Arduino for the Terrified

Part 3

by
Jack Purdum, Ph.D.
W8TEE

In this part of the series, we want to connect an LCD display to your Arduino so you can display output without requiring the use of your PC. Obviously, we're still using the PC to develop the programs, but at least we can see things displayed on the LCD directly. In Part 2, I suggested that you get a 16x2 LCD display that uses the I2C interface. The reason is because the I2C interface only uses two I/O lines to use the display. If you use the standard LCD parallel interface, it ties up 6 I/O lines, sometimes more.

Setting Up an I2C LCD Display

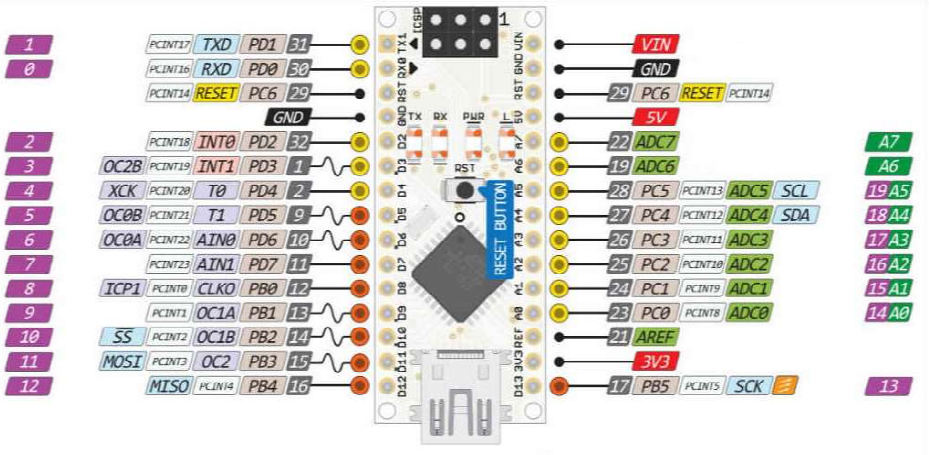
I like the Yourduino display mentioned in Part 2 because it's fairly inexpensive and its vendor has sample Arduino programs, also called *sketches*, that you can experiment with. Figure 1 shows the back of the display and the I2C interface board. Note it only uses four wires: 1) ground (GND), 2) power (VCC, 5V), 3) data (SDA), and 4) clock (SCL). The blue pot control is used to adjust the LCD display's backlight. If you don't see anything displayed or a 16x2 matrix of "blocks", just use a small screwdriver

Figure 1. The back of the I2C LCD display

to adjust it. (It's easier to do when something is printed on the display. Once it's set, you won't have to mess with it again.)

Okay, so where do the other ends of the display attach to the Arduino? Well, the power connections should be pretty obvious. The clock (SCL) and data (SDA) lines aren't obvious. To help you along, Figure 2 shows the pinout for the Arduino Nano. (If you are using an Arduino Uno or similar, Google "Arduino Uno pinout" and you'll find a similar drawing.)

If you look closely at Figure 2, you can see purple numbers displayed on the outer-most edges of the figure. These are the *logical* pin numbers that are used by the bootloader and some internal IDE functions. Lined up with each of those pins numbers are the capabilities that have been assigned to that pin. For example, in the upper-left corner you see a purple 1 and a purple 0 directly below it. If you look to the right of each of those purple numbers, you will see light blue fields where 1 is associated with TXD and 0 is associated with RXD. What this means is the logical pins 0 and 1 are used by the Serial monitor to transmit (TXD) and receive (RXD) data from the Serial monitor. In other words, the USB cable uses these two logical pins to form a



Figu. The pinouts for the Arduino Nano

communications link with the PC. Note this link is used by the IDE to move code from your PC into the Arduino during a compile. Likewise, if you use the Serial object as an input device like we did in the game in Part 2, those

two I/O lines need to be kept free.

While we're in the neighborhood, note that logical pins 2 and 3 have been assigned INT0 and INT1 (i.e., the pink fields). These are external interrupt pins. While we're not doing anything with those pins yet, I point it out now because I always try to avoid dedicating "normal" I/O work to those pins just in case I need to use an interrupt somewhere down the line in the project.

A lot of sample code you'll find online uses the two interrupt pins for connecting a parallel interfaced LCD display to the Arduino. If you see this, just move those two lines from the display to some other pins and reflect that in your code. For example, the Arduino user site has this code for creating an LCD display:

```
LiquidCrystal lcd(12,11,5,4,3,2);
```

which means the interrupt pins are being used for the display. If pins 6 and 7 aren't being used in the program, you can change the statement to:

```
LiquidCrystal lcd(12,11,7,6,5,4);
```

which makes it easier to add an interrupt routine if it's needed later.

Going back to Figure 2, look for the purple (logical) pins 18 and 19 on the right side of the figure. To the right of those numbers are A4 and A5 as displayed in green fields. These 'A' designations mean that those pins can be used for reading an analog device. The ADC (Analog to Digital Converter) capability within the Arduino is a 10-bit converter, so the voltage readings are mapped to the values 0-1023.

Now look to the left for those same two pins and you will see two light blue fields with SCL and SDA. Wait a minute...didn't you just see that in Figure 1? Yep, so those are the two pins where you connect the SCL and SDA lines from the I2C LCD display to the Arduino. If you look three pins "higher" from A4 and A5, you will see a 5V pin which provides a power source for the display. Two pins above that is a GND connection.

Keep in mind that all of the Arduino pins are low current devices with about a 40mA max on each pin. As a rule, I try to run at about half that power level. If you are using an LCD display using a parallel interface, you should put a dropping resistor in each of those lines. Almost any value between 200-1000 ohms will work. Experience has taught me that all electronic devices run on white smoke and once that white smoke gets out, it's *really* hard to put it back in. Moral: use dropping resistors even if the display seems to work without them.

