Development Kit For the PIC® MCU

Exercise Book

Embedded Serial Busses

March 2010

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

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 Embedded Serial Busses kit. Ensure every item is present.

Software

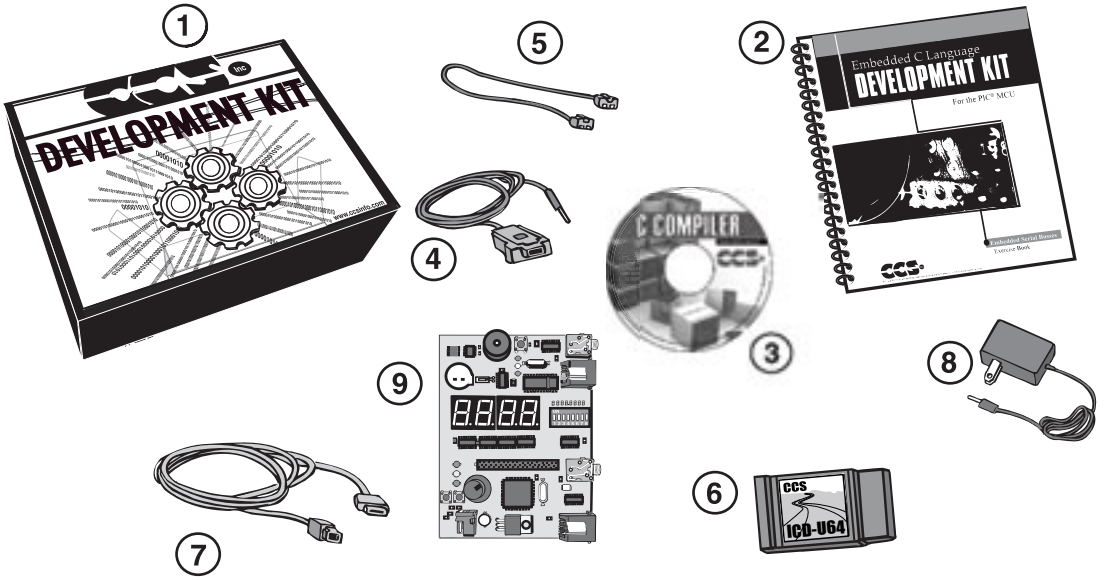
- ❑ Insert the CD into the computer and wait for the installation program to start. If your computer is not set up to auto-run CDs, then select **Start>Run** and enter **My Computer** and double-click on the CD drive.
- ❑ Click on **Install** and use the default settings for all subsequent prompts by 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 (9) to the ICD using the modular cable. Plug in the DC adaptor (8) 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 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 4. 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-U40. 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 PC to ICD cable
8. AC Adaptor (9VDC)
9. Prototyping board for embedded serial busses
   (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

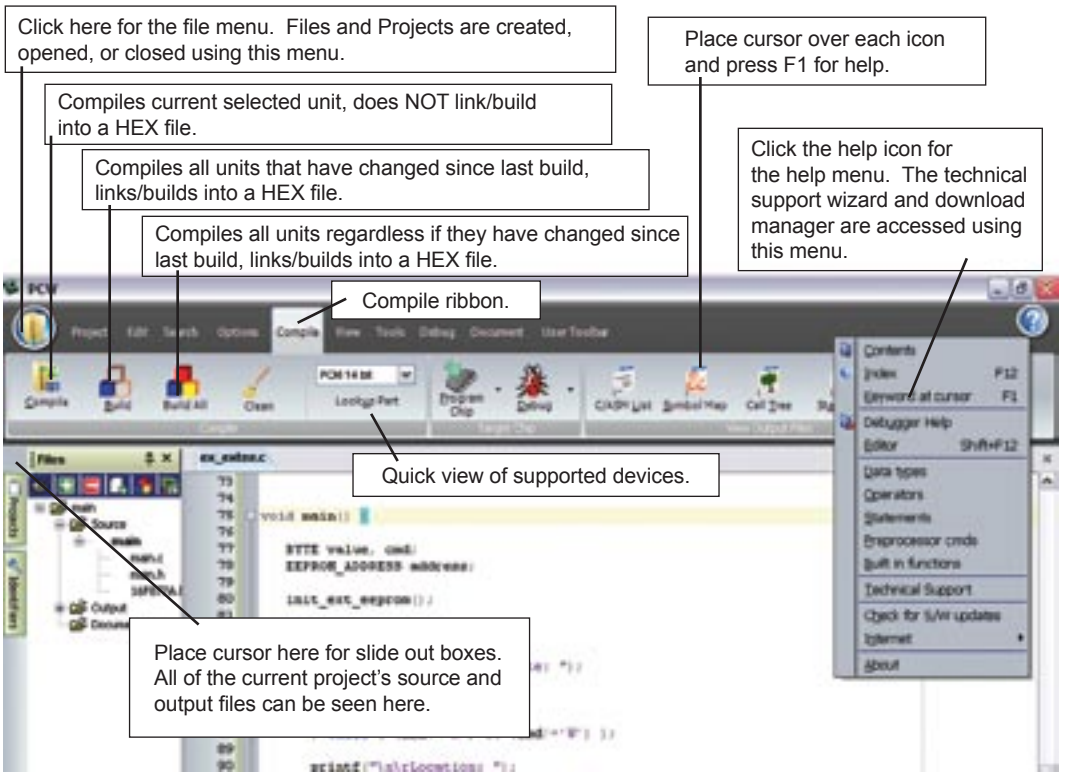
- ❑ Use the drop-down box under Compile to select the compiler. CCS offers different compilers for each family of Microchip parts. All the exercises in this booklet are for the PIC16F877A and PIC16F876A, a 14-bit opcode part. Make sure **PCM 14 bit** is selected in the drop-down box under the **Compiler** 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 Dirs…** 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 micro-controller 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:

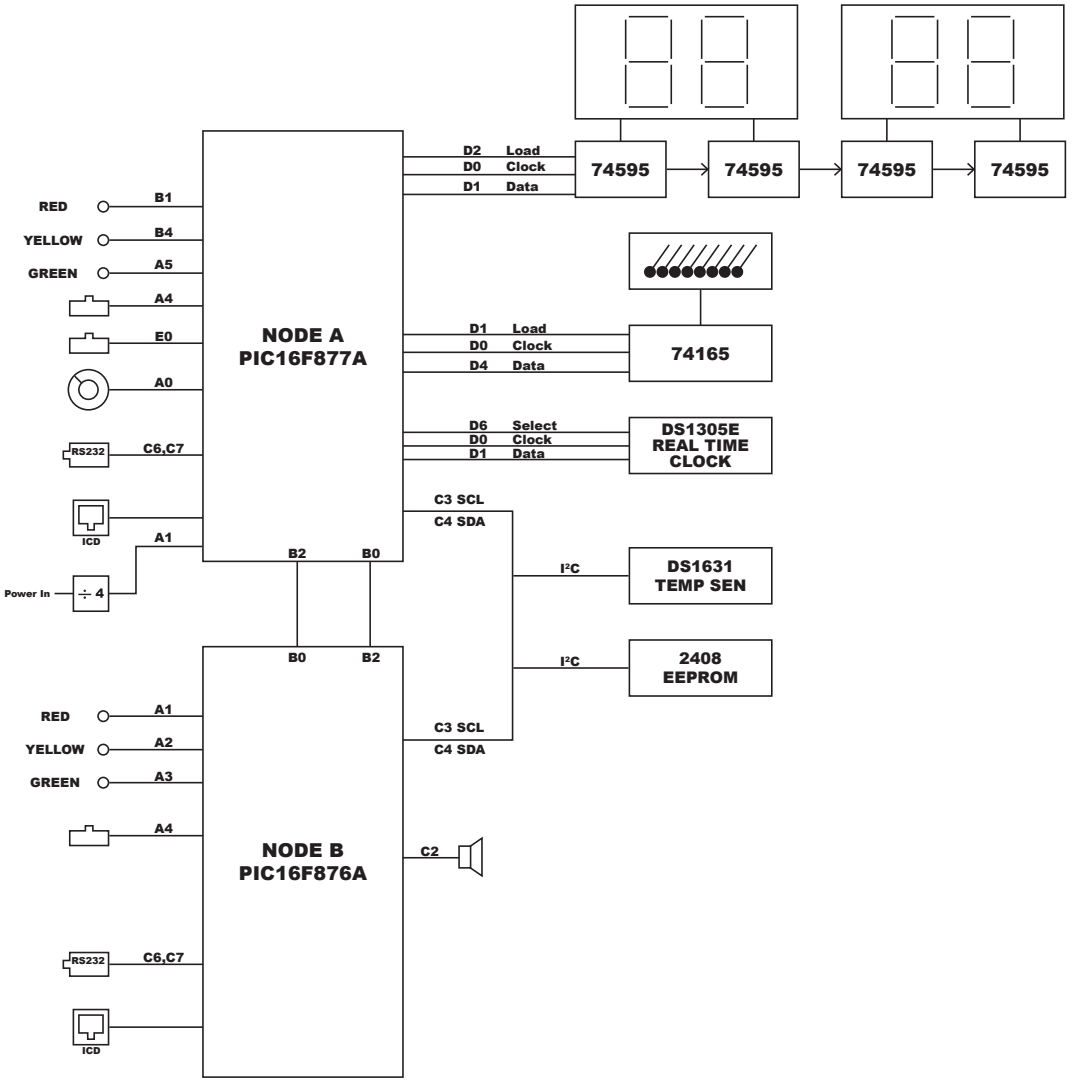
```
int_count=INTS_PER_SECOND;
```

- 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.



