CHAPTER 4

Serial Communications

4.0 Introduction

Serial communications provide an easy and flexible way for your Arduino board to interact with your computer and other devices. This chapter explains how to send and receive information using this capability.

Chapter 1 described how to connect the Arduino serial port to your computer to upload sketches. The upload process sends data from your computer to Arduino and Arduino sends status messages back to the computer to confirm the transfer is working. The recipes here show how you can use this communication link to send and receive any information between Arduino and your computer or another serial device.

Serial communications are also a handy tool for debugging. You can send debug messages from Arduino to the computer and display them on your computer screen.

The Arduino IDE (described in Recipe 1.3) provides a Serial Monitor (shown in Figure 4-1) to display serial data received by Arduino.

You can also send data from the Serial Monitor to Arduino by entering text in the text box to the left of the Send button. Baud rate is selected using the drop-down box on the bottom right. You can use the drop down labeled "No line ending" to automatically send a carriage return or a combination of a carriage return and a line at the end of each message sent when clicking the Send button.

Your Arduino sketch can use the serial port to indirectly access (usually via a proxy program written in a language like Processing) all the resources (memory, screen, keyboard, mouse, network connectivity, etc.) that your computer has. Your computer can also use the serial link to interact with sensors or other devices connected to Arduino.

Implementing serial communications involves hardware and software. The hardware provides the electrical signaling between Arduino and the device it is talking to. The

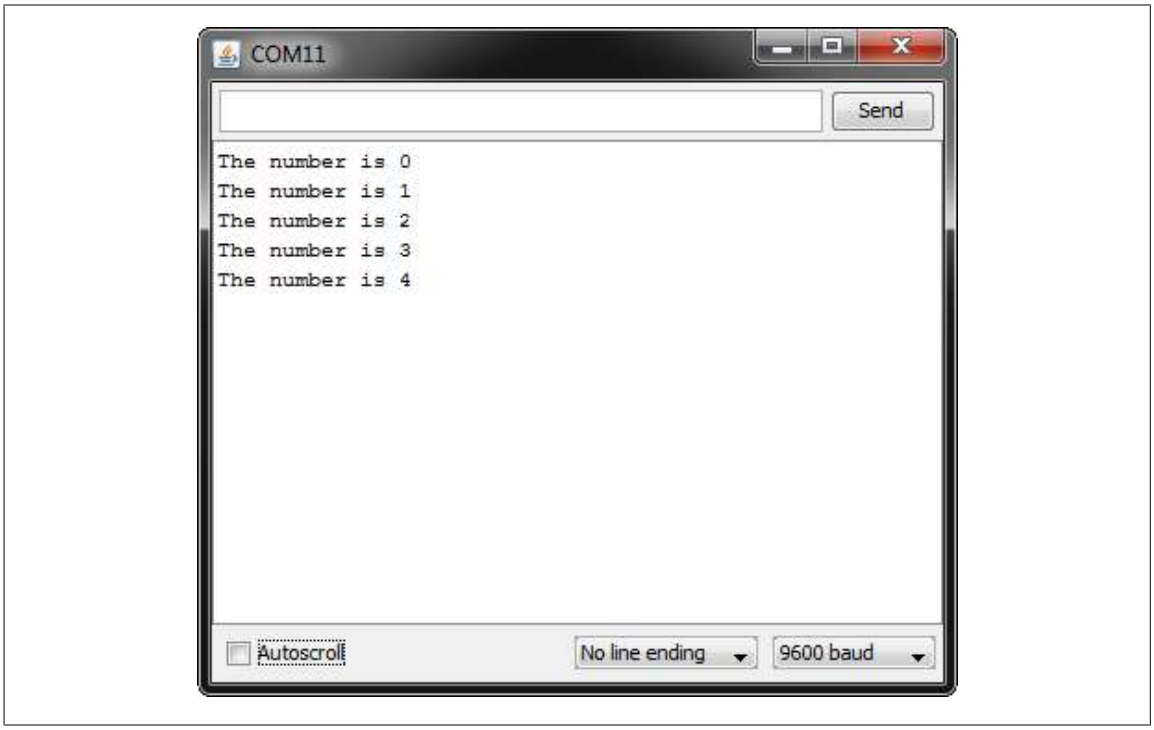

Figure 4-1. Arduino Serial Monitor screen

software uses the hardware to send bytes or bits that the connected hardware understands. The Arduino serial libraries insulate you from most of the hardware complexity, but it is helpful for you to understand the basics, especially if you need to troubleshoot any difficulties with serial communications in your projects.

Serial Hardware

Serial hardware sends and receives data as electrical pulses that represent sequential bits. The zeros and ones that carry the information that makes up a byte can be represented in various ways. The scheme used by Arduino is 0 volts to represent a bit value of 0, and 5 volts (or 3.3 volts) to represent a bit value of 1.

Using 0 volts (for 0) and 5 volts (for 1) is very common. This is referred to as the *TTL level* because that was how signals were represented in one of the first implementations of digital logic, called Transistor-Transistor Logic (TTL).

Boards including the Uno, Duemilanove, Diecimila, Nano, and Mega have a chip to convert the hardware serial port on the Arduino chip to Universal Serial Bus (USB) for connection to the hardware serial port. Other boards, such as the Mini, Pro, Pro Mini, Boarduino, Sanguino, and Modern Device Bare Bones Board, do not have USB support and require an adapter for connecting to your computer that converts TTL to USB. See *http://www.arduino.cc/en/Main/Hardware* for more details on these boards.

Some popular USB adapters include:

- Mini USB Adapter (*http://arduino.cc/en/Main/MiniUSB*)
- FTDI USB TTL Adapter (*http://www.ftdichip.com/Products/FT232R.htm*)
- Modern Device USB BUB board (*http://shop.moderndevice.com/products/usb-bub*)

Some serial devices use the RS-232 standard for serial connection. These usually have a nine-pin connector, and an adapter is required to use them with the Arduino. RS-232 is an old and venerated communications protocol that uses voltage levels not compatible with Arduino digital pins.