BTW, you can buy prototyping boards for the Nano that makes connecting things much easier. Figure 3 shows an example. I've marked the 4 pins that provide all of the I2C interface pins. As you might

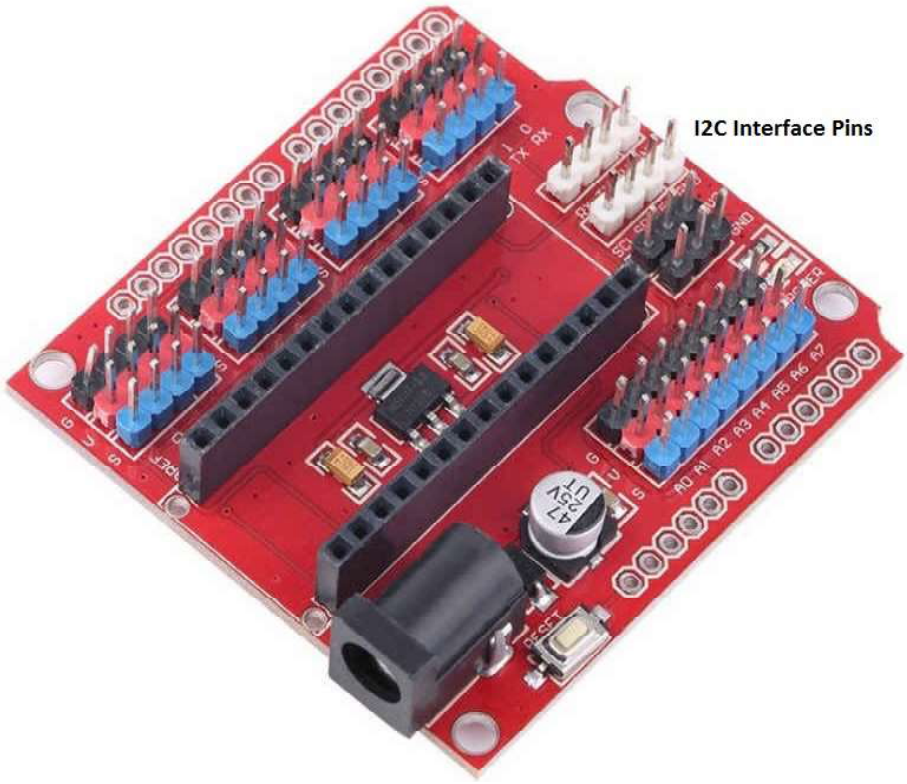


Figure 3. Nano prototype board.

guess, the Nano fits into the large header sockets in the middle and all of the board's pins tie into the Nano I/O and other pins. You can also power the board with a wall wart using voltages between 6V and 17V, although the regulator can get a little toasty at the higher voltages. A 9V wall wart is a good choice.

There is another nice feature of these prototype boards. As you may know, there's approximately a bazillion add-on boards, called *shields*, that can be directly "plugged into" an Arduino Uno. (Go to eBay and type in "Arduino Wifi shield" to see an example.) However, because the Nano uses a smaller footprint, you can't plug an Uno shield into a Nano.

Or can you?

If you look at the outer edge of the prototype board in Figure 3, you will see empty holes for more pins. If you solder socket headers into these holes, an Uno shield can plug directly into the sockets. This would allow you to use an Uno shield with a Nano! For about $2 each, I bought several of these online. I also bought an assortment of Dupont jumper wires (M-M, M-F, and F-F) online to make connecting to the pins easy. See:

http://www.ebay.com/itm/120pcs-Dupont-Wire-Male-to-Male-Male-to-Female-Female-to-Female-Jumper-Cable-/121868077477

An LCD Demo Program

Listing 1presents a short demo program using the I2C LCD display. I assume you've connected the 4 wires to the display from the Nano. Figure 4 shows my setup:

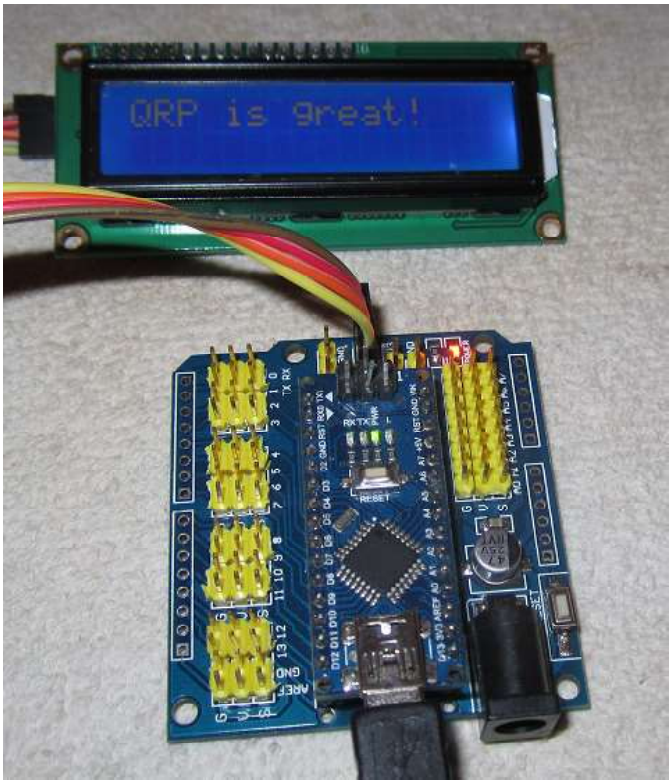


Figure 4. LCD and Nano protoype board

The first line in Listing 1 is a preprocessor *#include* directive, which is used to access pre-written code and make it available to your program. In this case, the program wants to access the code in the *Wire* library. When you installed the Arduino software package, it automatically copied the *Wire* library into the *libraries* subdirectory during the installation process. The angle brackets ('<' and '>') tell the compiler to look in the default library directory for this file, and that's exactly where you will find it. (If you used double quote marks instead of the angle brackets, it would first look in the same directory where the program you are writing is located on your disk.) You will also see in Listing 1 that a second *#include* is used to access the *LiquidCrystal_I2C*library. Alas, this library is not automatically included with the Arduino installation.