3 PROTOTYPING BOARD OVERVIEW

- This development kit includes a prototyping board designed to exercise a number of serial bus protocols. There is a layout in the front cover identifying the connectors. The full schematic is in the back cover. A block diagram of the board is shown on the next page.
- The board has two microprocessors. Node A is a PIC16F877A part and node B is a PIC16F876A part. Each node has an ICD interface for programming, three LEDs, a pushbutton, and RS-232 port. The two nodes are connected to a common I^2C bus and there are two additional wires connecting the two nodes that may be used for communication. One wire is connected to an interrupt pin on each node.
- The I^2C bus also has a digital thermometer chip (DS1631) and EEPROM chip (24LC08). Both parts are I^2C bus slave devices. The two microprocessor nodes can be set up as master/master, master/slave or master/idle.
- The remaining serial devices on the board are set up in a three wire SPI configuration. In this configuration each device shared a common clock. Each device has a dedicated select wire used to identify what device should pay attention to the clock and data. This bus is connected to only node A.
- A real time clock chip (DS1305E) is on the SPI bus.
- The board has eight DIP switches connected to a 74165 chip. This chip is simple parallel to serial shift register and sits on the SPI bus. This allows for 8 additional input lines to the microprocessor using the three wire SPI bus.
- The board has four 7-segment LED units each connected to a 74595 chip. These chips sit on the SPI bus, they are cascaded together and provide together 32 output lines using the three wire SPI bus.
- A piezo buzzer is connected to Node B. This can be used to make software controlled tones.
- The DC voltage into the board is divided down and fed to an ADC input so the input voltage can be measured.


NODE A
PIC16F877A
NODE B
PIC16F876A
RED
YELLOW
GREEN
B1
B4
A5
A4
E0
A0
RS232
C6,C7
ICD
A1
Power In
÷ 4
B2
B0
D2 Load
D0 Clock
D1 Data
74595
74595
74595
74595
D1 Load
D0 Clock
D4 Data
74165
D6 Select
D0 Clock
D1 Data
DS1305E
REAL TIME
CLOCK
C3 SCL
C4 SDA
I²C
DS1631
TEMP SEN
I²C
2408
EEPROM
C2

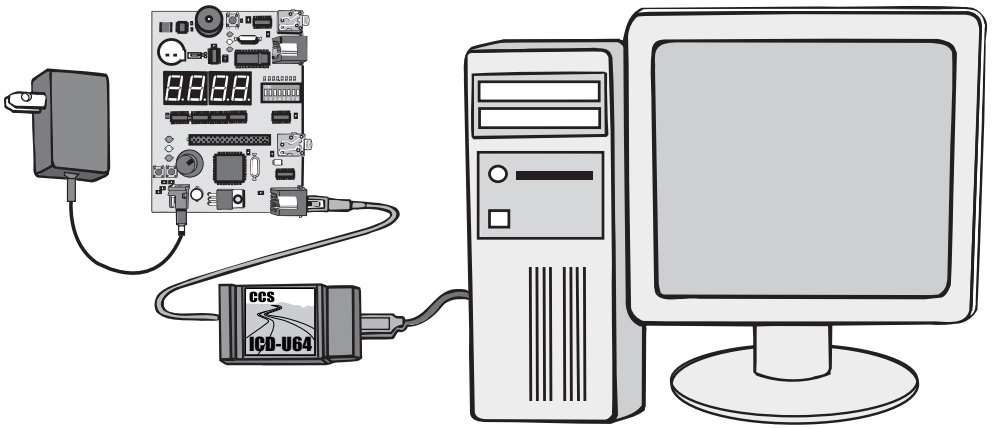
4 COMPILING AND RUNNING A PROGRAM

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>New>Source File** and enter the filename **EX4.C**
- ❑ Type in the following program and **Compile.**

```
#include <16f877A.h>
#device ICD=TRUE
#fuses  HS,NOLVP,NOWDT,PUT
#use delay (clock=10000000)

#define GREEN_LED PIN_A5

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 PIC16F877A, running at 10MHz 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.
- ❑ 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

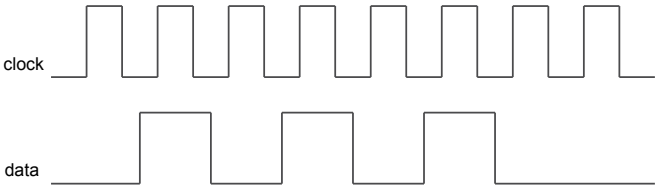
A *Modify the program to light the green LED for 5 seconds, then the yellow for 1 second and the red for 5 seconds.*

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 EX4A.c and EX4B.c and follow the same naming convention throughout this booklet.*

5 SERIAL-TO-PARALLEL SHIFT REGISTERS

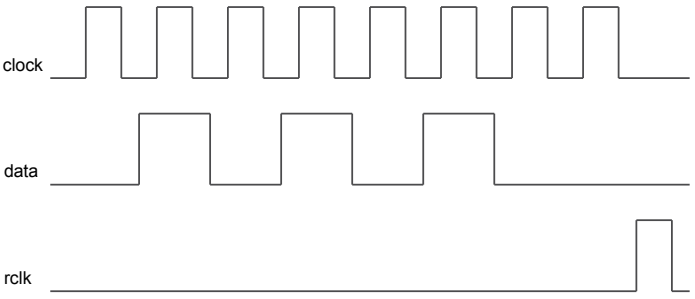
- The 74595 is a serial to parallel shift register. It has eight stages. Each stage has a clock and data input. Data is loaded into the shift register when the clock pin changes from a low to high. The eight shift registers are connected together such that the data line is connected to the first register and that registers latch output is connected to the next stage and so on. For example consider the following waveform:



- This loads the bit pattern 01010100 into the eight registers. To understand the process the following table shows the values in the eight registers after each rising edge of the clock. The left-most bit is the MSB and X represents unknown:

X	X	X	X	X	X	X	0
X	X	X	X	X	X	0	1
X	X	X	X	X	0	1	0
X	X	X	X	0	1	0	1
X	X	X	0	1	0	1	0
X	X	0	1	0	1	0	1
X	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0

- The 74LS595 has a second set of registers that drive the outputs. The data is copied from the first set of registers to the second set when the RCLK pin changes from a low to high. After the shift register has all the data in place one pulse on RCLK will cause the eight outputs to change to the new values at once. This prevents the outputs from changing while data is being shifted in. The waveform with the final pulse looks like this:



- The following code can be used to generate the previous pattern:

```
int data=0x54;

for(i=1;i<=8;++i) {
    output _ bit( SPI _ DATA, shift _ left(&data,1,0));
    output _ high( SPI _ CLOCK );
    output _ low( SPI _ CLOCK );
}
output _ high( SPI _ RCLK );
 output _ low( SPI _ RCLK );
```

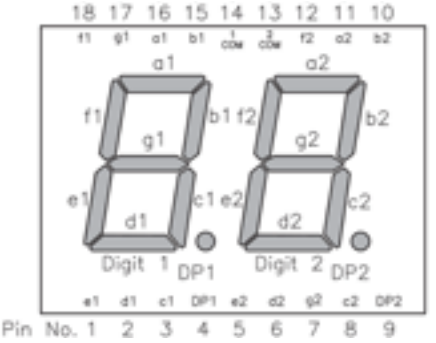
  - The shift_left function shifts data one bit to the left and the value returned from the function is the bit shifted out of the byte (0 or 1).
  - The output_bit function sets a pin (SPI_DATA) high or low depending on the value of the second parameter (the bit shifted out of data).
  - The first bit shifted into the shift register is the MSB and after eight clocks the MSB bit is now in the last shift register (register H).
  - The microprocessor operates much slower than the 74595 so timing is not a problem. Some other devices may require the clock to be high for a certain time or some other time requirement. For these devices add a delay_us(...) at the right spot(s) in the for loop.
- The output from the eighth shift register is provided on a pin of the 74595 allowing any number of 74595 parts to be connected together to make the shift register as long as you want. In this configuration the same clock and RCLK is connected to each 74595. The data is only connected to the first part. The first chips shift out pin (named Q'h) is connected to the second parts data in. This is repeated for each chip. In this configuration eight data bits are clocked in for each chip. For example there are four chips connected together on the prototype board. There will need to be 32 clock pulses to load all four chips. The code now becomes:

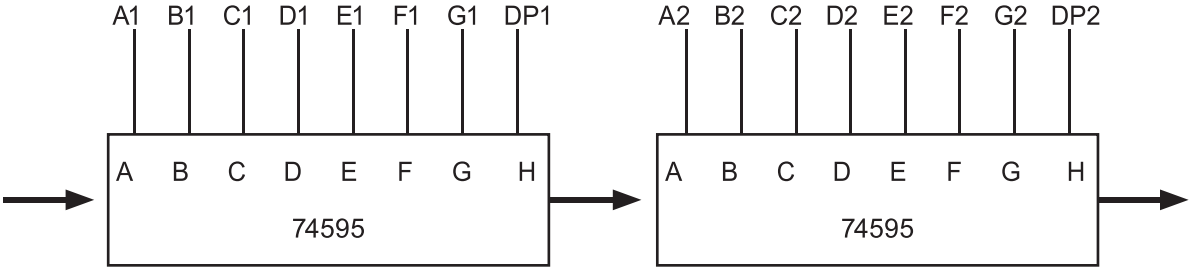
```
int data[NUMBER _ OF _ 74595]={0x54,0x32,0x49,0x10};

