti; break;}
         }

      if(reset==1){
         result_print(0,a,b,opr,calc);
         reset=0;
         send=0;
      }

      if((calc==0)&&(opr=='0')&&(opr_press!=1))
         printf(lcd_putc,"\f%lu",a);
      if((calc==0)&&(opr!='0')&&(opr_press!=1))
         printf(lcd_putc,"\f%lu",b);
      if(opr_press==1)
         printf(lcd_putc,"\f%c",opr);
      if(send==1){
         send_data(a);
         send=0;
```

continued...

15 CAPACITIVE TOUCH PROJECT EXAMPLE (CONT.)

...continued

```
         }
         if(calc==1){
            switch(opr){
               case '+' : result=a+b;
                  result_print(result,a,b,opr,calc);
                  break;
               case '-' : result=a-b;
                  result_print(result,a,b,opr,calc);
                  break;
            }
         }
      }
}
```

- ❑ Compile the program.
- ❑ Load the program onto the chip and Run the program.
- ❑ The prototyping board acts as an integer calculator with the keys '0' through '9' and '+', '-', and '=' active. When the '=' key is pressed, the result of the computation is stored as the first integer in a new computation.
- ❑ Press the "User Define" key after performing a computation. This key empties all calculator memory and returns the calculator to its initial state.
- ❑ Open the Serial Monitor from CCS IDE and ensure it is in ASCII mode. Pressing the "Calculator" button on the device should print the current value of the first integer in the computation (which is also the result of the last computation if the '=' has just been pressed), and listen for a short beep.
- ❑ Pressing and holding any of the active keys will cause the green LED to turn on and a clicking sound. Releasing the key will cause the green LED to turn off and another clicking sound.

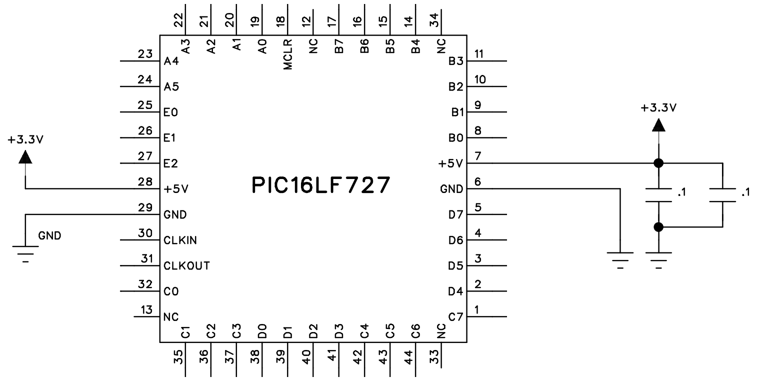
FURTHER STUDY

A *Modify the program so that the "Button Test" key automatically takes the square root of the current number.*
Note: *the square root of a number will be truncated to an integer.*
(Hint, use the built-in C function **sqrt()** and the header file **math.h**).

16 MIGRATING TO YOUR OWN HARDWARE

The following diagram is a somewhat minimal circuit for a PIC18F4520

- ❑ Notice this chip has two +3.3V and ground connections. Some chips have only one of each. A 0.1μf capacitor mounted near the chip is a good idea and one on either side of the chip is even better. This will reduce noise both to and from the chip.
- ❑ The PIC16LF727 circuit shown below relies on the internal oscillator present in the chip itself. There are multiple clock settings available for the PIC16LF727. Reference the PIC16F727/PIC16LF727 data sheet for the available options.