What is a Library?

A C programming library is simply a collection of functions that are designed to perform a specific task. The *Wire* library is used to implement the I2C progamming interface to the Arduino controllers. That library includes a bunch of functions designed to make your life as a programmer easier. Likewise, the *LiquidCrystal_I2C* library is used to interface to the LCD display using the I2C interface. The third statement line in Listing 1, for example, creates the *lcd* object, and the second statement line in *setup()* has a function, *lcd.begin(16, 2),* that initializes (Step 1, remember?) the *lcd* object to work with a 16x2 display.

However, since your installation doesn't include the *LiquidCrystal_I2C* library, how can you access it? Easy, go to the URL given right above the *#include* directive, download the *LiquidCrystal_I2C* library into the *libraries* subdirectory, and unzip it.

Installing a New Library

Unfortunately, installing a new library isn't always the same for all libraries. If you go to the URL given in Listing 1, the zip file is named *NewliquidCrystal_1.3.4.zip.* When you unzip that file, it creates two subdirectories: MACOSX for the Mac and NewLiquidCrystal for Windows. I'm using windows, so that's the subdirectory I want. If you look in that directory, you will find 37 directories and files in it. If you are using Windows Explore to look around on your disk system, use Organize --> Select All to highlight all of the directories and files. Now do Organize --> Copy to copy all of the files to the Windows copy buffer.

Now move to the libraries subdirectory of your installation. For example, if you installed the IDE as I suggested in Part 1, you would go to:

C:\Arduino1.6.9\libraries

Once there, create a new directory named *LiquidCrystal_I2C*. When you are done, you should see all of the original library subdirectories plus the new one you just created. Now, move into the new *LiquidCrystal_I2C* directory and select Organize --> Paste. This will copy those 37 items into the new directory. Now you are all set you use the I2C library for your LCD. If you

have the IDE open, you must close and reopen it for the IDE to know about the new library.

This process is pretty common for installing most new libraries, but the root library name may or may not be close to the name of the ZIP file. However, as a general rule, the library subdirectory name should match the header file name in the program. For example, our *#include* directory uses LiquidCrystal_I2C.h for the file name to be included. As a result, the directory name is the same, but drops the ".h" from the file name. If you look inside that directory, you will find the LiquidCrystal_I2C.h header file.

I don't want to explain this any further right now. As you gain more programming experience, all of this will make more sense. For now, just trust me that it will work!

Listing 1. LCD Demo Program

```
#include <Wire.h>                          // Arduino IDE has
this library

// Must add:   https://bitbucket.org/fmalpartida/new-
liquidcrystal/downloads
#include <LiquidCrystal_I2C.h>


// Some displays may use 0x3F instead of 0x27 for the I2C
device address
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3,
POSITIVE);

char message[17];              // Max message is 16 chars
char spaces[17] = "      ";    // 16 spaces to erase a line

void setup()
{
  int i;

  Serial.begin(9600);
  lcd.begin(16, 2);    // initialize lcd for 16 chars 2
lines, backlight on
```

```
  // ——— Quick 3 blinks of backlight  ————————-
  for (i = 0; i < 3; i++)
  {
    lcd.backlight();
    delay(250);
    lcd.noBacklight();
    delay(250);
  }
  lcd.backlight(); // finish with backlight on

  lcd.setCursor(0, 0); //Start at character 0 on line 0
  lcd.print("QRP rules!");
}


void loop()
{
  int charsRead;

  if (Serial.available() > 0) {
    lcd.setCursor(0, 1);      // second line
    lcd.print(spaces);
    charsRead = Serial.readBytesUntil('\0', message,
sizeof(message) - 1);
    message[charsRead] = NULL;
    lcd.setCursor(0, 1);      // second line
    lcd.print(message);
  }
}
```

The Program

In the *setup()* function, we create the Serial object as we have before, and we initialize the LCD display to support a 16 column by 2 line display. We then enter a *for* loop to blink the display 3 times. A *for* loop has three parts to it, each separated by a semicolon:

for (expression1 ; expression2 ; expression3)

expression1 sets the starting value for the loop. In this example, we set *i* to 0. *expression2* is usually a logical test of some kind. In this example, we are testing the value of *i* to see if it is less that 3. If it is True that *i* is less than 3,

we execute the statements that are between the opening ({) and closing (}) *for* statement braces. In our program, the statements in *for* statement block turn the backlight on, wait a quarter of a second, turn the backlight off, and wait another quarter of a second. The program execution then immediately goes to *expression3*.

expression3, *i*++, is a C idiom that takes the current value of a variable and increments it by 1. Therefore, you could also write *expression3* as:

```
i = i + 1;
```

However, since programmers are usually lazy, they prefer the shorter version. Once *i* has been incremented, program control immediately goes back to *expression2* and asks if *i* is still less than 3. Since it is, another pass through the *for* statement block is made. Control then goes to *expression3*...you get the idea. Eventually, *i* will equal 3, at which time the test is logic False and program control will go immediately to the next statement after the closing brace (}) of the *for* statement block. In our program, the next statement turns the backlight back on.