for(i=1;i<=NUMBER _ OF _ 74595*8;++i) {
    if((data[NUMBER _ OF _ 74595-1] & 0x80)==0)
        output _ low(SPI _ DATA);
    else
        output _ high(SPI _ DATA);
    shift _ left(data,NUMBER _ OF _ 74595,0);
    output _ high(SPI _ CLOCK);
    output _ low(SPI _ CLOCK);
}
output _ high(SPI _ RCLK);
output _ low(SPI _ RCLK);
```

  - In the above example the last part in the chain will have 0x54. The part with the real data line connected (first in the chain) will have 0x10. This is because array location data[0] is transferred out first. In order to do this the output_bit() could not be used and was replaced with an if statement.

6 APPLICATION WITH 7 SEGMENT LEDS

- The four 7-segment LEDs each have eight control lines (7-segments plus the decimal point) connected to its own 74595. The left most LED module is the first in the chain. The connection of each segment to the 74595 is shown below:





- Included with the C compiler is a driver for the 74595. To use the functions the pins for SPI must be #defined and and array with the data is passed to write_expanded_outputs(). Bit H is the most significant bit of the data.
- The following program will count from 1 to 9999 on the LED display of the prototyping board.
- Enter the following code in a file named **EX6.c**, load and run the program.

```
#include <16F877A.h>
#device ICD=TRUE
#fuses HS, NOLVP, NOWDT, PUT
#use delay(clock = 10000000)

#define EXP_OUT_DO      PIN_D1
#define EXP_OUT_CLOCK   PIN_D0
#define EXP_OUT_ENABLE  PIN_D2
#define NUMBER_OF_74595 4
#include <74595.c>

const byte number[10] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};

void led_display_number(long int data)
{
   int time_array[4];

   time_array[0] = number[data / 1000 % 10];
   time_array[1] = number[data / 100 % 10];
   time_array[2] = number[data / 10 % 10];
   time_array[3] = number[data % 10];

   write_expanded_outputs(time_array);
   output_low(EXP_OUT_ENABLE);
}

void main()
{
   long int time = 0;

   output_low(PIN_D6);              //disable RTC

   while(true)
   {
      led_display_number(time);
      time++;
      delay_ms(1000);
   }
}
```

- Begin a file with a collection of useful functions. Call the file ESBProto.c and put in it the #defines, then #include <74595.c>, and led_display_number().

7 DEBUGGING

- Using the code from chapter 6 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 `led _ display _ number(time);` line and notice that one click executed the entire function. This is the way step over works. Click step over on the next two lines, and wait for the delay to finish.
- Click the **Watch** tab, then the add icon  to add a watch. Enter **time or choose time the variables from list**, then click **Add Watch**. Notice the value shown. Continue to step over through the loop a few more times (wait as required) and notice the time watch increments.
- Step over until the call to `led _ display _ number(time);` 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. Watch the 7-segment LED's to see the program run. Click the stop icon  to halt execution.
- In the editor, click on `led _ display _ number(time);` 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.
- 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-S 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 start a new file **EX7.c** as follows:

```
#include <ESBProto.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 is executed. Add a watch for c and 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;

B *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.

8 PARALLEL-TO-SERIAL SHIFT REGISTER

- The eight DIP switches on the prototyping board are connected to a 74165 parallel to serial shift register. This part takes the eight inputs and with each clock pulse shifts them out bit for bit to the data pin. When the Shift/Load pin on the part is high then the data line output is high impeadance. When the pin goes low then the data is latched and shifted out each time the clock goes high.

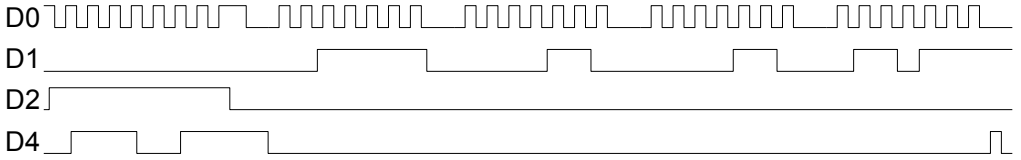
- The code to read a 74165 looks like this:

```
output_high(SPI_ENABLE);
output_low(SPI_ENABLE);                              // Latch all inputs
output_high(SPI_ENABLE);                             // Enter shift mode

for(i=1;i<=8;++i) {
    shift_left(&data,1,input(SPI_DATA));
    output_low(SPI_CLOCK);
    output_high(SPI_CLOCK);
}
output_low(SPI_ENABLE);
```

- The following program reads the eight switches, uses the eight positions as an eight bit number then displays the number of the LED display. Enter the following code in a file named **EX8.c**, load and run the program.

```
#include <ESBProto.c>

#define EXP_IN_ENABLE              PIN_D1
#define EXP_IN_CLOCK               PIN_D0
#define EXP_IN_DI                  PIN_D4
#define NUMBER_OF_74165 1
#include <74165.c>

void main()
{
    int dips;

    while(true)
    {
        read_expanded_inputs(&dips);
        led_display_number(dips);
        delay_ms(100);
    }
}
```

- In this diagram, the PIC16F877A reads the number 0x73 from the DIP switches, and then outputs 0x3F06066D to display the decimal number 0115 on the LEDs. The 74165 uses pin D4 for data, while the 74595s use pin D1.




```
#define EXP_OUT_DO      PIN_D1
#define EXP_OUT_CLOCK   PIN_D0
#define EXP_OUT_ENABLE  PIN_D2
```

9 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 code from Chapter 8 after including the **ESBProto.c** file:

```
#use RS232(DEBUGGER, xmit = PIN_B5, rcv = PIN_B5)
```

- Save the code from section 8 and name it **EX9.c**. Add the following line of code after the call to read_expanded_inputs (&dips);

```
printf("read dips: %X\r\n", dips);
```

- 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.
- Set a breakpoint on the line:

```
printf("read dips: %X\r\n", dips);
```

- Click the debugger **Break Log** tab, check the LOG box, set the breakpoint as 1 and expression as **dips**.
- Click GO, then click the **Log** tab and notice that each time the breakpoint was hit the value of the **dips** 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.
- Uncheck the LOG box under the log tab.
- Enter a watch for **dips**.
- Click GO 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** and name it **EX9.TXT**.
- 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** 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.
- ❑ Use **File>Open>Any File** to find the file **EX9.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 **led_display_number(dips);**.
- ❑ Click Reset then Go.
- ❑ 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 0.2ms. Multiply your number of ticks by 0.2 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 PIC16F877A 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.*

10 3-WIRE SPI BUS REAL TIME CLOCK AND RS-232

- The Real time clock chip on the prototyping board (DS1305) has a three wire interface as well.
- Unlike the previous three wire interfaces the RTC chip has a more complex protocol on top of the basic byte transfer. All transfers begin with a write to the RTC of 8 bits. The low 7 bits represent a register address in the part. If the top bit is 0 then this is a read request. If the top bit is 1 then this is a write.
- As an example register #1 is the minutes register. Sending a 0x81 followed by a 0x02 will set the minutes to 2. Sending a 0x01 followed by a read will return the current minutes.
- Many of the registers in the DS1305 are in BCD format. Numbers in the microprocessor are stored in binary format. BCD treats each base 10 digit as 4 bits in binary. For example the number 35 in binary is 00100011. In the BCD format it is 0011 0101. Numbers going to and from the DS1305 must be converted. BCD conversion functions as well as the basic transfer functions are in the DS1305.C include file.
- The following is a table of all the DS1305 registers.