Troubleshooting

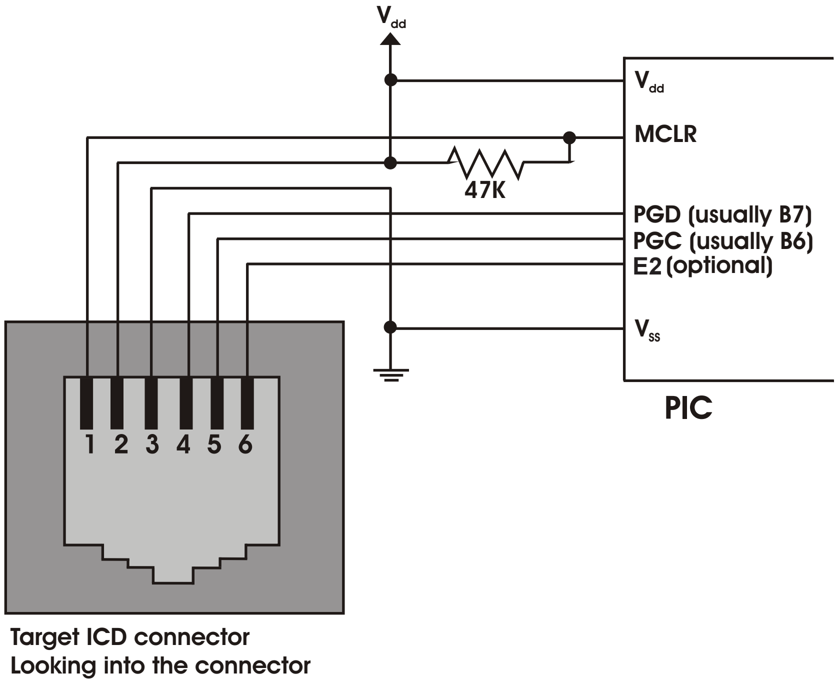
- ❑ The MCLR pin must be in a high state for the chip to run. Note the Prototyping board schematic uses a pushbutton to ground this pin and to reset the chip.
- ❑ Most problems involve the clock. Make sure the configuration fuses are set to the proper oscillator setting. In the above case, for an internal oscillator operating at 8 MHz, HS (High-Speed) is the proper setting.
- ❑ If the program does not seem to be running, verify 3.3 Volts on the MCLR pin and the two power pins.
- ❑ Isolate hardware problems from firmware problems by running a program with the following at the start of main ( ) and check B0 with a logic probe or scope:

```
while(TRUE)   {
        output_low (PIN_B0);
        delay_ms (1000);
        output_high (PIN_B0);
        delay_ms (1000);
}
```

MIGRATING TO YOUR OWN HARDWARE (CONT.)

The In-Circuit Progamming/Debugging Interface

- ❑ To program and/or debug in circuit, two I/O pins (B6, B7) are reserved. If debugging is not to be done, then these pins may also be used in the target circuit. However, care must be taken to ensure the target circuit has high impedance during programming.
- ❑ The MCLR pin is also used by the programmer and for debugging. Note that during programming, the voltage on this is 13 volts. The 47K resistor to 3.3V is sufficient isolation for the 13V. However, if anything else is connected to the MCLR pin, be sure the 13V will not damage or interfere.
- ❑ The ICD unit requires V_{dd} from the target. It is easiest to power up the target normally and then, connect the target board V_{dd} to the ICD for power. The ICD-S40 is powered by this pin (5V) and the ICD-U40 uses it to pull up the signals (3V-5V).
- ❑ The E2 pin is optional and is not used for programming. However, the monitor feature of the debugger does use E2. It is possible to program and debug (without monitor) and allocate E2 to the target hardware. In this case do not connect B3 to the ICD connector.



- ❑ Note that the ICD to target cable reverses the pins so the MCLR signal is ICD pin 6 and that connects to the target pin 1.

Layout Hints

- ❑ When designing a capacitive pad, the focus should be on the size of the pad, not the shape. A larger pad will have a greater sensitivity than a smaller pad. Also, sufficient space should be left between the pads to avoid false presses. Experimentally, a separation of approximately 5mm was found to be adequate.

- ❑ When designing the physical layout of the pads on the PCB, be sure to keep the traces and pads away from ground. Being too close to a ground trace or plane will change the capacitive properties of the pad, and will reduce or eliminate the functionality of the board.

- ❑ If possible, keep the area above and below the sensors and sensor traces on the PCB clear of any high frequency devices or wires, even if they are shielded or insulated. If this is not possible, it is good design practice to keep the sensor traces perpendicular to any high frequency traces, in order to minimize high frequency interference.