The next line after that sets the LCD cursor to the first character position on the first line and the next statement displays a message on the display. Note: In programming, almost all counting begins with 0, not 1. So the x-y coordinates for the first character of the first line is 0,0. You'll get used to is.

That's it for *setup()*.

loop()

In *loop()*, we are using the Serial object's *available()* function again like we did in Part 2, to see if the user has type anything into the Serial monitor. However, this time we want the user to type in a short message to display on the LCD rather than just a single letter. The first two statements simply place the cursor on the second line and overwrite (i.e., erase?) anything that might be on the second line of the display with spaces. (Near the top of the program we defined a character array with 17 characters in it, but using only 16 spaces. We'll explain the different numbers in a bit.)

The statement:

```
    charsRead = Serial.readBytesUntil('\n', message,
sizeof(message) - 1);
```

uses a different Serial function to read what the user typed into the Serial monitor. Notice that there are 3 arguments that are passed to the *readBytesUntil()* function. The first argument, '\n', tells the function: "I want you to read characters into the Serial buffer until you see the newline character ('\n')." The newline character is sent by the Serial monitor when the user presses the Enter key or clicks the Send button.

The second argument says: "Once you see the newline character, I want you to copy the contents of the Serial buffer into the *char* array named *message[]*." The final parameter says: "Oh, by the way, if the user is an idiot and tries to write more characters than will fit in *message[]*, move a newline character into the Serial buffer and then copy the contents into *message[]* and go home." The third argument, therefore, provides a means by which we can avoid overflowing the *message[]* array. Overflowing any array is almost always a train wreck. But what's this *sizeof* stuff all about?

sizeof Operator

The *sizeof* operator is used to find out how many bytes of memory have been allocated to a variable. So, if you wrote *sizeof(message)*, it would return 17. The reason is because each *char* data type takes 1 byte of memory. Since we defined the *message[]* array to hold 17 *char*s, it returns 17. Suppose I define *vals[]* as follows:

```
int vals[17];
Serial.print(sizeof(vals));
```

What would be displayed on the Serial monitor? It would display 34. The reason is because each *int* data type takes 2 bytes of storage, so it would use 34 bytes of memory.

Going back to our statement:

```
    charsRead = Serial.readBytesUntil('\n', message,
sizeof(message) - 1);
```

the third argument uses the *sizeof()* operator to get 17, but it then subtracts 1 to give the value of the third argument as 16. Why? Well, the display only holds 16 characters, and we told the code to write the newline character as the 17th character if they try to overflow the buffer. The problem is that computers start counting at 0, not 1. This means the 17 array elements go from 0 through 16, not 17. So, the 17th character actually lives at element 16 in the array (i.e., *message[16]*). The function returns the number of characters that were read into the *message[]* array. At a maximum, the return count for *charsRead* would be 16. If they typed in a shorter message, it would be the number of characters they did type.

String Variables in C

We now want to take the contents of the *message[]* array and display it on the LCD. To do that, we need to make what the user typed in look like a *string* variable. A string variable is nothing more than a *char* array arranged so it can be used in a textual context. For example, take the message we displayed in *setup()*. Somewhere in memory, the compiler made enough room for that message and it would look like:

QRP rules!0

Note that at the end of the message there is a zero. If you look up the ASCII code for 0, you will see it is defined as *nul*. *nul* is represented as a character as '\0'. The backslash is used so the compiler knows it is *nul*, not the digit character '0' (i.e., zero). In C, any function that works with string data knows that *nul* marks the end of the string. So if the function needs to read the contents of a string variable, it just starts at the beginning and reads until it find the *nul* character. (This is a lot more efficient than some other languages that use string descriptor blocks at the front of the string data to describe its length.)

With this in mind, look at the next line in the program:

```
message[charsRead] = NULL;
```

The Arduino IDE defines *NULL* as '\0'. So what does the line do? Suppose you wrote "Hello" on the Serial monitor and pressed Enter. The variable

charsRead would equal 5, since that's how many characters you typed. So the statement becomes:

```
message[5] = NULL;
```

which means that you have written the *nul* string termination character into the 5th element of the *message[]* array. In memory, this would look like:

H e l l o 0

0 1 2 3 4 5 --> *Element number in the message[] array*

This now means that we can use *message[]* as a string variable in the program. True, it's still a *char* array, but because of the *nul* character at the end, any function that expects string data can now use *message[]* as a string variable.

As you no doubt figured out by now, the program then displays what you typed onto the LCD array. The code then sends control back up to the first statement to see if you typed something new into the Serial monitor. If not, the program spins around in the loop waiting 'til the cows come home...

Your assignment: Have the user type a number in the Serial monitor and you display that number along with its square. That is, if they type in 5, you display that and 25. Keep in mind that textual data from the Serial monitor is text, not numbers, so you need to convert from text to numbers. To help you along, take a look at the *atoi()* function. To find out about it, type in "atoi for C" into your search engine.

If that's too easy for you, use a different algorithm. I discovered by accident the following algorithm to square a number:

The sum of n odd integers, starting with 1, equals the square of n.

Example: what is the square of 3? The three odd integers are:

square = 1 + 3 + 5 = 9

For 5:

square = 1 + 3 + 5 + 7 + 9 = 25

If you can implement this algorithm in code on the first try, you probably aren't going to learn much here!

Create some of your own programs and have fun!