HEX ADDRESS		Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	RANGE
READ	WRITE									
00H	80H	0	10-SEC			SEC				00–59
01H	81H	0	10-MIN			MIN				00–59
02H	82H	0	12	P	10-HR	HOURS				01–12 + P/A
				A						00–23
			24	10						
03H	83H	0	0	0	0	DAY				1–7
04H	84H	0	0	10-DATE		DATE				1–31
05H	85H	0	0	10-MONTH		MONTH				01–12
06H	86H	10 YEAR				YEAR				00–99
—	—	ALARM 2								—
07H	87H	M	10-SEC ALARM			SEC ALARM				00–59
08H	88H	M	10-MIN ALARM			MIN ALARM				00–59
09H	89H	M	12	P	10-HR	HOUR ALARM				01–12 + P/A
				A						00–23
			24	10						
0AH	8AH	M	0	0	0	DAY ALARM				01–07
—	—	ALARM 1								—
0BH	8BH	M	10-SEC ALARM			SEC ALARM				00–59
0CH	8CH	M	10-MIN ALARM			MIN ALARM				00–59
0DH	8DH	M	12	P	10-HR	HOUR ALARM				01–12 + P/A
				A						00–23
			24	10						
0EH	8EH	M	0	0	0	DAY ALARM				01–07
0FH	8FH	CONTROL REGISTER								—
10H	90H	STATUS REGISTER								—
11H	91H	TRICKLE CHARGER REGISTER								—
12–1FH	92–9FH	RESERVED								—
20–7FH	A0–FFH	96 BYTES USER RAM								00–FF

- The basic transfer of data to the DS1305 uses similar code as in Chapters 5 and 8.
- The functions write_ds1305() and read_ds1305() in ds1305.c perform these functions. To set the hour register (0x82) to 5 you can do this:

  write_ds1305(0x82, 5);

  The ds1305.c driver also has functions like rtc_get_date() to get all the date values.
- Enter the code and save as **EX10.c** load and run the following program to demonstrate the real time clock interface.
- Note that this program uses the debugger MONITOR window to communicate with the user. This program allows you to set and read the clock. Be sure to set the correct time before moving to the next chapter.

```
#include <ESBProto.c>
#use rs232(debugger, xmit=PIN_B5, rcv=PIN_B5)

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

#define RTC_SCLK PIN_D0
#define RTC_IO   PIN_D1
#define RTC_RST  PIN_D6
#include <ds1305.c>

void main()
{
  int dow, day, month, year, hour, minute, second, oldsec;

  output_high(PIN_D2);                                    //turn off displays
  rtc_init();

  printf("\r\nSet date and time? (y/n) ");
  if(kbhit())
    getc();                                               //clear read buffer
  if(getc() == 'y')
  {
    printf("\r\nMonth: ");
    month = get_int();
    printf("\r\nDay: ");
    day = get_int();
    printf("\r\nYear: ");
    year = get_int();
    printf("\r\nDay of week (1=Sunday, 7=Saturday): ");
    dow = get_int();
    printf("\r\nHour: ");
    hour = get_int();
    printf("\r\nMinute: ");
    minute = get_int();
    rtc_set_datetime(day, month, year, dow, hour, minute);
  }
  while(true)
  {
    rtc_get_time(hour, minute, second);
    if(second != oldsec)
    {
      rtc_get_date(day, month, year, dow);
      printf("\r\n %u/%u/%02u [%u] %u:%02u:%02u",
        month, day, year, dow, hour, minute, second);
      oldsec = second;
    }
  }
}
```

FURTHER STUDY

A Use the functions already developed to write a stand-alone program that dispalys the time HHMM on the four LEDs. The program should update the LEDs like a clock.

11 I²C OVERVIEW

- The I2C bus is a simple two wire multi-drop serial protocol. The I2C concept was developed by Philips for comunication with consumer electronics IC chips. The advantages over the previous serial bus examples is that I2C does not require a third wire for device selection. All of the system devices can share the same two wires. In I2C the two wires are identified as SCL (clock) and SDA (data).
- Each device has a unique address. The standard I2C has a 7 bit address (1-127) and there is an extended protocol that allows for a 10 bit address (1-1023). Some devices allow the designer to select the address by grounding or pulling high address lines on the part. This allows a number of identical parts to be on the same bus. For example a designer may put three 8K serial EEPROMs on the same bus to get 24K. Each device will have a different address set in hardware.
- The bus operates in a master/slave mode where a master device controls the clock line and decides what will happen on the bus. The protocol allows for multiple masters on the same bus and defines a method for the masters to stay out of each other way.
- The protocol has defined two special conditions START and STOP that can be initiated on the bus.
  A START condition looks like this:

  SDA
  SCL

  A STOP condition looks like this:

  SDA
  SCL

  A normal data bit is transfered like this:

  SDA
  SCL

  Shown is a transfer of 1, the SDA changes while the clock is low and the receiver grabs the data when the clock transitions high. Note that for START and STOP the clock line is high while the SDA transitions. This allows the slave to distinguish data from control.
- After a START the master will issue a slave address and one bit to indicate if the data transfer is from slave to master or if it is from master to slave. These two items are packed into a 8 bit byte. The direction is the LSB. After that, any number of data bytes are transferred and a STOP is issued. If the data direction needs to change, the master issues another START (no STOP needed) and a new address/direction byte.
- Each byte transferred over the bus has eight data bits and one ACK bit. The master issues the 9th clock and the device receiving data (may be master or slave) will pull the data line low if the byte was received. If the 9th bit is high, then the receiver is not acknowledging the data. There is no retry defined. The master should issue a STOP and determine how to best proceed. Note that some devices will withhold an ACK in order to stop transmission normally. For example, a serial EEPROM device may keep sending bytes sequentially through memory until a byte is not ACKed.
- Some simple devices have a small number of predefined registers that are always read or written in order. For example a master would write to these registers as follows:

  START
  WRITE address/0 (0 is a master write)
  WRITE register A
  WRITE register B
  WRITE register C
  STOP

And read like this:

START
WRITE address/1 (1 is a master read)
READ register A
READ register B
READ register C
STOP

- Other devices allow the registers to be addressed and the master must set the register address. For example they write like:

START
WRITE device_address/0
WRITE register_address
WRITE register_value
STOP

And read like this:

START
WRITE device_address/0
WRITE register_address
START
WRITE device_address/1
READ register_value
STOP

- To allow multiple writers on the BUS the I2C bus has external pull-up resistors on both SCL and SDA. Devices on the bus either drive the pin low or allow it to float high.
- During data transfer a slave device can slow down the master device if the slave is not ready by holding the clock line low. The master will detect that the clock did not float high and wait for the slave. This is called clock stretching.
- The C compiler has built-in functions to control the bus. These functions require the following directive:

#USE I2C(MASTER, SCL=PIN_B0, SDA=PIN_B1, FAST)

The last parameter sets the bus speed. The I2C standard defines two speeds FAST (400khz) and SLOW (100khz).

- The following functions may be used after the above directive defined the bus:

I2C_START()
I2C_STOP()
ack = I2C_WRITE()
data = I2C_READ()

FURTHER STUDY

A Review the 2432 serial EEPROM data sheet and write a simple function to write a given value to a given address in the serial EEPROM using the compiler functions. Compare your function to the function in 2432.C.

12 I²C MASTER

- The following program sets up an I2C bus with the PIC16F877A chip as the master, and the temperature chips and serial EEPROM chips as slaves. Note that the temperature chip data sheet shows the address as:

  1 0 0 1 A2 A1 A0 RW

  Since the A0,A1,A2 pins are grounded the address is 0x90. Sometimes this address is expressed as 0x48 (ignoring the RW bit). Make sure you understand how the address is given. The compiler uses the 0x90 format (real address times 2). The serial EEPROM is address 0xA0.

- The program shows the temperature in the first two digits and the serial EEPROM data in location 0 in the last two digits. The serial EEPROM data does not change. You can press your finger on the temperature chip to change the temperature.

- Enter the following code in a file named **EX12.c**, load and run the program.

```
#include <ESBProto.c>
#use i2c(master, scl=PIN_C3, sda=PIN_C4)

void main()
{
  long int temp, display;
  int read_data, temp_high, temp_low;

  i2c_start();
  i2c_write(0x90);
  i2c_write(0xAC);
  i2c_write(0x0C);                          //write config register
  i2c_start();
  i2c_write(0x90);
  i2c_write(0x51);                          //start temperature conversions
  i2c_stop();

  delay_ms(750);                            //time before first valid temperature reading

  while(true)
  {
    i2c_start();
    i2c_write(0x90);
    i2c_write(0xAA);
    i2c_start();
    i2c_write(0x91);
    temp_high = i2c_read(1);
    temp_low = i2c_read(0);
    i2c_stop();

    i2c_start();
    i2c_write(0xA0);
    i2c_write(0x00);
    i2c_start();
    i2c_write(0xA1);
    read_data = i2c_read(0);
    i2c_stop();
```

(continued...)