- ❑ The covering plate over the printed circuit board (PCB) is ideally an insulating, strong dielectric material. The higher the dielectric constant of the material, the better the sensitivity will be. Conductive materials should not be used, as they will ruin the capacitive properties of the plates.

- ❑ Sandwiching the PCB and covering plate is appropriate, but not necessary. Since the capacitive sensing module relies on electric field interactions, contact is not a requirement. However, the closer a finger gets to the capacitive pad, the more responsive the sensor will be. This development kit has fixed sensors on the PCB, but this not a requirement. The sensors may be detachable from the PCB, and jumpers may be run from the microcontroller to the sensors.

References

This booklet is not intended to be a tutorial for the PIC16LF727 or the C programming language. It does attempt to cover the basic use and operation of the development tools. There are some helpful tips and techniques covered, however, this is far from complete instruction on C programming. For the reader not using this as a part of a class and without prior C experience the following references should help.

Exercise	PICmicro® MCU C: An introduction to Programming the Microchip PIC® in CCS by Nigel Gardner	The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (2nd ed.)
3	1.1 The structure of C Programs 1.2 Components of a C Program 1.3 main() 1.5 #include 1.8 constants 1.11 Macros 1.13 Hardware Compatibility 5.5 While loop 9.1 Inputs and Outputs	1.1 Getting Started 1.4 Symbolic Constants 3.1 Statements and Blockx 3.5 Loops 1.11 The C Preprocessor
4	1.7 Variables 1.10 Functions 2.1 Data Types 2.2 Variable Declaration 2.3 Variable Assignment 2.4 Enumeration 3.1 Functions 3.4 Using Function Arguments 4.2 Relational Operators 5.7 Nesting Program Control Statements 5.10 Switch Statement	1.2 Variables and Arithmetic Expr 2.1 Variable Names 2.2 Data Types and Sizes 2.3 Constants 2.4 Declarations 2.6 Relational and Logical Operators 3.4 Switch 1.7 Functions 1.8 Arguments 4.1 Basics of Functions
5	4.3 Logical Operators 4.4 Bitwise Operators 4.5 Increment and Decrement 5.1 if Statements 5.2 if-else Statements 9.3 Advanced BIT Manipulation	3.2 if-Else 2.8 Increment and Decrement Ops 2.90 Bitwise Operators
6	4.1 Arithmetic Operators	2.5 Arithmetic Operators
7	9.5 A/D Conversion	3.3 Else

8	5.4 For Loop 6.1 One-Dimensional Arrays	1.3 The For Statement 1.6 Arrays 2.10 Assignments Operators and Exp
10	1.6 printf Function 9.6 Data Comms/RS-232	1.5 Character Input and Output 2.6 Loops-Do-While 7.1 Standard Input and Output 7.2 Formatted Output - printf
11	6.2 Strings 6.4 Initializing Arrays 8.1 Introduction to Structures	7.9 Character Arrays 6.1 Basics of Structures 6.3 Arrays of Structures
13	9.4 Timers	
14	2.6 Type Conversion 9.11 Interrupts	2.7 Type Conversions
16	9.8 SPI Communications	
17	9.7 I^{2}C Communications	
18	5.2 ? Operator	2.11 Conditional Expressions
19	4.6 Precedence of Operators	2.12 Precedence and Order Eval

On The Web

Comprehensive list of PIC® MCU Development tools and information	www.mcuspace.com
Microchip Home Page	www.microchip.com
CCS Compiler/Tools Home Page	www.ccsinfo.com
CCS Compiler/Tools Software Update Page	www.ccsinfo.com click: Support → Downloads
C Compiler User Message Exchange	www.ccsinfo.com/forum
Device Datasheets List	www.ccsinfo.com click: Support → Device Datasheets
C Compiler Technical Support	support@ccsinfo.com

Other Development Tools

EMULATORS

The ICD used in this booklet uses two I/O pins on the chip to communicate with a small debug program in the chip. This is a basic debug tool that takes up some of the chip's resources (I/O pins and memory). An emulator replaces the chip with a special connector that connects to a unit that emulates the chip. The debugging works in a simulator manner except that the chip has all of its normal resources, the debugger runs faster and there are more debug features. For example an emulator typically will allow any number of breakpoints. Some of the emulators can break on an external event like some signal on the target board changing. Some emulators can break on an external event like some that were executed before a breakpoint was reached. Emulators cost between $500 and $3000 depending on the chips they cover and the features.