Arduino for the Terrified,Part 4
by Jack Purdum, Ph.D., W8TEE

So, you have been experimenting with your Arduino and doing some conversions of textual data to numbers, right? In this part, we're going to build a *really* simple code practice oscillator, but we're going to use the keyboard instead of a key or keyer. If you've used the digital modes, you've already used this kind of setup. In this case, however, we're going to capture a message you type into the Serial monitor and display it on the LCD display. We're also going to generate an audible tone in Morse code that matches the character being sent. So the real purpose here is to add a simple external device, a piezo buzzer, and show a little more programming with the LCD display. Also, this isn't a bad way to learn Morse code...listening and seeing at the same time.

[sidebar]
I learned Morse code the wrong way. Back when I got my license in 1954, you had to know Morse to get your Novice license. So, I sat down and memorized the code one letter at a time, starting with 'A'. I didn't realize it at the time, but to figure out the difference between a 'B' and a 'D', I was counting dits in my head and then writing the letter down on paper.

This approach pretty much dooms you to around 15wpm, max.

If you're in the process of learning Morse, or you want to improve your speed, the method used by Learn CW Online (lcwo.net) is the way to go. Their approach is for you to pick a target speed that you would ultimately like to be able to receive. In my case, I want to be able to receive at 30wpm, so that's my target speed. What's interesting is that they *start* you out at that rate, but they place a large time delay between characters. (You have control over that delay.) What this approach forces you to do that I didn't do was listen to the *rhythm* of the characters rather than counting dits and dahs. As you become more comfortable at that pace, you reduce the time gap between letters. Eventually, with some practice, you'll hit your target speed. This kind of encoding is modeled after Farnsworth encoding and is more common than you think. W1AW, for example, uses Farnsworth encoding when practice speeds exceed 18wpm.

Anyway, the application that lcwo uses is free, so you might give it a try if you' re learning or just want to increase your speed.
[end sidebar]

To perform the experiment in this part, you' ll have to cough up enough cash to buy a 5V piezo buzzer. I had one laying around here that was mounted on a small board, but you can find them pretty cheap:

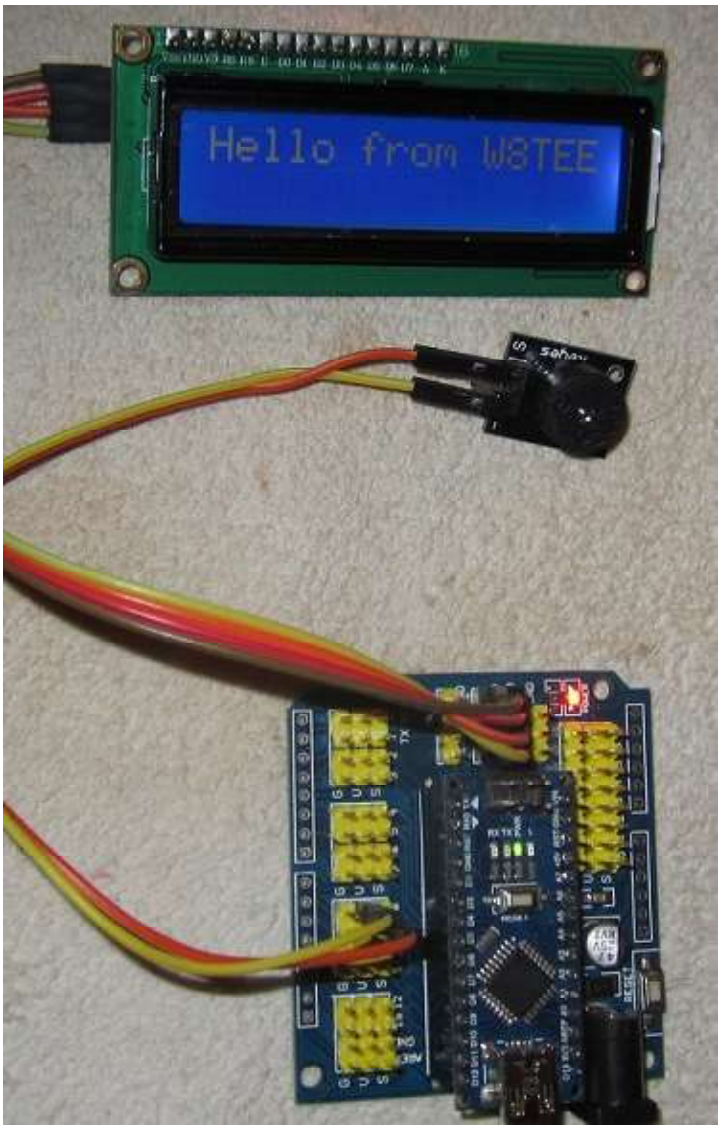


Figure 1. LCD display and piezo buzzer.

http://www.ebay.com/itm/10Pcs-5V-Active-Piezo-Electronic-DC-Buzzer-Alarm-Speaker-for-Arduino-85db-/172189503879
hash=item28174b5187:g:TN8AAOSw2GlXJk0N

This is a domestic company that sells 10 for at $0.65 a piece, including shipping. Just make sure it's a 5V buzzer and that it doesn't draw over 30mA or so. If it draws more than that, you may have to put a resistor in line with the buzzer. Figure 1 shows my setup. The LCD display hookup is unchanged from

the last program in Part 3. You can tie the buzzer to almost any pin you wish. I used pin 9 because...well, I don't know why...I just did. I connected the other buzzer lead to +5V. I did this to show you that turning it on actually requires pulling the I/O pin LOW, which may be different than the way you'd normally think about it. You can do it either way. Listing 1 shows the complete source code.