You can buy Arduino boards that are built for RS-232 signal levels, such as the Freeduino Serial v2.0 (*http://www.nkcelectronics.com/freeduino-serial-v20-board-kit-arduino-diecimila-compatib20.html*).

RS-232 adapters that connect RS-232 signals to Arduino 5V (or 3.3V) pins include the following:

- RS-232 to TTL 3V-5.5V adapter (*http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html*)
- P4 RS232 to TTL Serial Adapter Kits (*http://shop.moderndevice.com/products/p4*)
- RS232 Shifter SMD (*http://www.sparkfun.com/commerce/product_info.php?products_id=449*)

A standard Arduino has a single hardware serial port, but serial communication is also possible using software libraries to emulate additional ports (communication channels) to provide connectivity to more than one device. Software serial requires a lot of help from the Arduino controller to send and receive data, so it's not as fast or efficient as hardware serial.

The Arduino Mega has four hardware serial ports that can communicate with up to four different serial devices. Only one of these has a USB adapter built in (you could wire a USB-TTL adapter to any of the other serial ports). Table 4-1 shows the port names and pins used for all of the Mega serial ports.

Table 4-1. Arduino Mega serial ports

Port name	Transmit pin	Receive pin
Serial	1 (also USB)	0 (also USB)
Serial1	18	19
Serial2	16	17
Serial3	14	15

Software Serial

You will usually use the built-in Arduino serial library to communicate with the hardware serial ports. Serial libraries simplify the use of the serial ports by insulating you from hardware complexities.

Sometimes you need more serial ports than the number of hardware serial ports available. If this is the case, you can use an additional library that uses software to emulate serial hardware. Recipes 4.13 and 4.14 show how to use a software serial library to communicate with multiple devices.

Serial Message Protocol

The hardware or software serial libraries handle sending and receiving information. This information often consists of groups of variables that need to be sent together. For the information to be interpreted correctly, the receiving side needs to recognize where each message begins and ends. Meaningful serial communication, or any kind of machine-to-machine communication, can only be achieved if the sending and receiving sides fully agree how information is organized in the message. The formal organization of information in a message and the range of appropriate responses to requests is called a *communications protocol*.

Messages can contain one or more special characters that identify the start of the message—this is called the *header*. One or more characters can also be used to identify the end of a message—this is called the *footer*. The recipes in this chapter show examples of messages in which the values that make up the body of a message can be sent in either text or binary format.

Sending and receiving messages in text format involves sending commands and numeric values as human-readable letters and words. Numbers are sent as the string of digits that represent the value. For example, if the value is 1234, the characters 1, 2, 3, and 4 are sent as individual characters.

Binary messages comprise the bytes that the computer uses to represent values. Binary data is usually more efficient (requiring fewer bytes to be sent), but the data is not as human-readable as text, which makes it more difficult to debug. For example, Arduino represents 1234 as the bytes 4 and 210 (4 * 256 + 210 = 1234). If the device you are connecting to sends or receives only binary data, that is what you will have to use, but if you have the choice, text messages are easier to implement and debug.

There are many ways to approach software problems, and some of the recipes in this chapter show two or three different ways to achieve a similar result. The differences (e.g., sending text instead of raw binary data) may offer a different balance between simplicity and efficiency. Where choices are offered, pick the solution that you find easiest to understand and adapt—this will probably be the first solution covered. Alternatives may be a little more efficient, or they may be more appropriate for a specific protocol that you want to connect to, but the "right way" is the one you find easiest to get working in your project.

The Processing Development Environment

Some of the examples in this chapter use the Processing language to send and receive serial messages on a computer talking to Arduino.

Processing is a free open source tool that uses a similar development environment to Arduino. You can read more about Processing and download everything you need at the Processing website.

Processing is based on the Java language, but the Processing code samples in this book should be easy to translate into other environments that support serial communications. Processing comes with some example sketches illustrating communication between Arduino and Processing. SimpleRead is a Processing example that includes Arduino code. In Processing, select File→Examples→Libraries→Serial→SimpleRead to see an example that reads data from the serial port and changes the color of a rectangle when a switch connected to Arduino is pressed and released.

See Also

An Arduino RS-232 tutorial is available at *http://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232*. Lots of information and links are available at the Serial Port Central website, *http://www.lvr.com/serport.htm*.

In addition, a number of books on Processing are also available:

- *Getting Started with Processing: A Quick, Hands-on Introduction* by Casey Reas and Ben Fry (Make).
- *Processing: A Programming Handbook for Visual Designers and Artists* by Casey Reas and Ben Fry (MIT Press).
- *Visualizing Data* by Ben Fry (O'Reilly).
- *Processing: Creative Coding and Computational Art* by Ira Greenberg (Apress).
- *Making Things Talk* by Tom Igoe (Make). This book covers Processing and Arduino and provides many examples of communication code.

4.1 Sending Debug Information from Arduino to Your Computer

Problem

You want to send text and data to be displayed on your PC or Mac using the Arduino IDE or the serial terminal program of your choice.

Solution

This sketch prints sequential numbers on the Serial Monitor:

```
/*
 * SerialOutput sketch
 * Print numbers to the serial port
*/
void setup()
{
  Serial.begin(9600); // send and receive at 9600 baud
}

int number = 0;

void loop()
{
  Serial.print("The number is ");
  Serial.println(number);    // print the number

  delay(500); // delay half second between numbers
  number++; // to the next number
}
```

Connect Arduino to your computer just as you did in Chapter 1 and upload this sketch. Click the Serial Monitor icon in the IDE and you should see the output displayed as follows:

```
The number is 0
The number is 1
The number is 2
```

Discussion

To print text and numbers from your sketch, put the `Serial.begin(9600)` statement in `setup()`, and then use `Serial.print()` statements to print the text and values you want to see.