DEVICE PROGRAMMERS

The ICD can be used to program FLASH chips as was done in these exercises. A stand alone device programmer may be used to program all the chips. These programmers will use the .HEX file output from the compiler to do the programming. Many standard EEPROM programmers do know how to program the Microchip parts. There are a large number of Microchip only device programmers in the $100-$200 price range. Note that some chips can be programmed once (OTP) and some parts need to be erased under a UV light before they can be re-programmed (Windowed). CCS offers the Mach X which is a stand-alone programmer and can be used as an in-circuit debugger.

PROTOTYPING BOARDS

There are a large number of Prototyping boards available from a number of sources. Some have an ICD interface and others simply have a socket for a chip that is externally programmed. Some boards have some advanced functionality on the board to help design complex software. For example, CCS has a Prototyping board with a full 56K modem on board and a TCP/IP stack chip ready to run internet applications such as an e-mail sending program or a mini web server. Another Prototyping board from CCS has a USB interface chip, making it easy to start developing USB application programs.

SIMULATORS

A simulator is a program that runs on the PC and pretends to be a microcontroller chip. A simulator offers all the normal debug capability such as single stepping and looking at variables, however there is no interaction with real hardware. This works well if you want to test a math function but not so good if you want to test an interface to another chip. With the availability of low cost tools, such as the ICD in this kit, there is less interest in simulators. Microchip offers a free simulator that can be downloaded from their web site. Some other vendors offer simulators as a part of their development packages.

CCS Programmer Control Software