List 1. Code practice oscillator

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

#define SCREENWIDTH     16    // Chars on one line of LCD
#define WPM             20
#define FREQ           700
#define MAXMSGLENGTH   128

char msg[MAXMSGLENGTH + 1];    // Don't forget the NULL
char spaces[SCREENWIDTH + 1];

int ditlen = 1200 / WPM;

const int LEDPIN  = 13;       // blink the LED for now...
const int TONEPIN = 9;         // tone pin

char letterTable[] = {              // Morse coding
  0b101,           // A
  0b11000,          // B
  0b11010,          // C
  0b1100,          // D
  0b10,           // E
  0b10010,          // F
  0b1110,          // G
  0b10000,          // H
  0b100,           // I
  0b10111,          // J
  0b1101,          // K
```

```
 0b10100,         // L
 0b111,           // M
 0b110,           // N
 0b1111,          // O
 0b10110,         // P
 0b11101,         // Q
 0b1010,          // R
 0b1000,          // S
 0b11,            // T
 0b1001,          // U
 0b10001,         // V
 0b1011,          // W
 0b11001,         // X
 0b11011,         // Y
 0b11100          // Z
};

char ntab[] = {
 0b111111,        // 0
 0b101111,        // 1
 0b100111,        // 2
 0b100011,        // 3
 0b100001,        // 4
 0b100000,        // 5
 0b110000,        // 6
 0b111000,        // 7
 0b111100,        // 8
 0b111110         // 9
};

LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);

/*****
 Purpose: to cause a delay in program execution

 Paramter list:
  unsigned long millisWait    // the number of millseconds to wait

 Return value:
  void
*****/
void mydelay(unsigned long millisWait)
{
  unsigned long now = millis();
```

```
 while (millis() - now < millisWait)
   ;     // Twiddle thumbs...
}

/*****
 Purpose: to send a Morse code dit

 Paramter list:
  void

 Return value:
  void

 CAUTION: Assumes that a global named ditlen holds the value for dit spacing
*****/
void dit()
{
 digitalWrite(LEDPIN, HIGH);
 digitalWrite(TONEPIN, LOW);
 mydelay(ditlen);
 digitalWrite(LEDPIN, LOW);
 digitalWrite(TONEPIN, HIGH);
 mydelay(ditlen);

}

/*****
 Purpose: to send a Morse code dah

 Paramter list:
  void

 Return value:
  void

 CAUTION: Assumes that a global named ditlen holds the value for dit spacing
*****/
void dah()
{
 digitalWrite(LEDPIN, HIGH);
 digitalWrite(TONEPIN, LOW);
 mydelay(3 * ditlen);
 digitalWrite(LEDPIN, LOW);
```

```
 digitalWrite(TONEPIN, HIGH);
 mydelay(ditlen);
}

/*****
 Purpose: to provide spacing between letters

 Paramter list:
  void

 Return value:
  void

 CAUTION: Assumes that a global named ditlen holds the value for dit spacing
*****/
void lspace()
{
 mydelay(3 * ditlen);
}

/*****
 Purpose: to provide spacing between words

 Paramter list:
  void

 Return value:
  void

 CAUTION: Assumes that a global named ditlen holds the value for dit spacing
*****/
void space()
{
 mydelay(7 * ditlen);
}

/*****
 Purpose: to send a Morse code character

 Paramter list:
  char code       the code for the letter to send

 Return value:
  void
```

```
*****/
void SendCode(char code)
{
 int i;

 for (i = 7; i >= 0; i—)
  if (code & (1 << i))
    break;

 for (i—; i >= 0; i—) {
  if (code & (1 << i))
    dah();
  else
    dit();
 }
 lspace();
}

/*****
 Purpose: to send a Morse code character


 Paramter list:
  char myChar       The character to be sent

 Return value:
  void
*****/void send(char myChar)
{
 if (isalpha(myChar)) {
  if (islower(myChar)) {
   myChar = toupper(myChar);
   }
   SendCode(letterTable[myChar - ‘A’]);   // Make into a zero-based array index
   return;
 } else if (isdigit(myChar)) {
  SendCode(ntab[myChar - ‘0’]);   // Same deal here...
   return;
 }
 switch (myChar) {               // Non-alpha and non-digit characters
   case ‘ ‘:
   case ‘\r’:
   case ‘\n’:
     space();
```

```
     break;
    case '.':
     SendCode(0b1010101);
      break;
    case ',':
     SendCode(0b1110011);
      break;
    case '!':
     SendCode(0b1101011);
      break;
    case '?':
     SendCode(0b1001100);
      break;
    case '/':
     SendCode(0b110010);
      break;
    case '+':
     SendCode(0b101010);
      break;
    case '-':
     SendCode(0b1100001);
      break;
    case '=':
     SendCode(0b110001);
      break;
    case '@':
     SendCode(0b1011010);
      break;
    default:
      lcd.setCursor(0, 1);
      lcd.print("unknown char");
      break;
  }
}
//=====================================================================
void  setup()
{
 pinMode(LEDPIN, OUTPUT);
 pinMode(TONEPIN, OUTPUT);
 digitalWrite(TONEPIN, HIGH);

  memset(spaces, ' ', sizeof(spaces));    // Fill array with space character

 Serial.begin(9600);
```

```
 lcd.begin(SCREENWIDTH, 2);
}

void loop()
{
  int charsRead;
  int index;

  if (Serial.available() > 0) {
    lcd.setCursor(0, 0);
     lcd.print(spaces);
    lcd.setCursor(0, 1);
     lcd.print(spaces);

    charsRead = Serial.readBytesUntil('\n', msg, sizeof(msg) - 1);
    msg[charsRead] = NULL;
    Serial.println(msg);
    index = 0;
     ShowChar(msg, charsRead);
    while (msg[index]) {
      ShowChar(msg, 0);
      send(msg[index]);
     digitalWrite(TONEPIN, HIGH);
      index++;
    }
  }
}

void ShowChar(char s[], int whichOne)
{
  static int index;
  char temp[SCREENWIDTH + 1];

  if (whichOne != 0) {
    index = 0;
  }
  if (index < SCREENWIDTH) {
    if (s[index] < 32)       // Probably null
      return;
    lcd.setCursor(index, 0);
    lcd.print(s[index++]);
  } else {
    memcpy(temp, &s[index - SCREENWIDTH], SCREENWIDTH);
    temp[SCREENWIDTH] = '\0';
```