(continued...)

```
    //convert 12-bit celcius to 2-digit fahrenheit
    temp = ((long)temp_high << 4) + (temp_low >> 4);
    temp = temp * 9 / 5;
    display = (temp >> 4) + 32;
    display = display * 100 + read_data;
    led_display_number(display);
  }
}
```

- Note that you can pass a 0 to I2C_READ() if you do not want to ACK the byte. This is done with the serial EEPROM to indicate the master is done receiving data
- The compiler comes with simple drivers for many chips including the DS1631 and 2408. The following is a rewrite of the same program using the compiler drivers.

```
#include <ESBProto.c>

#define DAL_SDA PIN_C4
#define DAL_SCL PIN_C3
#include <ds1631.c>

#define EEPROM_SDA PIN_C4
#define EEPROM_SCL PIN_C3
#include <2408.c>

void main()
{
  long int display;
  int read_data;

  init_ext_eeprom();
  init_temp();

  delay_ms(750);                          //time before first valid temperature reading

  while(true)
  {
    display = read_full_temp();
    read_data = read_ext_eeprom(0);
    display = display / 100 * 100 + read_data;
    led_display_number(display);
  }
}
```

FURTHER STUDY

A The above program sets the DS1631 chip to continuously update the temperature registers. The data sheet describes another mode where the chip only reads the temperature when commanded. This is used for low power applications. Redesign the program to set the chip up in one-shot mode, start a conversion and read the result each time through the loop.

13 I²C - MULTI MASTER

- The chapter 12 program had a single master on the I2C bus. To add another master, the code needs to be modified so each master checks to see if the bus is in use. This is done in the compiler functions by looking at the I2C_WRITE() result. Normally this result is 0 (ACK) or 1 (NACK). In a multi-master mode it can also return a 2 if two masters attempt to use the bus at the same time.

- The simple code to read location 0 from the serial EEPROM converts the following for multi-master. Note that the C continue and break statements have been used to improve the readability.

```
do {
    i2c_start();
    if( i2c_write(0xa0) !=0)
        continue;
    if( i2c_write(0) !=0)
        continue;
    i2c_start();
    if( i2c_write(0xa1) !=0)
        continue;
    data=i2c_read(0);
    i2c_stop();
    break;
} while(TRUE);
```

- For these examples the compiler includes multi-master versions of some drivers. Use DS1631MM.C and 2408MM.C for multi-master versions of the drivers.

- Modify the prgram from chapter 12 to use these multi-master drivers. Load the prgram onto the PIC16F877A.

- Create the following program for the PIC16F876A chip. This program will increment the value in location 0 of the serial EEPROM every time the button is pressed. Load and run the program on the target board to demonstrate two masters on the I2C bus. The last two digits should increment each time the button is pressed.

```
#include <16F876A.h>
#fuses HS,NOLVP,NOWDT,PUT
#use delay(clock = 20000000)
#use RS232(baud = 9600, xmit = PIN_C6, rcv = PIN_C7)

#define EEPROM_SDA PIN_C4
#define EEPROM_SCL PIN_C3
#include <2408mm.c>

void main()
{
  long int data;

  init_ext_eeprom();

  while(true)
  {
    while(input(PIN_A4))
      data = read_ext_eeprom(0);

    while(!input(PIN_A4));
    data = read_ext_eeprom(0);
    if(data >= 99)
      data = 0;
    else data++;
    write_ext_eeprom(0, data);
  }
}
```

FURTHER STUDY

A Modify the above PIC16F876A program to keep a count of how many times the I2C bus access failed. Then modify the program to write to location 0 of the EEPROM the value of the counter every 2 seconds. Hint: Incrementing the count will require modifications to the driver file.

14 I²C - PIC SLAVE

- The previous programs all use the PICs as bus masters. In this chapter we will design an I²C slave program. It is always best to use interrupts to handle slave requests.
- The function I2C_ISR_STATE() determines why the I²C interrupt happened. It returns 0 when the address byte is recieved. It returns a 1..127 when the master sends data bytes 1 to 127. It returns 128..255 when the master requests data bytes 1 to 127.
- Enter and load the following example slave program in the PIC16F876A chip.

```
#include <16F876A.h>
#fuses HS,NOLVP,NOWDT,PUT
#use delay(clock = 20000000)
#use i2c(slave, scl=PIN_C3, sda=PIN_C4, address=0x70)

#define red_led PIN_A1
#define yellow_led PIN_A2
#define green_led PIN_A3

#int_ssp
void ssp_isr()
{
  char c;
  int state;

  state = i2c_isr_state();

  c = i2c_read();

    if(state==1)
    if(c == 'r')
      output_toggle(red_led);
    else if(c == 'y')
      output_toggle(yellow_led);
    else if(c == 'g')
      output_toggle(green_led);
  }
}

void main()
{
  enable_interrupts(int_ssp);
  enable_interrupts(global);
  while(true);
}
```

❑ Enter and load the following example program into the PIC16F877A chip.

```
#include <ESBProto.c>
#use i2c(master, scl=PIN_C3, sda=PIN_C4)

void main()
{
  char c;

  output_high(PIN_D2);                              //turn off displays

  while(true)
  {
    printf("\r\nPick a color (r,y,g): ");
    do
    {
      c = getc();
      putc(c);
    }while(!(c == 'r' || c == 'y' || c == 'g'));

    i2c_start();
    i2c_write(0x70);
    i2c_write(c);
    i2c_stop();
  }
}
```

FURTHER STUDY

A Write a slave program to return the value of the push button when the master requests data. Program the master to read the slave push button value and display it in the LEDs.

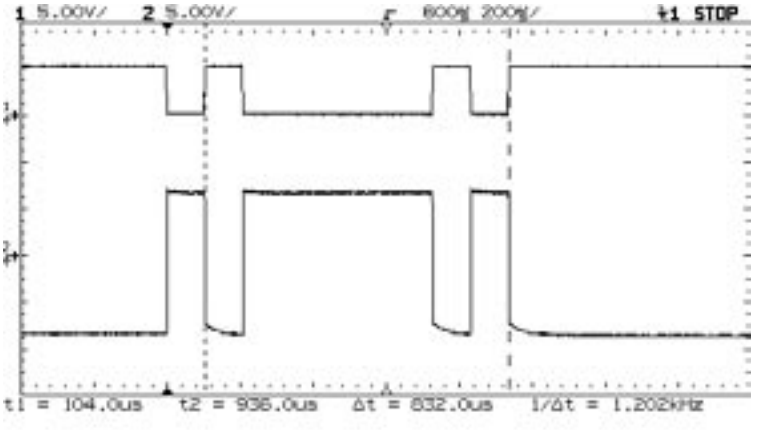
15 ASYNCHRONOUS SERIAL BUS

- The prior programs all have a clock to indicate when the data is to be sampled. This is referred to as synchronous communication. Asynchronous communication only sends data and relies on the receiver to reconstruct the clock. This is done by the receiver starting a bit timer when a byte starts. For this to work the first bit must always be opposite the idle state. A start bit is inserted for this purpose. To help ensure proper transmission a stop bit of the opposite polarity of the start bit is inserted at the end. For this to work, each bit must be a fixed time and both receiver and transmitter must know that time. For example to send a 0x12 byte when the idle state of the bus is a 1 the following is sent:

  0 0 1 0 0 1 0 0 0 1

  Notice the data is sent LSB first. In all it takes 10 bit times to send one byte.

- The compiler has a set of built-in functions brought in with #USE RS232 for asynchronous communication. The baud= option specifies how many bits are sent per second. The bit stream, as specified above, is a start bit (always 0), Eight data bits (lsb first) and a stop bit (always 1). The line then remains at the 1 level. The number of bits may be changed with a bits= option and a parity bit can be added before the stop bit with a parity= option. The following diagram shows a single character A (01000001) as sent at 9600 baud. The eight data bits are between the dotted lines. Each bit is 104us.

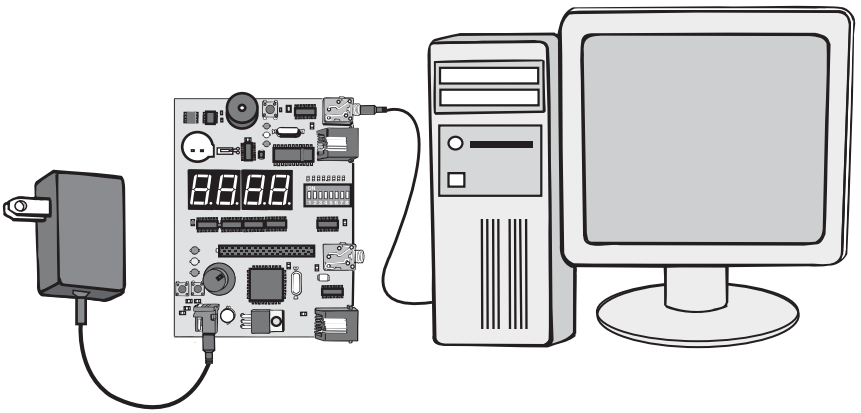


- The basic functions for RS-232 are putc() and getc().
- getc() will cause the program to stop and wait for a character to come in before it returns.
- printf calls putc() repeatedly to output a whole string and format numbers if requested. The % in the printf indicates another parameter is included in the printf call and it should be formatted as requested. %lu indicates to format as an unsigned long.