The CCSLOAD software will work for all the CCS device programmers and replaces the older ICD.EXE and MACHX.EXE software. The CCSLOAD software is stand-alone and does not require any other software on the PC. CCSLOAD supports ICD-Sxx, ICD-Uxx, Mach X, Load-n-Go, and PRIME8.

Powerful Command Line Options in Windows and Linux

· Specify operational settings at the execution level
· Set-up software to perform, tasks like save, set target Vdd
· Preset with operational or control settings for user

Easy to use Production Interface

· Simply point, click and program
· Additions to HEX file organization include associating comments or a graphic image to a file to better ensure proper file selection for programming
· Hands-Free mode auto programs each time a new target is connected to the programmer
· PC audio cues indicate success and fail

Extensive Diagnostics

· Each target pin connection can be individually tested
· Programming and debugging is tested with known good programs
· Various PC driver tests to identify specific driver installation problems

Enhanced Security Options

· Erase chips that failed programming
· Verify protected code cannot be read after programming
· File wide CRC checking

Automatic Serial Numbering Options

· Program memory or Data EEPROM
· Incremented, from a file list or by user prompt
· Binary, ASCII string or UNICODE string

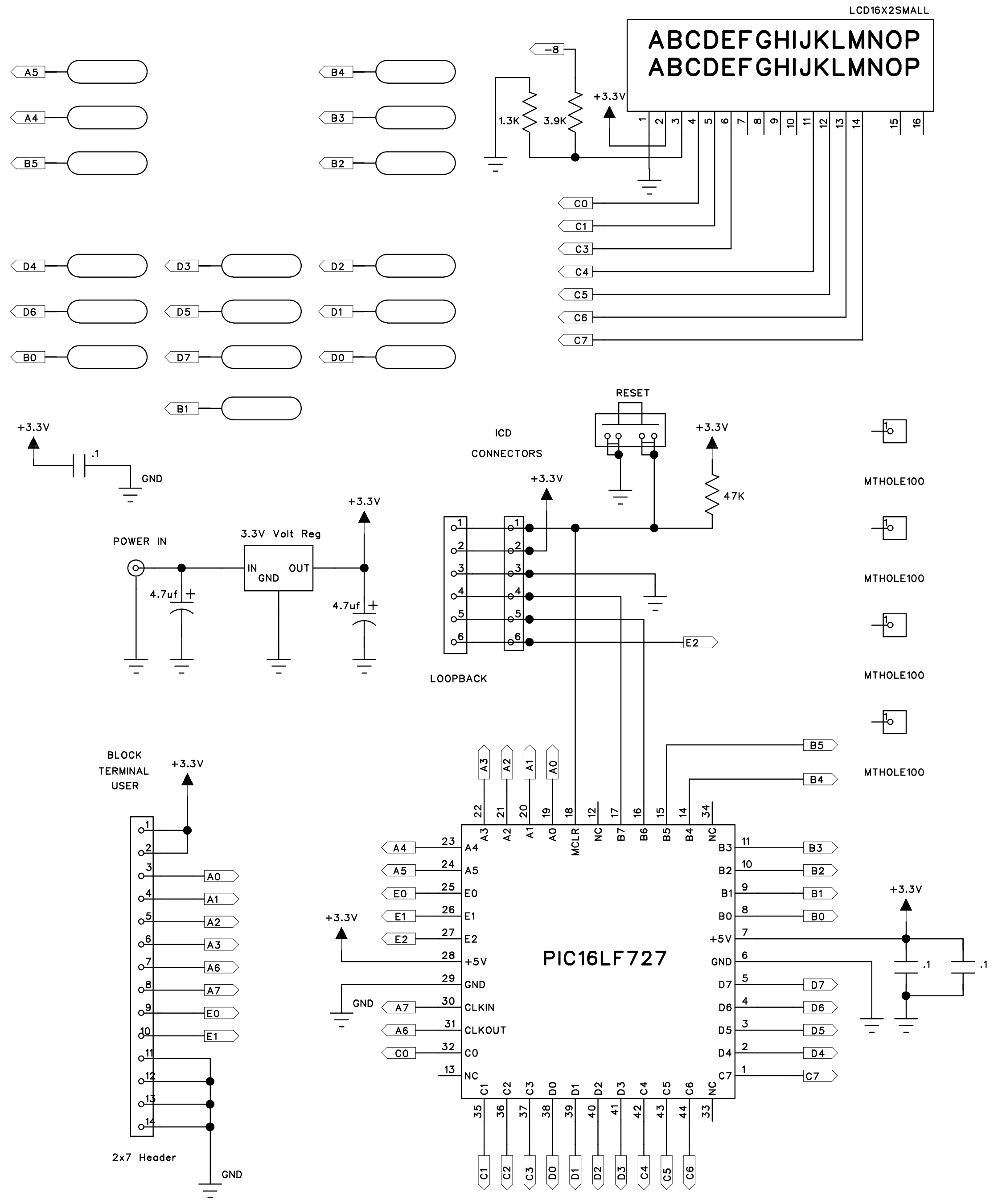
CCS IDE owners can use the CCSLOAD program with:

· MPLAB®ICD 2/ICD 3
· MPLAB®REAL ICE™
· All CCS programmers and debuggers

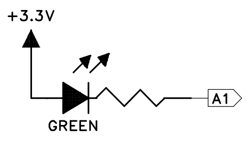
How to Get Started:

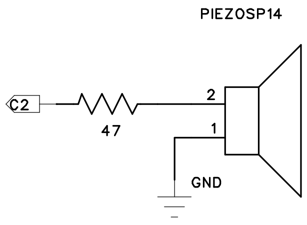
Step 1: *Connect Programmer to PC and target board. Software will auto-detect the programmer and device.*
Step 2: *Select Hex File for target board.*
Step 3*: Select Test Target. Status bar will show current progress of the operation.*
Step 4: *Click "Write to Chip" to program the device.*

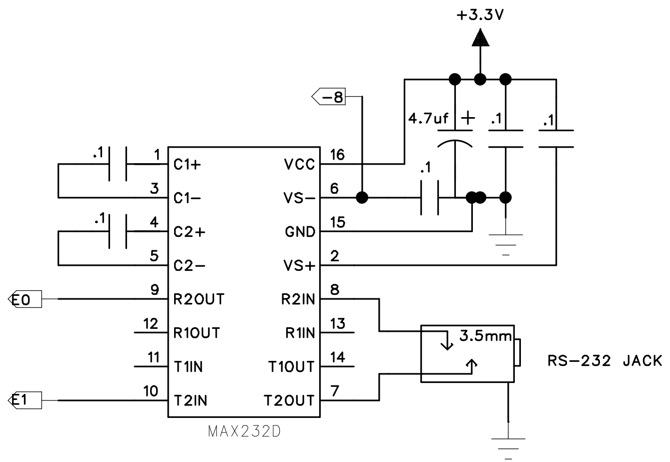
Use the Diagnostics tab for troubleshooting or the ccsload.chm help file for additional assistance.