```
   lcd.setCursor(0, 0);
   lcd.print(temp);
   index++;
  }
 }
}
```

#define Preprocessor Directive

The program begins with a series of preprocessor directives for symbolic constants used in the program. For example:

```
#define SCREENWIDTH     16    // Chars on one line of LCD
```

A *#define* is a preprocessor directive that causes a *textual* replacement in the source code. So, anywhere you see SCREENWIDTH in the program, the proprocessor replaced that word with the digit characters 16. Most people find it easier to read and understand the symbolic constant SCREENWIDTH than a magic number like 16.

Still, since we know we're using a 16x2 display, why not just use 16 instead of SCREENWIDTH? Go back and count how many times you see SCREENWIDTH used in the program. There are six places where it is used. So, now you upgrade to a 20x4 display. Which is easier: Change SCREENWIDTH from 16 to 20 in the preprocessor directive once at the top of the file, or search-and-replace six times in the file? Which is more error-prone?

Also, it's been shown that when someone is doing a search-and-replace of 16 with 20, the programmer is sorely tempted to do a global Replace All action. That can lead to disastrous consequences. For example, suppose software to mill a canon with a 1600mm bore contains the statement:

```
canonBore = 1600;
```

and you do a global search-and-replace with 20. The canon's bore just got changed to 2000mm. Usually, test firing a canon designed for a 1600mm shell using a barrel with a 2000mm bore is almost never going to be a happy day at the firing range. The lesson: You should never do a global search-and-replace in program code. It seems global replace will always come back to bite you in the butt later on.

Array Definitions

Also near the top of the program you'll see the definition and initialization of an array that begins with:

```
char letterTable[] = {              // Morse coding
  0b101,            // A
  0b11000,          // B
```

An array is nothing more that a group of identical data types collected under a single variable name. The variable *letterTable[]* is a *char* array that holds binary values that are used to represent letters of the alphabet. While you could use *letterTable[26]* in the definition (note the element count is supplied here but not in the program code), you can omit the size parameter when you have an initialization list because the compiler is *really* good at counting things, so it knows how many elements it needs to make room for in memory.

As you probably know, the letter 'A' is dit-dah (.-) in Morse code. We are using a coding algorithm that says: starting with the most significant bit, search for the first binary digit that has the value 1. You also probably know that each memory byte consists of eight binary digits, called *bits*. (Did you know that 4 bits is called a *nibble*? You'd be surprised how rarely that piece of information comes up at a cocktail party.)

Actually, the binary value we've assigned to the first element of the array is 0b00000101. Reading from left to right (i.e., most significant bit (msb) to least significant bit (lsb)), the 0b tells the compiler that we are representing this number in binary (i.e., base 2) rather than the default representation (i.e., decimal, or base 10). Because each *char* data type uses 8 bits for storage, we are assigning the next 5 bits with the value 0. However, since those bits are zero, the compiler is just as happy with 0b101 as with 0b00000101. Now, suppose we say: When you find the first 1 digit, throw it away. We are just using that first 1 bit as a marker, or *sentinel*, to mark the start of the real data. Once you've tossed that leading 1 bit away, interpret each 0 bit as a dit and each 1 bit as a dah.

So, let's look at the first couple of elements in the array:

0b101	becomes 01	becomes dit-dah	A
0b11000	becomes 1000	becomes dah-dit-dit-dit	B
ob11010	becomes 1010	becomes dah-dit-dah-dit	C

and so on. If you look at some alternatives for coding Morse code, often you will often see the letter A stored as .-, and the digit 0 as - - - - -. Storing each Morse symbol as a small string means every period and dash uses one *char* (i.e., 1 byte). With the numbers, each requires 6 bytes of storage (don't forget the *NULL* character at the end of each string!) Punctuation characters are even worse. However, our algorithm only uses 1 byte for each Morse symbol regardless of whether it's alpha, numeric, or punctuation. Think about it...

Now that you've thought about it, the process I just described is done by the *SendCode()* function. I've repeated it here with a few more comments:

```
void SendCode(char code)
{
 int i;

 for (i = 7; i >= 0; i—)        // Start with MSB, 7, and work down to a 1 bit
```

```
  if (code & (1 << i))          // Did you find the 1 bit?
    break;                                  // Yep...get outta here...

 for (i—; i >= 0; i—) {         // Now start looking for...
  if (code & (1 << i))          // ...1 bits or...
    dah();
   else
    dit();                      // ...0 bits
 }
 lspace();                                          // Add in a letter space
}
```

The first *for* loop starts with the most significant bit, bit 7, and performs a logical AND on the binary coded number after setting the ith bit to 1 in the logic mask. The '<<' operator is the left shift operator and it has the effect of setting the ith bit to logic 1. The logic AND operator, *&*, performs a logical AND with the binary number of *code* to figure out whether it's read a 1 bit or not. If we are looking at the letter 'A', 00000101, the first *for* loop spins through bits 7-2 before it finds the 1 bit in position 2. It then breaks out of the first *for* loop with i = 2.

The second *for* loop first decrements i (*expression1* in the *for* loop, or i—) then sees if the next character is a 0 after the bit masking AND operation on *code*, so it sends a dit by a call to the cleverly-named function *dit()*. On the next iteration of the loop, it will see the 1 and send a dah via the call to *dah()*.

I really don't want to take the time to discuss bit shifting and logic masking. A good introductory programming book on C with such details is *Beginning C for Arduino*, 2nd edition. (In the spirit of full disclosure, I am biased about this book.) You can also get details by Googling bit shifting in C and truth tables in C.

Much of the rest of the code uses things we discussed in earlier programs. If you run the program and type in a message that is longer than the screen width, the code begins to horizontally scroll the message. This section of code would benefit from some explanation because horizontal scrolling is a common need when using LCD displays.

Horizontal Scrolling

The *ShowChar()* function is responsible for horizontal scrolling. The following code fragment controls the scrolling:

```
 if (index < SCREENWIDTH) {         // Have we shown less than SCREENWIDTH
chars?
   if (s[index] < 32)       // Yes, so is it a non-printing character?
     return;                            // Yes...go home
   lcd.setCursor(index, 0);  // Nope...so display it.
```

```
  lcd.print(s[index++]);
 } else {                                  // Yep...more than SCREENWIDTH chars
  memcpy(temp, &s[index - SCREENWIDTH], SCREENWIDTH);
  temp[SCREENWIDTH] = '\0';
  lcd.setCursor(0, 0);
  lcd.print(temp);
  index++;
 }
```

The *if* test checks to see if we even need to worry about scrolling yet. If not, we check to make sure it's a character than can be displayed. (ASCII codes below 32 are non-printing.) If it can't be displayed, program control returns to the place where *ShowChar()* was called from. If it is a printable character, we display it and return to the caller.

The index++ expression is called a *post-increment* operation. For example, suppose index is 5 when we reach this statement:

```
  lcd.print(s[index++]);
```

what this means is that we want to send element position 5 to the LCD display:

```
  lcd.print(s[5]);
```

Remember that this is actually the 6th character in *s[]* because arrays start counting with 0. So, the code fetches that character and sends it to the display. Then the code increments *index* to 6. Therefore, a post-increment uses the current value of a variable and, *after* it is used in the expression, it increments it. Guess what *++index* does? How 'bout *index—* and *index*? All of these operators are just a shorthand way of doing something. For example:

```
index++;
```

is the same as

```
index = index + 1;
```

You can probably figure the other ones out, too.

The memcpy() function

Now comes the tricky part; the use of the *memcpy()* function. The *mem*()* family of string processing functions are freely available for use as part of the IDE. For the most part, these functions are used so often they are library routines written in hand-tweaked assembler code and are about as fast and memory-efficient as you can make them. They are worth learning about. More details can be found at:

http://www.tutorialspoint.com/c_standard_library/string_h.htm

If you look at the signature for the *memcpy()* function, you will find:

```
void *memcpy(void *dest, const void *src, size_n t)
```

Overlooking all of the confusing parts of the signature, what is says is: Grab *t* bytes of data starting at the memory address associated with *src* (the data source) and copy those bytes to the memory address associated with *dest* (the destination address). In other words, the function is a block copy from one location in memory to another memory location, moving *t* bytes of data.

Suppose you type the message When in the course of human events into the *msg[]* array. The display will eventually look like:

WHEN IN THE COUR

At this point, the LCD display is full. Now what? Keep in mind that the 'W' is at memory address *msg[0]* and the 'R' is at *msg[15]* so the display has every text cell filled in. Now look at the code for scrolling. The *memcpy()* function is written:

```
memcpy(temp, &s[index - SCREENWIDTH], SCREENWIDTH);
```

The *temp[]* variable is just a temporary string array we are using. Anytime you use a string name without brackets (i.e., [and]), it's the same as using the memory address of the zeroth element of that array. So *temp* in the statement above is the same as writing *&temp[0]*. The '&' operator in this context means address of, so writing *&temp[0]* means you want to use the memory address of the first element in the array. In fact, you could substitute *&temp[0]* in the code and it will work exactly as before.

Now, let's figure out what we're looking at. We have displayed the first 16 characters of the message, but want to display the 17th character. Because *index* is now 17, we want to display the 'S' in the message. This means that we have:

```
memcpy(temp, &s[index - SCREENWIDTH], SCREENWIDTH);
```

which is

```
memcpy(temp, &s[17 - 16], 16);
```

which reduces to

```
memcpy(temp, &s[1], 16);
```

which says we want to copy 16 characters starting with the memory address of *msg[1]* (not zero!) into *temp*. So, what this says is to copy 16 bytes from *msg[]* starting with the memory address of the 'H' in WHEN . When *memcpy()* finishes, this changes the display to:

HEN IN THE COURS

In other words, we have slid the message down by one character and let the first

character on the display fall off the left edge of the display, thus showing the next new character on the right edge of the screen. If you think about it, the LCD display acts like a 16-character wide window that we are sliding over the message as each letter is sent in Morse code.

Spend some time with this program, because it is probably one of the trickiest we will use. Don't be afraid to add *Serial.print()* messages in the code if you want to inspect the value of a variable at some point in time. It's a great way to learn! Also, if you have some small speakers that don't require much power (or headphones, earbuds), you could modify the circuit to use those instead. If you want to experiment, look at the Tone library:

https://code.google.com/archive/p/rogue-code/wikis/ToneLibraryDocumentation.wiki

make sure you read their warnings about connecting audio devices before you start experimenting.

EDITORS NOTE: All of the program code is now on the norcalqrp user group on yahoo groups. It is in the file section labeled "Arduino for the Terrified". This should have been placed there from the beginning. It was not due to my decision. Jack wasn't the hold up, it was me. I thought that it would be more conduicive to learning if people typed in the program. Bad decision. Doug, KI6DS