The Arduino Serial Monitor function can display serial data sent from Arduino. To start the Serial Monitor, click the Serial Monitor toolbar icon as shown in Figure 4-2. A new window will open for displaying output from Arduino.

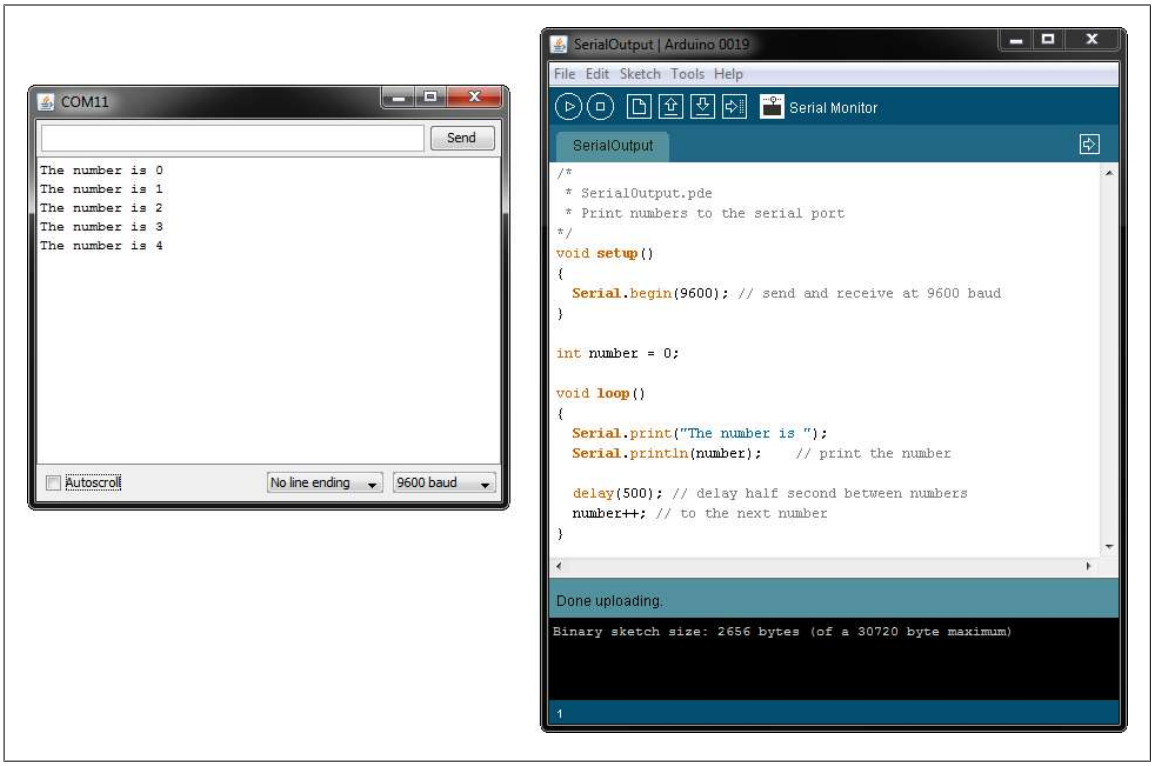

Figure 4-2. Clicking the Serial Monitor icon to see serial output

Your sketch must call the `Serial.begin()` function before it can use serial input or output. The function takes a single parameter: the desired communication speed. You must use the same speed for the sending side and the receiving side, or you will see gobbledygook (or nothing at all) on the screen. This example and most of the others in this book use a speed of 9,600 baud (*baud* is a measure of the number of bits transmitted per second). The 9,600 baud rate is approximately 1,000 characters per second. You can send at lower or higher rates (the range is 300 to 115,200), but make sure both sides use the same speed. The Serial Monitor sets the speed using the baud rate drop down (at the bottom right of the Serial Monitor window in Figure 4-2). If your output looks something like this:

```
`3??f<ÌxÌ▯▯▯ü`³??f<
```

you should check that the selected baud rate on your computer matches the rate set by `Serial.begin()` in your sketch.

If your sending and receiving serial speeds are set correctly but you are still getting unreadable text, check that you have the correct board selected in the IDE Tools→Board menu. If you have selected the wrong board, change it to the correct one and upload to the board again.

You can display text using the `Serial.print()` function. Strings (text within double quotes) will be printed as is (but without the quotes). For example, the following code:

```
Serial.print("The number is ");
```

prints this:

```
The number is
```

The values (numbers) that you print depend on the type of variable; see Recipe 4.2 for more about this. But for now, printing an integer will print its numeric value, so if the variable `number` is `1`, the following code:

```
Serial.println(number);
```

will print this:

```
1
```

In the example sketch, the number printed will be `0` when the loop starts and will increase by one each time through the loop. The `ln` at the end of `println` causes the next print statement to start on a new line.

That should get you started printing text and the decimal value of integers. See Recipe 4.2 for more detail on print formatting options.

You may want to consider a third-party terminal program that has more features than Serial Monitor. Displaying data in text or binary format (or both), displaying control characters, and logging to a file are just a few of the additional capabilities available from the many third-party terminal programs. Here are some that have been recommended by Arduino users:

CuteCom
: An open source terminal program for Linux

Bray Terminal
: A free executable for the PC

GNU screen
: An open source virtual screen management program that supports serial communications; included with Linux and Mac OS X

moserial
: Another open source terminal program for Linux

PuTTY
: An open source SSH program for Windows; supports serial communications

RealTerm
: An open source terminal program for the PC

ZTerm
: A shareware program for the Mac

In addition, an article in the Arduino wiki explains how to configure Linux to communicate with Arduino using TTY (see *http://www.arduino.cc/playground/Interfacing/LinuxTTY*).