- When a level converter is used to convert the microprocessor 0-5V output to a +/- voltage then the asynchronous communication is RS-232 compliant. At the RS232 voltages the signal can travel farther. RS232 ports are on many computers. A MAX232 level converter is on each of the PICs on the protoboard.

- The following is a simple RS232 program and a hookup diagram.

```
#include <ESBProto.c>
#use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7)

void main()
{
  output_high(PIN_D2);                                    //turn off displays

  printf("\r\nprintf sends serial data to STDOUT\r\n");
  while(true)
    putc(getc());
}
```



16 MATH CO-PROCESSOR

- Asyncronous communication has the advantage of only needing one wire for each receiver/ transmitter pair. The examples in this chapter show how a simple two wire asyncronous communication path can be set up between microprocessors.
- We will use the PIC16F876A chip as a math co-processor. This chip will perform floating point math functions that require ample ROM. This way the main program in the PIC16F877A chip does not need to waist space for the floating point library.
- The PIC16F877A chip will send a request to the 876A chip over the serial interface. The PIC16F876A chip will send the result back over the serial interface.

```
#include <ESBProto.c>
#use rs232(stream=out, baud=9600, xmit=PIN_C6, rcv=PIN_C7)
#use rs232(stream=chip, baud=9600, xmit=PIN_B0, rcv=PIN_B2)
#include <stdlib.h>
#include <input.c>

void flop(byte *a, char op, byte *b, byte *result)
{
  int i;
  for(i=0;i<4;i++)
    fputc(a[i], chip);                    //send four bytes for float
  fputc(op, chip);
  for(i=0;i<4;i++)
    fputc(b[i], chip);
  for(i=0;i<4;i++)
    result[i]=fgetc(chip);
}

void main()
{
  float a, b, result;
  char op;

  output_high(PIN_D2);                    //turn off displays
  delay_ms(10);                           //ignore noise and synchronize

  while(true)
  {
    fprintf(out, "\r\n\r\nEnter first number: ");
    a = get_float();
    fprintf(out, "\r\nEnter operator (+-*/): ");
    op = fgetc(out);
    fputc(op, out);
    fprintf(out, "\r\nEnter second number: ");
    b = get_float();

    flop(&a, op, &b, &result);
    fprintf(out, "\r\nResult is: %12f", result);
  }
}
```

```
#include <16F876A.h>
#fuses HS,NOLVP,NOWDT,PUT
#use delay(clock = 20000000)
#use rs232(stream=chip, baud=9600, xmit=PIN_B0, rcv=PIN_B2)
#use rs232(stream=out, baud=9600, xmit=PIN_C6, rcv=PIN_C7)
#include <stdlib.h>
#include <input.c>

void get4(byte *data)
{
  int i;
  for(i=0;i<4;i++)
    data[i] = fgetc(chip);
}

void put4(byte *data)
{
  int i;
  for(i=0;i<4;i++)
    fputc(data[i],chip);
}

void main()
{
  float a, b, result;
  char op;

  delay_ms(9);                                    //ignore noise and synchronize

  while(true)
  {
    get4(&a);
    op = fgetc(chip);
    get4(&b);
    switch(op)
    {
      case '+':
        result = a + b;
        break;
      case '-':
        result = a - b;
        break;
      case '*':
        result = a * b;
        break;
      case '/':
        result = a / b;
        break;
      default:
        result = 99;
        break;
    }
    put4(&result);
  }
}
```

17 ADDITIONAL PROJECTS

Part A - Piezo Speaker

- The embedded busses prototyping board has a piezo speaker on it. Using the pulse width modulation module built into the PIC16F876A the following code will sound a tone at 1.22khz for one second.

```
setup_ccp1(CCP_PWM);

setup_timer_2(T2_DIV_BY_16, 255, 1);
set_pwm1_duty(127);
delay_ms(1000);
setup_timer_2(T2_DISABLED, 0, 1);
```

Part B - Analog to Digital Converter

- There is a potentiometer on the PIC16F877A node that can be read as follows. Add the directive #device adc=10 at the top of your code. Expect a range of 0 (far left) to 1023 (far right).

```
long int a;

setup_adc_ports(AN0);
setup_adc(ADC_CLOCK_INTERNAL);
set_adc_channel(0);
while(TRUE) {
   a=read_adc();
   led_display_number(a);
}
```

- The 9V input power is divided down and fed to an A/D pin. This may be used to measure the input voltage to the card and even know the AC wall voltage since the wall transformer divides down the wall voltage by a factor of approximately 11.
- Write a program to display the wall voltage on the LEDs.

Part C - Clock

- ❑ The PIC16F877A node has two pushbuttons that can be read.
- ❑ Modify the stand-alone clock program to allow the clock time to be set by the two pushbuttons (fast advance and slow advance).
- ❑ Add an alarm clock feature. Set the alarm time with the potentiometer. When the potentiometer moves, display the alarm time until the potentiometer stops moving for 5 seconds. To make it easy to set, change the time in 15 min increments. Sound the piezo buzzer until a pushbutton is depressed.

Part D - Happy Birthday

- ❑ Modify the EX_TONES.C program to play happy birthday through the piezo speaker.

Part E - EEPROM Access

- ❑ Design two programs (node A & B) to access a location in the external EEPROM as fast as possible. The A node program should record the number of collisions and display the data on the LED. The RED LED should light if the data is read wrong. This would indicate a undetected collision.

Part F - Baud Rate Speed

- ❑ Modify the example 14 programs to increase the baud rate to see how fast you can go.

References

This booklet is not intended to be a tutorial for 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^2C 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 PICmicro® 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

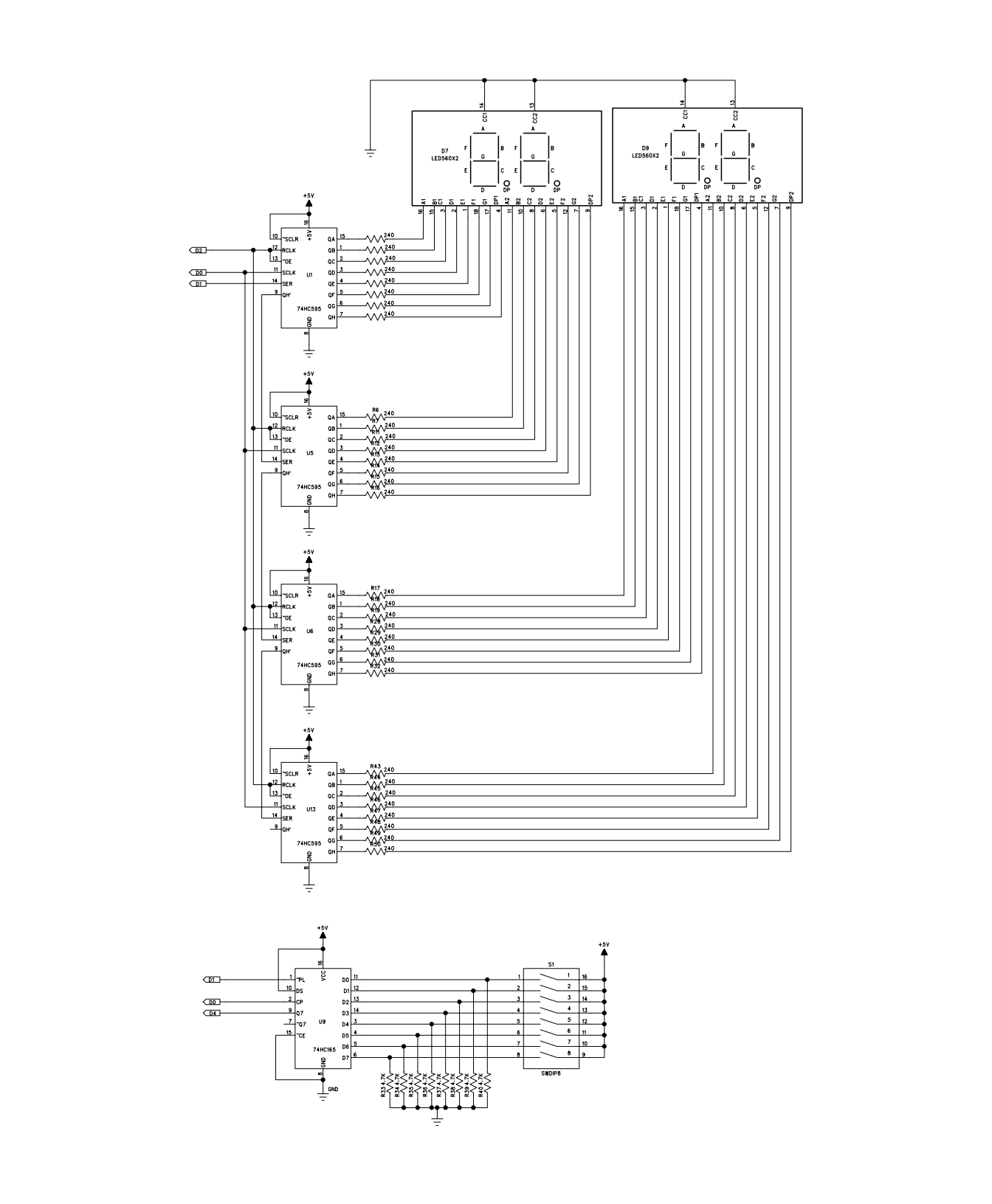
CCS IDE owners can use the CCSLOAD program with:

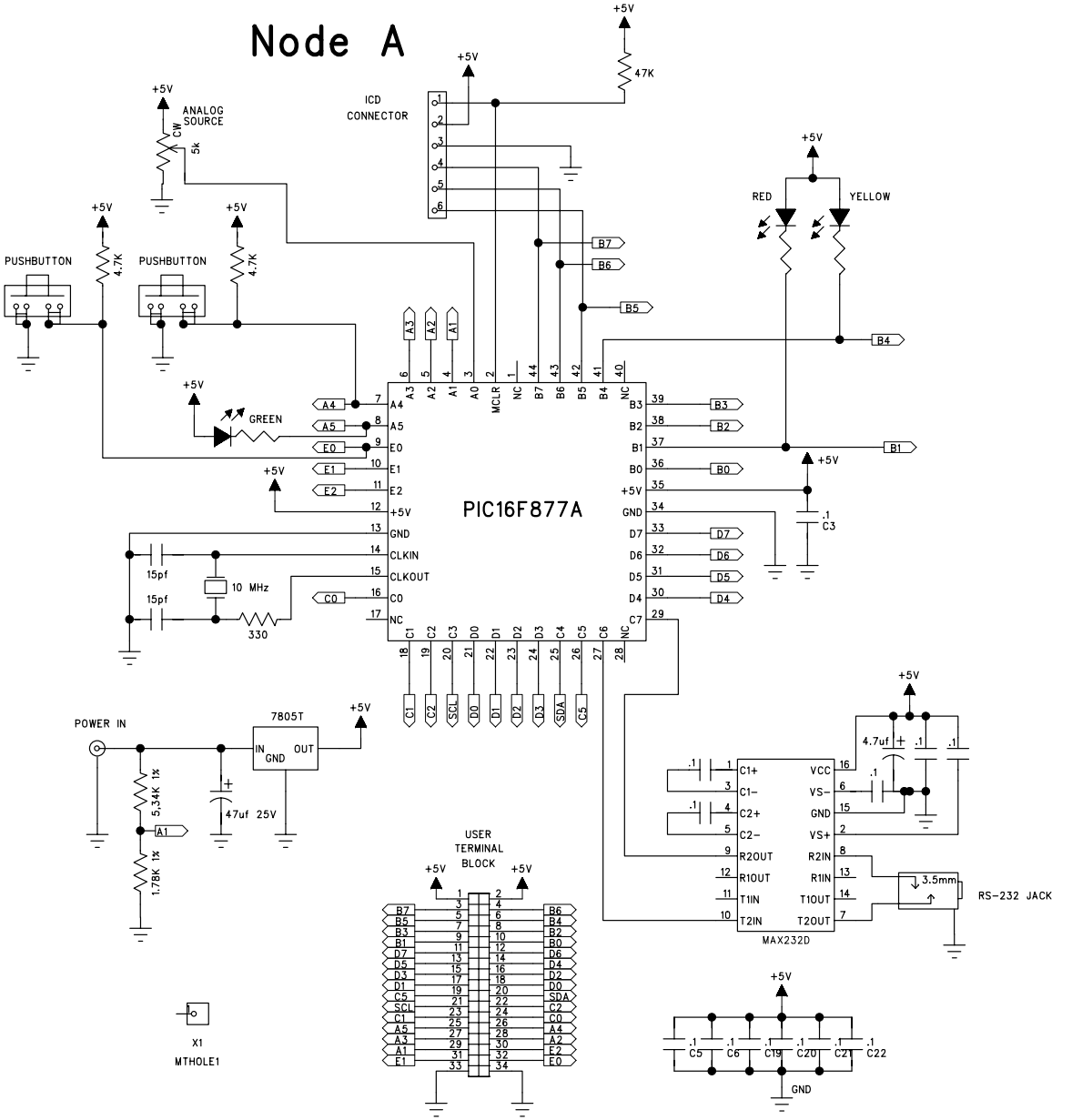
- MPLAB®ICD 2/ICD 3
- MPLAB®REAL ICE™
- **All CCS programmers and debuggers**

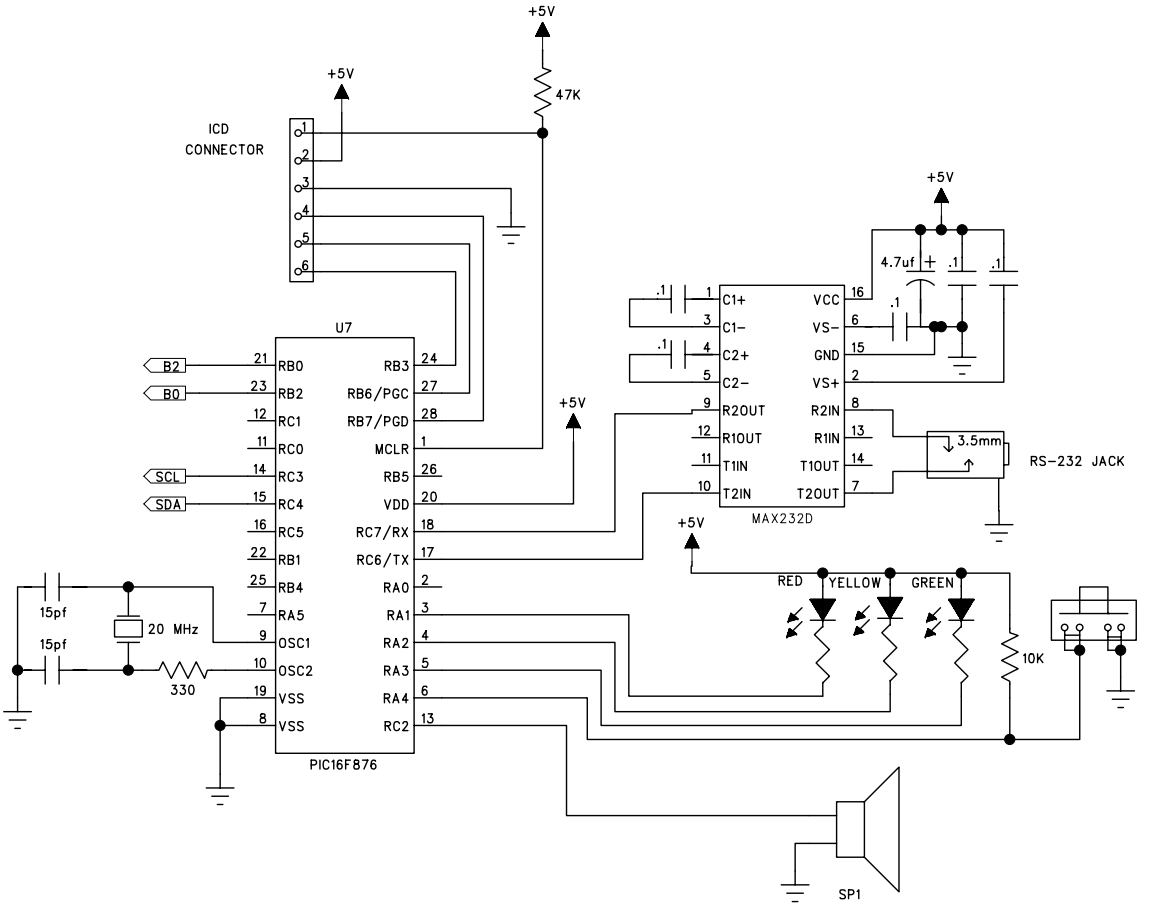
How to Get Started:

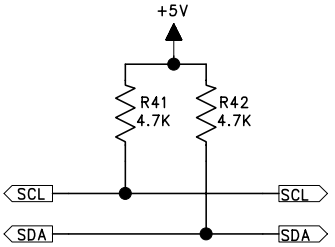
Step 1: *Connect Programmer to PC and target board. Software will auto-detect the programmer and device.*
Step 2: *Select Hex File for target board.*
Step 3*: Select Test Target. Status bar will show current progress of the operation.*
Step 4: *Click "Write to Chip" to program the device.*

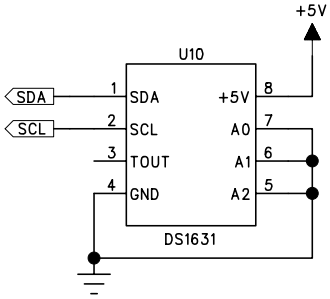
Use the Diagnostics tab for troubleshooting or the ccsload.chm help file for additional assistance.