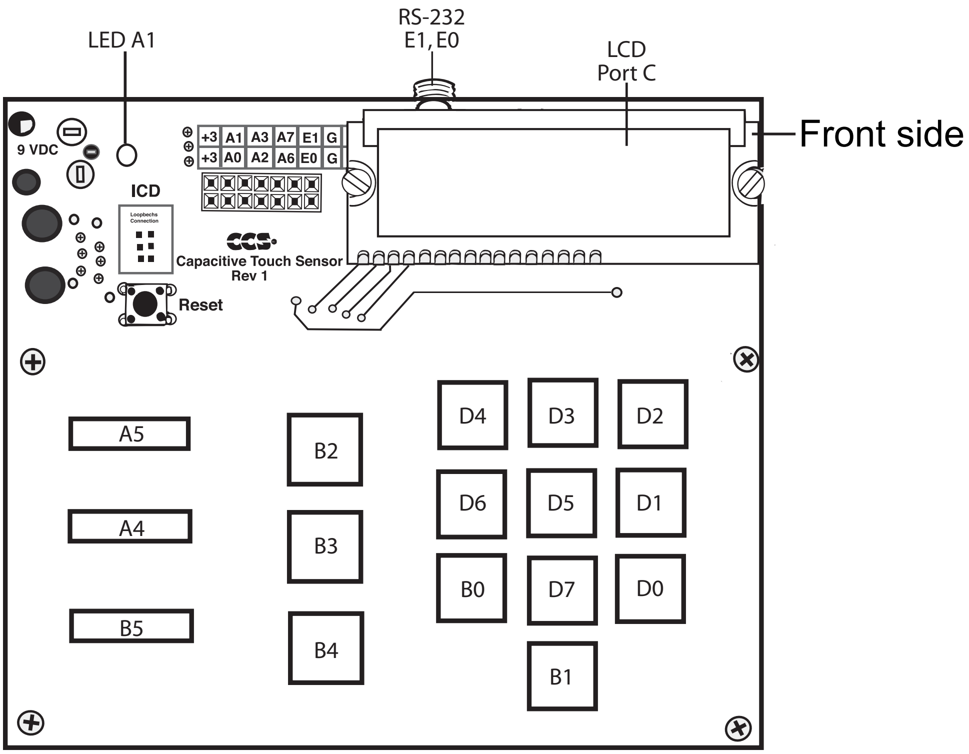


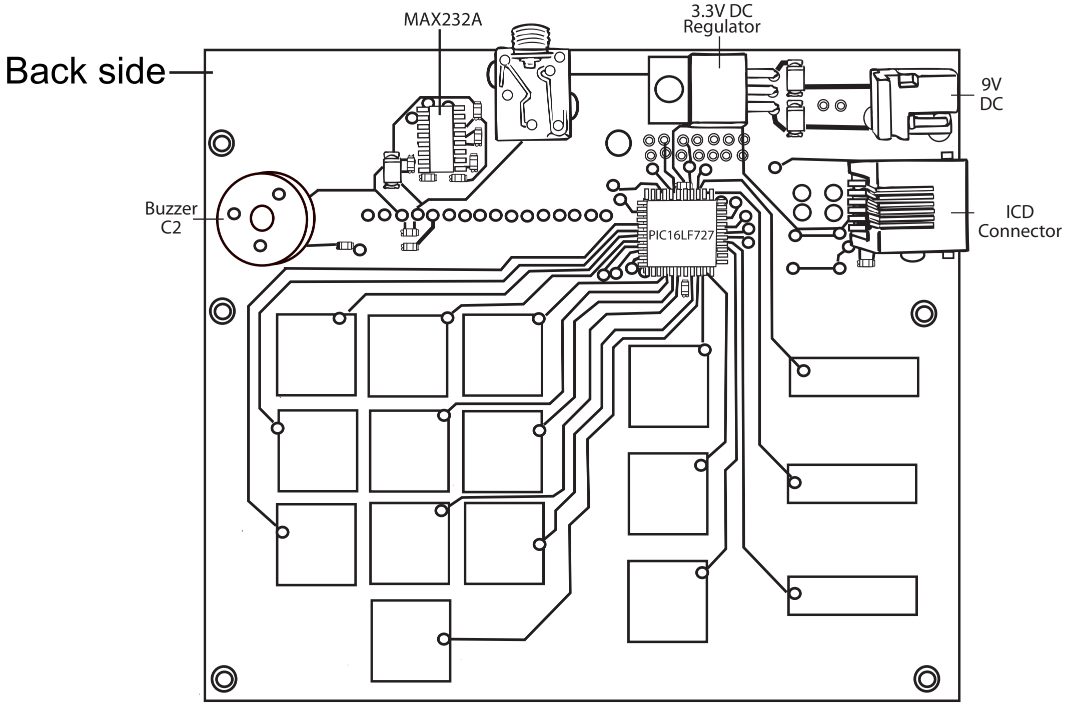
LCD16X2SMALL
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
-8
1.3K
3.9K
+3.3V
C0
C1
C3
C4
C5
C6
C7
A5
A4
B5
B4
B3
B2
D4
D6
B0
D3
D5
D7
B1
D2
D1
D0
+3.3V
.1
GND
POWER IN
3.3V Volt Reg
IN
GND
OUT
4.7uf
4.7uf
+3.3V
ICD
CONNECTORS
LOOPBACK
RESET
47K
E2
MTHOLE100
MTHOLE100
MTHOLE100
MTHOLE100
BLOCK
TERMINAL
USER
2x7 Header
GND
A0
A1
A2
A3
A6
A7
E0
E1
PIC16LF727
A3
A2
A1
A0
MCLR
NC
B7
B6
B5
B4
NC
A4
A5
E0
E1
E2
+5V
GND
CLKIN
CLKOUT
C0
NC
B3
B2
B1
B0
+5V
GND
D7
D6
D5
D4
C7
C1
C2
C3
D0
D1
D2
D3
C4
C5
C6
NC
.1
.1


+3.3V
GREEN
A1


PIEZOSP14
C2
47
2
1
GND


+3.3V
−8
4.7uf
.1
.1
.1
.1
.1
1 C1+
3 C1−
4 C2+
5 C2−
9 R2OUT
12 R1OUT
11 T1IN
10 T2IN
16 VCC
6 VS−
15 GND
2 VS+
8 R2IN
13 R1IN
14 T1OUT
7 T2OUT
E0
E1
MAX232D
3.5mm
RS−232 JACK


LED A1
RS-232
E1, E0
LCD
Port C
Front side
9 VDC
ICD
Loopbacks Connection
+3 A1 A3 A7 E1 G
+3 A0 A2 A6 E0 G
CCS
Capacitive Touch Sensor
Rev 1
Reset
A5
A4
B5
B2
B3
B4
D4
D3
D2
D6
D5
D1
B0
D7
D0
B1


MAX232A
3.3V DC
Regulator
Back side
9V
DC
Buzzer
C2
PIC16LF727
ICD
Connector