You can use a liquid crystal display as a serial output device, although it will be very limited in functionality. Check the documentation to see how your display handles carriage returns, as some displays may not automatically advance to a new line after `println` statements.

See Also

The Arduino LiquidCrystal library for text LCDs uses underlying print functionality similar to the Serial library, so you can use many of the suggestions covered in this chapter with that library (see Chapter 11).

4.2 Sending Formatted Text and Numeric Data from Arduino

Problem

You want to send serial data from Arduino displayed as text, decimal values, hexadecimal, or binary.

Solution

You can print data to the serial port in many different formats; here is a sketch that demonstrates all the format options:

```
/*
 * SerialFormatting
 * Print values in various formats to the serial port
*/
char chrValue = 65;  // these are the starting values to print
int intValue  = 65;
float floatValue = 65.0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.println("chrValue: ");
  Serial.println(chrValue);
  Serial.println(chrValue,BYTE);
  Serial.println(chrValue,DEC);
```

```
  Serial.println("intValue: ");
  Serial.println(intValue);
  Serial.println(intValue,BYTE);
  Serial.println(intValue,DEC);
  Serial.println(intValue,HEX);
  Serial.println(intValue,OCT);
  Serial.println(intValue,BIN);

  Serial.println("floatValue: ");
  Serial.println(floatValue);

  delay(1000); // delay a second between numbers
  chrValue++;  // to the next value
  intValue++;
}
```

The output (condensed here onto a few lines) is as follows:

```
chrValue: A  A 65
intValue: 65 A 65 41 101 1000001
floatValue: 65.00

chrValue: B  B 66
intValue: 66 B 66 42 102 1000010
```

Discussion

Printing a text string is simple: `Serial.print("hello world");` sends the text string "hello world" to a device at the other end of the serial port. If you want your output to print a new line after the output, use `Serial.println()` instead of `Serial.print()`.

Printing numeric values can be more complicated. The way that byte and integer values are printed depends on the type of variable and an optional formatting parameter. The Arduino language is very easygoing about how you can refer to the value of different data types (see Recipe 2.2 for more on data types). But this flexibility can be confusing, because even when the numeric values are similar, the compiler considers them to be separate types with different behaviors. For example, printing a `char` will not necessarily produce the same output as printing an `int` of the same value.

Here are some specific examples; all of them create variables that have similar values:

```
char asciiValue = 'A';   // ASCII A has a value of 65
char chrValue = 65;      // an 8 bit character, this also is ASCII 'A'
int intValue = 65;       // a 16 bit integer set to a value of 65
float floatValue = 65.0; // float with a value of 65
```

Table 4-2 shows what you will see when you print variables using Arduino routines.

Table 4-2. Output formats using Serial.print

Data type	Print (val)	Print (val,DEC)	Print (val,BYTE)	Print (val,HEX)	Print (val,OCT)	Print (val,BIN)
char	A	65	A	41	101	1000001
int	65	65	A	41	101	1000001
long	Format of long is the same as int					
float	65.00	Formatting not supported for floating-point values				
double	65.00	double is the same as float				

The sketch in this recipe uses a separate line of source code for each print statement. This can make complex print statements bulky. For example, to print the following line:

```
At 5 seconds: speed = 17, distance = 120
```

you'd typically have to code it like this:

```
Serial.print("At ");
Serial.print(seconds);
Serial.print(" seconds: speed = ");
Serial.print(speed);
Serial.print(", distance = ");
Serial.println(distance);
```

That's a lot of code lines for a single line of output. You could combine them like this:

```
Serial.print("At "); Serial.print(seconds); Serial.print(" seconds, speed = ");
Serial.print(speed); Serial.print(", distance = ");Serial.println(distance);
```

Or you could use the *insertion-style* capability of the compiler used by Arduino to format your print statements. You can take advantage of some advanced C++ capabilities (streaming insertion syntax and templates) that you can use if you declare a streaming template in your sketch. This is most easily achieved by including the Streaming library developed by Mikal Hart. You can read more about this library and download the code from Mikal's website.

If you use the Streaming library, the following gives the same output as the lines shown earlier:

```
Serial << "At " << seconds << " seconds, speed = " << speed << ", distance =
" << distance;
```

See Also

Chapter 2 provides more information on data types used by Arduino. The Arduino web reference at *http://arduino.cc/en/Reference/HomePage* covers the serial commands, and the Arduino web reference at *http://www.arduino.cc/playground/Main/StreamingOutput* covers streaming (insertion-style) output.

4.3 Receiving Serial Data in Arduino

Problem

You want to receive data on Arduino from a computer or another serial device; for example, to have Arduino react to commands or data sent from your computer.

Solution

It's easy to receive 8-bit values (chars and bytes), because the `Serial` functions use 8-bit values. This sketch receives a digit (single characters 0 through 9) and blinks the LED on pin 13 at a rate proportional to the received digit value:

```
/*
 * SerialReceive sketch
 * Blink the LED at a rate proportional to the received digit value
*/
const int ledPin = 13; // pin the LED is connected to
int   blinkRate=0;     // blink rate stored in this variable

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud
  pinMode(ledPin, OUTPUT); // set this pin as output
}

void loop()
{
  if ( Serial.available()) // Check to see if at least one character is available
  {
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9') // is this an ascii digit between 0 and 9?
    {
       blinkRate = (ch - '0');      // ASCII value converted to numeric value
       blinkRate = blinkRate * 100; // actual blinkrate is 100 mS times received
digit
    }
  }
  blink();
}

// blink the LED with the on and off times determined by blinkRate
void blink()
{
  digitalWrite(ledPin,HIGH);
  delay(blinkRate); // delay depends on blinkrate value
  digitalWrite(ledPin,LOW);
  delay(blinkRate);
}
```

Upload the sketch and send messages using the Serial Monitor. Open the Serial Monitor by clicking the Monitor icon (see Recipe 4.1) and type a digit in the text box at the top

of the Serial Monitor window. Clicking the Send button will send the character typed into the text box; you should see the blink rate change.