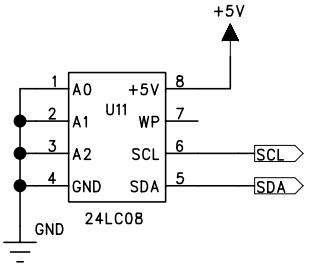

D7
LED560X2
D9
LED560X2
CC1
CC2
DP
+5V
U1
U5
U6
U13
74HC595
SCLR
RCLK
OE
SCLK
SER
QH'
QA
QB
QC
QD
QE
QF
QG
QH
GND
240
D2
D0
D1
U9
74HC165
PL
DS
CP
Q7
Q7'
CE
D3
D4
D5
D6
D7
VCC
S1
SWDIP8
4.7K

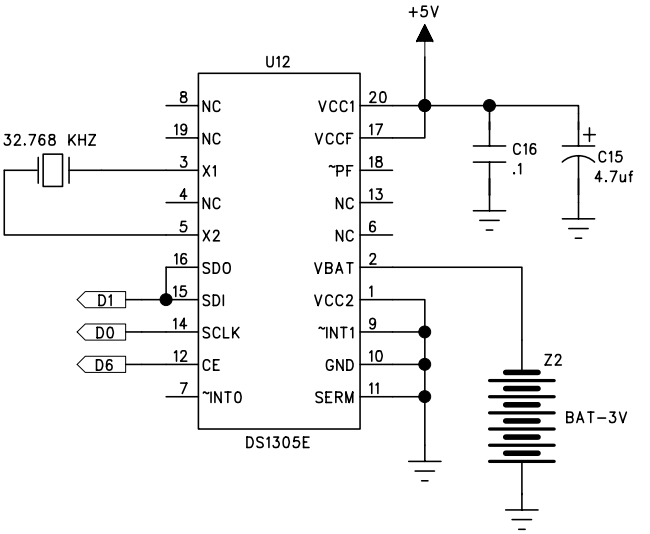

Node A
+5V
47K
ICD
CONNECTOR
ANALOG
SOURCE
CW
5k
PUSHBUTTON
4.7K
RED
YELLOW
GREEN
PIC16F877A
MCLR
NC
A0
A1
A2
A3
A4
A5
E0
E1
E2
B0
B1
B2
B3
B4
B5
B6
B7
C0
C1
C2
C3
D0
D1
D2
D3
D4
D5
D6
D7
SCL
SDA
CLKIN
CLKOUT
15pf
10 MHz
330
.1
C3
POWER IN
7805T
IN
GND
OUT
47uf 25V
5.34K 1%
1.78K 1%
4.7uf
MAX232D
C1+
C1-
C2+
C2-
VCC
VS-
VS+
R2OUT
R1OUT
T1IN
T2IN
R2IN
R1IN
T1OUT
T2OUT
3.5mm
RS-232 JACK
USER
TERMINAL
BLOCK
X1
MTHOLE1
C5
C6
C19
C20
C21
C22
GND

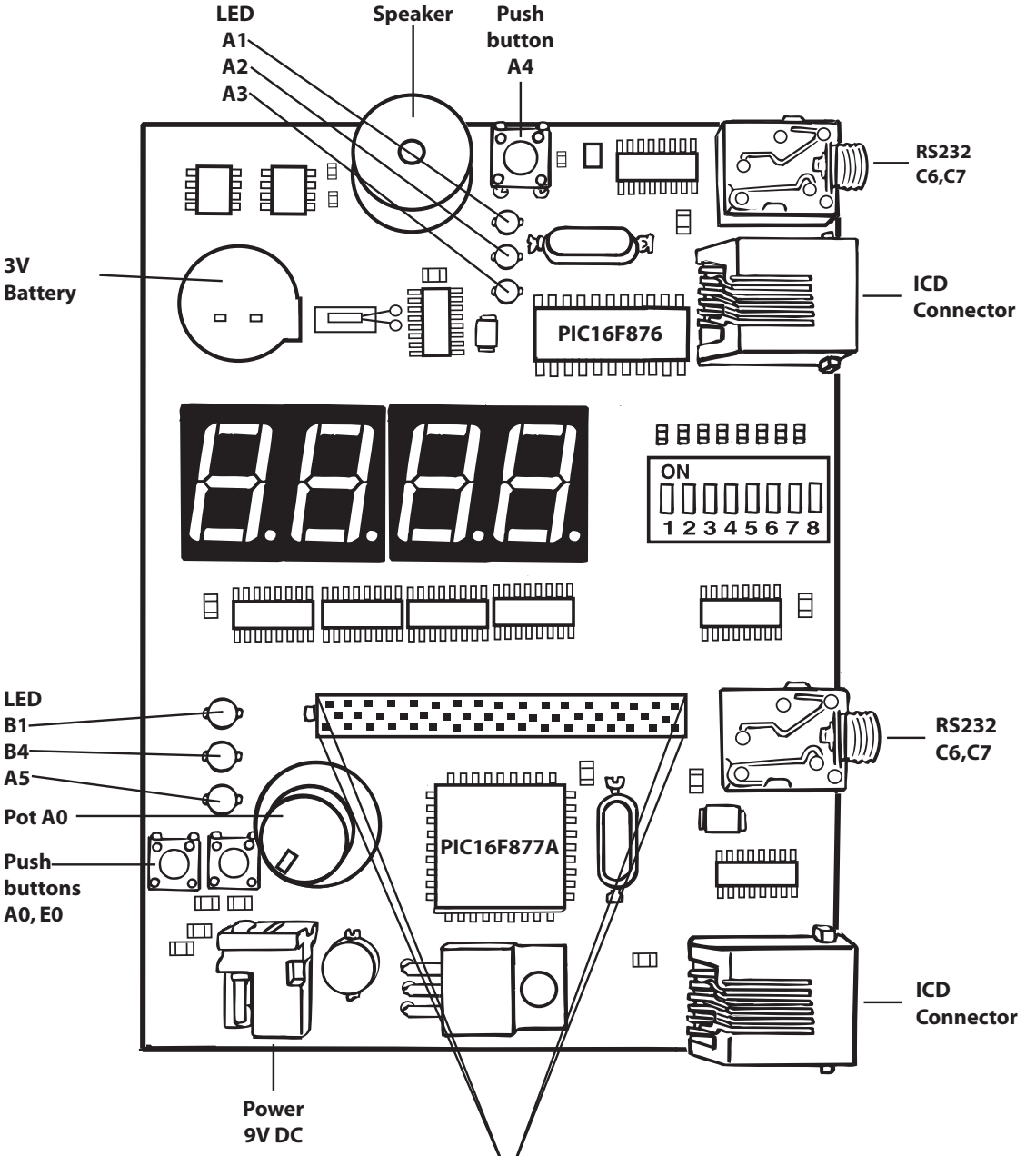

+5V
47K
+5V
ICD
CONNECTOR
1
2
3
4
5
6
U7
B2
B0
SCL
SDA
21 RB0
23 RB2
12 RC1
11 RC0
14 RC3
15 RC4
16 RC5
22 RB1
25 RB4
7 RA5
9 OSC1
10 OSC2
19 VSS
8 VSS
24 RB3
27 RB6/PGC
28 RB7/PGD
1 MCLR
26 RB5
20 VDD
18 RC7/RX
17 RC6/TX
2 RA0
3 RA1
4 RA2
5 RA3
6 RA4
13 RC2
PIC16F876
15pf
15pf
20 MHz
330
+5V
+5V
.1
.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
MAX232D
4.7uf
.1
.1
3.5mm
RS-232 JACK
+5V
RED
YELLOW
GREEN
10K
SP1


+5V
R41
4.7K
R42
4.7K
SCL
SCL
SDA
SDA


+5V
U10
SDA
SCL
1
2
3
4
SDA
SCL
TOUT
GND
+5V
A0
A1
A2
8
7
6
5
DS1631


+5V
U11
1
2
3
4
A0
A1
A2
GND
+5V
WP
SCL
SDA
8
7
6
5
SCL
SDA
GND
24LC08


+5V
U12
32.768 KHZ
8
19
3
4
5
16
15
14
12
7
NC
NC
X1
NC
X2
SDO
SDI
SCLK
CE
~INT0
D1
D0
D6
20
17
18
13
6
2
1
9
10
11
VCC1
VCCF
~PF
NC
NC
VBAT
VCC2
~INT1
GND
SERM
C16
.1
C15
4.7uf
Z2
BAT-3V
DS1305E



+5	B6	B4	B2	B0	D6	D4	D2	D0	C4	C2	C0	A4	A2	E2	E0	G
+5	B7	B5	B3	B1	D7	D5	D3	D1	C5	C3	C1	A5	A3	A1	E1	G