Discussion

Converting the received ASCII characters to numeric values may not be obvious if you are not familiar with the way ASCII represents characters. The following converts the character `ch` to its numeric value:

```
blinkRate = (ch - '0');   // ASCII value converted to numeric value
```

This is done by subtracting 48, because 48 is the ASCII value of the digit 0. For example, if `ch` is representing the character 1, its ASCII value is 49. The expression `49- '0'` is the same as `49-48`. This equals 1, which is the numeric value of the character 1.

In other words, the expression `(ch - '0')` is the same as `(ch - 48)`; this converts the ASCII value of the variable `ch` to a numeric value.

To get a clearer idea of the relationship between the ASCII values of characters representing the digits 0 through 9 and their actual numeric values, see the ASCII table in Appendix G.

Receiving numbers with more than one digit involves accumulating characters until a character that is not a valid digit is detected. The following code uses the same `setup()` and `blink()` functions as those shown earlier, but it gets digits until the newline character is received. It uses the accumulated value to set the blink rate.

The newline character (ASCII value 10) can be appended automatically each time you click Send. The Serial Monitor has a drop-down box at the bottom of the Serial Monitor screen (see Figure 4-1); change the option from "No line ending" to "Newline."

Change the code that the `loop` code follows. Enter a value such as `123` into the Monitor text box and click Send, and the blink delay will be set to 123 milliseconds:

```
int value;

void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9') // is this an ascii digit between 0 and 9?
    {
       value = (value * 10) + (ch - '0'); // yes, accumulate the value
    }
    else if (ch == 10)  // is the character the newline character
    {
       blinkRate = value;  // set blinkrate to the accumulated value
       Serial.println(blinkRate);
       value = 0; // reset val to 0 ready for the next sequence of digits
```

```
    }
  }
  blink();
}
```

Each digit is converted from its ASCII value to its numeric value. Because the numbers are decimal numbers (base 10), each successive number is multiplied by 10. For example, the value of the number 234 is 2 * 100 + 3 * 10 + 4. The code to accomplish that is:

```
if(ch >= '0' && ch <= '9') // is this an ascii digit between 0 and 9?
{
   value = (value * 10) + (ch - '0'); // yes, accumulate the value
}
```

If you want to handle negative numbers, your code needs to recognize the minus (`'-'`) sign. For example:

```
int value = 0;
int sign = 1;

void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9') // is this an ascii digit between 0 and 9?
       value = (value * 10) + (ch - '0'); // yes, accumulate the value
    else if( ch == '-')
       sign = -1;
    else // this assumes any char not a digit or minus sign terminates the value
    {
       value = value * sign ;  // set value to the accumulated value
       Serial.println(value);
       value = 0; // reset value to 0 ready for the next sequence of digits
       sign = 1;
    }
  }
}
```

Another approach to converting text strings representing numbers is to use the C language conversion function called `atoi` (for `int` variables) or `atol` (for `long` variables). These obscurely named functions convert a string into integers or long integers. To use them you have to receive and store the entire string in a character array before you can call the conversion function.

This code fragment terminates the incoming digits on any character that is not a digit (or if the buffer is full):

```
const int MaxChars = 5; // an int string contains up to 5 digits and
                        // is terminated by a 0 to indicate end of string
char strValue[MaxChars+1]; // must be big enough for digits and terminating null
int index = 0;         // the index into the array storing the received digits

void loop()
```

```
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(index <  MaxChars && ch >= '0' && ch <= '9'){
      strValue[index++] = ch; // add the ASCII character to the string;
    }
    else
    {
      // here when buffer full or on the first non digit
      strValue[index] = 0;        // terminate the string with a 0
      blinkRate = atoi(strValue);  // use atoi to convert the string to an int
      index = 0;
    }
  }
  blink();
}
```

`strValue` is a numeric string built up from characters received from the serial port.

See Recipe 2.6 for information about character strings.

`atoi` (short for ASCII to integer) is a function that converts a character string to an integer (`atol` converts to a long integer).

See Also

A web search for "atoi" or "atol" provides many references to these functions. Also see the Wikipedia reference at *http://en.wikipedia.org/wiki/Atoi*.

4.4 Sending Multiple Text Fields from Arduino in a Single Message

Problem

You want to send a message that contains more than one piece of information (field). For example, your message may contain values from two or more sensors. You want to use these values in a program such as Processing, running on your PC or Mac.

Solution

The easiest way to do this is to send a text string with all the fields separated by a delimiting (separating) character, such as a comma:

```
// CommaDelimitedOutput sketch

void setup()
```

```
{
  Serial.begin(9600);
}

void loop()
{
  int value1 = 10;    // some hardcoded values to send
  int value2 = 100;
  int value3 = 1000;

  Serial.print('H'); // unique header to identify start of message
  Serial.print(",");
  Serial.print(value1,DEC);
  Serial.print(",");
  Serial.print(value2,DEC);
  Serial.print(",");
  Serial.print(value3,DEC);
  Serial.print(",");  // note that a comma is sent after the last field
  Serial.println();  // send a cr/lf
  delay(100);
}
```

Here is the Processing sketch that reads this data from the serial port:

```
//CommaDelimitedInput.pde (Processing Sketch)

import processing.serial.*;

Serial myPort;        // Create object from Serial class
char HEADER = 'H';    // character to identify the start of a message
short LF = 10;        // ASCII linefeed
short portIndex = 0;  // select the com port, 0 is the first port

void setup() {
  size(200, 200);

  // WARNING!
  // If necessary, change the definition of portIndex at the top of this
  // sketch to the desired serial port.
  //
  println(Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this,Serial.list()[portIndex], 9600);
}

void draw() {

}

void serialEvent(Serial p)
{
  String message = myPort.readStringUntil(LF); // read serial data
```

```
  if(message != null)
  {
    print(message);
    String [] data  = message.split(","); // Split the comma-separated message
    if(data[0].charAt(0) == HEADER)       // check for header character in the
first field
    {
      for( int i = 1; i < data.length-1; i++) // skip the header and terminating
cr and lf
      {
        int value = Integer.parseInt(data[i]);
        println("Value" +  i + " = " + value);  //Print the value for each field
      }
      println();
    }
  }
}
```

Discussion

The code in this recipe's Solution will send the following text string to the serial port (`\r` indicates a carriage return and `\n` indicates a line feed):

```
H10,100,1000,\r\n
```

You must choose a separating character that will never occur within actual data; if your data consists only of numeric values, a comma is a good choice for a delimiter. You may also want to ensure that the receiving side can determine the start of a message to make sure it has all the data for all the fields. You do this by sending a header character to indicate the start of the message. The header character must also be unique; it should not appear within any of the data fields and it must also be different from the separator character. The example here uses an uppercase *H* to indicate the start of the message. The message consists of the header, three comma-separated numeric values as ASCII strings, and a carriage return and line feed.

The carriage return and line-feed characters are sent whenever Arduino prints using the `println()` function, and this is used to help the receiving side know that the full message string has been received. A comma is sent after the last numerical value to aid the receiving side in detecting the end of the value.

The Processing code reads the message as a string and uses the Java `split()` method to create an array from the comma-separated fields.

In most cases, the first serial port will be the one you want when using a Mac and the last serial port will be the one you want when using Windows. The Processing sketch includes code that shows the ports available and the one currently selected—check that this is the port connected to Arduino.

See Also

The Processing website provides more information on installing and using this programming environment. See *http://processing.org/*.

4.5 Receiving Multiple Text Fields in a Single Message in Arduino

Problem

You want to receive a message that contains more than one field. For example, your message may contain an identifier to indicate a particular device (such as a motor or other actuator) and what value (such as speed) to set it to.

Solution

Arduino does not have the `split()` function used in the Processing code in Recipe 4.4, but the functionality can be implemented as shown in this recipe. The following code receives a message with three numeric fields separated by commas. It uses the technique described in Recipe 4.4 for receiving digits, and it adds code to identify comma-separated fields and store the values into an array:

```
/*
 * SerialReceiveMultipleFields sketch
 * This code expects a message in the format: 12,345,678
 * This code requires a newline character to indicate the end of the data
 * Set the serial monitor to send newline characters
 */

const int NUMBER_OF_FIELDS = 3; // how many comma separated fields we expect
int fieldIndex = 0;            // the current field being received
int values[NUMBER_OF_FIELDS];   // array holding values for all the fields



void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud
}

void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9') // is this an ascii digit between 0 and 9?
    {
      // yes, accumulate the value
      values[fieldIndex] = (values[fieldIndex] * 10) + (ch - '0');
    }
```

```
    else if (ch == ',')  // comma is our separator, so move on to the next field
    {
      if(fieldIndex < NUMBER_OF_FIELDS-1)
        fieldIndex++;   // increment field index
    }
    else
    {
      // any character not a digit or comma ends the acquisition of fields
      // in this example it's the newline character sent by the Serial Monitor
      Serial.print( fieldIndex +1);
      Serial.println(" fields received:");
      for(int i=0; i <= fieldIndex; i++)
      {
        Serial.println(values[i]);
        values[i] = 0; // set the values to zero, ready for the next message
      }
      fieldIndex = 0;  // ready to start over
    }
  }
}
```

Discussion

This sketch accumulates values (as explained in Recipe 4.3), but here each value is added to an array (which must be large enough to hold all the fields) when a comma is received. A character other than a digit or comma (such as the newline character; see Recipe 4.3) triggers the printing of all the values that have been stored in the array.

Another approach is to use a library called TextFinder, which is available from the Arduino Playground or from the website for this book. TextFinder was created to extract information from web streams (see Chapter 15), but it works just as well with serial data. The following sketch uses TextFinder to provide similar functionality to the previous sketch:

```
#include <TextFinder.h>

TextFinder finder(Serial);

const int NUMBER_OF_FIELDS = 3; // how many comma-separated fields we expect
int fieldIndex = 0;             // the current field being received
int values[NUMBER_OF_FIELDS];   // array holding values for all the fields


void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud
}

void loop()
{

  for(fieldIndex = 0; fieldIndex  < 3; fieldIndex ++)
  {
    values[fieldIndex] = finder.getValue(); // get a numeric value
```

```
    }
    Serial.print( fieldIndex);
    Serial.println(" fields received:");
    for(int i=0; i <  fieldIndex; i++)
    {
       Serial.println(values[i]);
    }
    fieldIndex = 0;  // ready to start over
  }
```

You can download the TextFinder library from *http://www.arduino.cc/playground/Code/TextFinder*.

Here is a summary of the methods supported by TextFinder (not all are used in the preceding example):

`boolean find(char *target);`
: Reads from the stream until the given target is found. It returns `true` if the target string is found. A return of `false` means the data has not been found anywhere in the stream and that there is no more data available. Note that TextFinder takes a single pass through the stream; there is no way to go back to try to find or get something else (see the `findUntil` method).

`boolean findUntil(char *target, char *terminate);`
: Similar to the `find` method, but the search will stop if the terminate string is found. Returns `true` only if the target is found. This is useful to stop a search on a keyword or terminator. For example:

```
finder.findUntil("target", "\n");
```

  will try to seek to the string `"value"`, but will stop at a newline character so that your sketch can do something else if the target is not found.

`long getValue();`
: Returns the first valid (long) integer value. Leading characters that are not digits or a minus sign are skipped. The integer is terminated by the first nondigit character following the number. If no digits are found, the function returns `0`.

`long getValue(char skipChar);`
: Same as `getValue`, but the given `skipChar` within the numeric value is ignored. This can be helpful when parsing a single numeric value that uses a comma between blocks of digits in large numbers, but bear in mind that text values formatted with commas cannot be parsed as a comma-separated string.

`float getFloat();`
: The `float` version of `getValue`.

`int getString(char *pre_string,char *post_string,char *buf,int length);`
: Finds the `pre_string` and then puts the incoming characters into the given buffer until the `post_string` is detected. The end of the string is determined by a match of a character to the first `char post_string`. Strings longer than the given `length`

are truncated to fit. The function returns the number of characters placed in the buffer (0 means no valid data was found).

See Also

Chapter 15 provides more examples of TextFinder used to find and extract data from a stream.

4.6 Sending Binary Data from Arduino

Problem

You need to send data in binary format, because you want to pass information with the fewest number of bytes or because the application you are connecting to only handles binary data.

Solution

This sketch sends a header followed by two integer (16-bit) values as binary data. The values are generated using the Arduino `random` function (see Recipe 3.11):

```
/*
 * SendBinary sketch
 * Sends a header followed by two random integer values as binary data.
*/

int intValue;    // an integer value (16 bits)

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.print('H'); // send a header character

  // send a random integer
  intValue = random(599); // generate a random number between 0 and 599
  // send the two bytes that comprise an integer
  Serial.print(lowByte(intValue), BYTE);  // send the low byte
  Serial.print(highByte(intValue), BYTE); // send the high byte

  // send another random integer
  intValue = random(599); // generate a random number between 0 and 599
  // send the two bytes that comprise an integer
  Serial.print(lowByte(intValue), BYTE);  // send the low byte
  Serial.print(highByte(intValue), BYTE); // send the high byte

  delay(1000);
}
```

Discussion

Sending binary data requires careful planning, because you will get gibberish unless the sending side and the receiving side understand and agree exactly how the data will be sent. Unlike text data, where the end of a message can be determined by the presence of the terminating carriage return (or another unique character you pick), it may not be possible to tell when a binary message starts or ends by looking just at the data—data that can have any value can therefore have the value of a header or terminator character.

This can be overcome by designing your messages so that the sending and receiving sides know exactly how many bytes are expected. The end of a message is determined by the number of bytes sent rather than detection of a specific character. This can be implemented by sending an initial value to say how many bytes will follow. Or you can fix the size of the message so that it's big enough to hold the data you want to send. Doing either of these is not always easy, as different platforms and languages can use different sizes for the binary data types—both the number of bytes and their order may be different from Arduino. For example, Arduino defines an `int` as two bytes, but Processing (Java) defines an `int` as four bytes (`short` is the Java type for a 16-bit integer). Sending an `int` value as text (as seen in earlier text recipes) simplifies this problem because each individual digit is sent as a sequential digit (just as the number is written). The receiving side recognizes when the value has been completely received by a carriage return or other nondigit delimiter. Binary transfers can only know about the composition of a message if it is defined in advance or specified in the message.

This recipe's Solution requires an understanding of the data types on the sending and receiving platforms and some careful planning. Recipe 4.7 shows example code using the Processing language to receive these messages.

Sending single bytes is easy; use `Serial.print(byteVal)`. To send an integer from Arduino you need to send the low and high bytes that make up the integer (see Recipe 2.2 for more on data types). You do this using the `lowByte` and `highByte` functions (see Recipe 3.14):

```
Serial.print(lowByte(intValue), BYTE);
Serial.print(highByte(intValue), BYTE);
```

The preceding code sends the low byte followed by the high byte. The code can also be written without the `BYTE` parameter (see Recipe 4.2), but using the parameter is a useful reminder (when you come back later to make changes, or for others who may read your code) that your intention is to send bytes rather than ASCII characters.

Sending a long integer is done by breaking down the four bytes that comprise a `long` in two steps. The `long` is first broken into two 16-bit integers; each is then sent using the method for sending integers described earlier:

```
int longValue = 1000;
int intValue;
```

First you send the lower 16-bit integer value:

```
intValue = longValue && 0xFFFF;  // get the value of the lower 16 bits
Serial.print(lowByte(intVal), BYTE);
Serial.print(highByte(intVal), BYTE);
```

Then you send the higher 16-bit integer value:

```
intValue = longValue >> 16;  // get the value of the higher 16 bits
Serial.print(lowByte(intVal), BYTE);
Serial.print(highByte(intVal), BYTE);
```

You may find it convenient to create functions to send the data. Here is a function that uses the code shown earlier to print a 16-bit integer to the serial port:

```
// function to send the given integer value to the serial port
void sendBinary(int value)
{
  // send the two bytes that comprise a two byte (16 bit) integer
  Serial.print(lowByte(value), BYTE);  // send the low byte
  Serial.print(highByte(value), BYTE); // send the high byte
}
```

The following function sends the value of a `long` (4-byte) integer by first sending the two low (rightmost) bytes, followed by the high (leftmost) bytes:

```
// function to send the given long integer value to the serial port
void sendBinary(long value)
{
  // first send the low 16 bit integer value
  int temp = value && 0xFFFF;  // get the value of the lower 16 bits
  sendBinary(temp);
  // then send the higher 16 bit integer value:
  temp = value >> 16;  // get the value of the higher 16 bits
  sendBinary(temp);
}
```

These functions to send binary `int` and `long` values have the same name: `sendBinary`. The compiler distinguishes them by the type of value you use for the parameter. If your code calls `printBinary` with a 2-byte value, the version declared as `void sendBinary(int value)` will be called. If the parameter is a `long` value, the version declared as `void sendBinary(long value)` will be called. This behavior is called *function overloading*. Recipe 4.2 provides another illustration of this; the different functionality you saw in `Serial.print` is due to the compiler distinguishing the different variable types used.

You can also send binary data using structures. Structures are a mechanism for organizing data, and if you are not already familiar with their use you may be better off sticking with the solutions described earlier. For those who are comfortable with the concept of structure pointers, the following is a function that will send the bytes within a structure to the serial port as binary data:

```
void sendStructure( char *structurePointer, int structureLength)
{
  int i;
```

```
  for (i = 0 ; i < structureLength ; i++)
    serial.print(structurePointer[i], BYTE);
  }

sendStructure((char *)&myStruct, sizeof(myStruct));
```

Sending data as binary bytes is more efficient than sending data as text, but it will only work reliably if the sending and receiving sides agree exactly on the composition of the data. Here is a summary of the important things to check when writing your code:

Variable size
: Make sure the size of the data being sent is the same on both sides. An integer is 2 bytes on Arduino, 4 bytes on most other platforms. Always check your programming language's documentation on data type size to ensure agreement. There is no problem with receiving a 2-byte Arduino integer as a 4-byte integer in Processing as long as Processing expects to get only two bytes. But be sure that the sending side does not use values that will overflow the type used by the receiving side.

Byte order
: Make sure the bytes within an `int` or `long` are sent in the same order expected by the receiving side.

Synchronization
: Ensure that your receiving side can recognize the beginning and end of a message. If you start listening in the middle of a transmission stream, you will not get valid data. This can be achieved by sending a sequence of bytes that won't occur in the body of a message. For example, if you are sending binary values from `analogRead`, these can only range from 0 to 1,023, so the most significant byte must be less than 4 (the `int` value of 1,023 is stored as the bytes 3 and 255); therefore, there will never be data with two consecutive bytes greater than 3. So, sending two bytes of 4 (or any value greater than 3) cannot be valid data and can be used to indicate the start or end of a message.

Structure packing
: If you send or receive data as structures, check your compiler documentation to make sure the *packing* is the same on both sides. Packing is the padding that a compiler uses to align data elements of different sizes in a structure.

Flow control
: Either choose a transmission speed that ensures that the receiving side can keep up with the sending side, or use some kind of *flow control*. Flow control is a handshake that tells the sending side that the receiver is ready to get more data.

See Also

Chapter 2 provides more information on the variable types used in Arduino sketches.

Also, check the Arduino references for `lowByte` at *http://www.arduino.cc/en/Reference/LowByte* and `highByte` at *http://www.arduino.cc/en/Reference/HighByte*.

The Arduino compiler packs structures on byte boundaries; see the documentation for the compiler you use on your computer to set it for the same packing. If you are not clear on how to do this, you may want to avoid using structures to send data.

For more on flow control, see *http://en.wikipedia.org/wiki/Flow_control*.

4.7 Receiving Binary Data from Arduino on a Computer

Problem

You want to respond to binary data sent from Arduino in a programming language such as Processing. For example, you want to respond to Arduino messages sent in Recipe 4.6.

Solution

This recipe's Solution depends on the programming environment you use on your PC or Mac. If you don't already have a favorite programming tool and want one that is easy to learn and works well with Arduino, Processing is an excellent choice.

Here are the two lines of Processing code to read a byte, taken from the Processing `SimpleRead` example (see this chapter's introduction):

```
if ( myPort.available() > 0) {  // If data is available,
  val = myPort.read();          // read it and store it in val
```

As you can see, this is very similar to the Arduino code you saw in earlier recipes.

The following is a Processing sketch that sets the size of a rectangle proportional to the integer values received from the Arduino sketch in Recipe 4.6:

```
/*
 * ReceiveBinaryData_P
 *
 * portIndex must be set to the port connected to the Arduino
*/
import processing.serial.*;

Serial myPort;        // Create object from Serial class
short portIndex = 1;  // select the com port, 0 is the first port

char HEADER = 'H';
int value1, value2;         // Data received from the serial port

void setup()
{
  size(600, 600);
  // Open whatever serial port is connected to Arduino.
  String portName = Serial.list()[portIndex];
  println(Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this, portName, 9600);
```

```
}

void draw()
{
  // read the header and two binary *(16 bit) integers:
  if ( myPort.available() >= 5)  // If at least 5 bytes are available,
  {
    if( myPort.read() == HEADER) // is this the header
    {
      value1 = myPort.read();                 // read the least significant byte
      value1 =  myPort.read() * 256 + value1; // add the most significant byte

      value2 = myPort.read();                 // read the least significant byte
      value2 =  myPort.read() * 256 + value2; // add the most significant byte

      println("Message received: " + value1 + "," + value2);
    }
  }
  background(255);             // Set background to white
  fill(0);                     // set fill to black
// draw rectangle with coordinates based on the integers received from Arduino
  rect(0, 0, value1,value2);
}
```

Discussion

The Processing language influenced Arduino, and the two are intentionally similar. The `setup` function in Processing is used to handle one-time initialization, just like in Arduino. Processing has a display window, and `setup` sets its size to 600 × 600 pixels with the call to `size(600,600)`.

The line `String portName = Serial.list()[portIndex];` selects the serial port—in Processing, all available serial ports are contained in the `Serial.list` object and this example uses the value of a variable called `portIndex`. `println(Serial.list())` prints all the available ports, and the line `myPort = new Serial(this, portName, 9600);` opens the port selected as `portName`. Ensure that you set `portIndex` to the serial port that is connected to your Arduino.

The `draw` function in Processing works like `loop` in Arduino; it is called repeatedly. The code in `draw` checks if data is available on the serial port; if so, bytes are read and converted to the integer value represented by the bytes. A rectangle is drawn based on the integer values received.

See Also

You can read more about Processing on the Processing website.

4.8 Sending Binary Values from Processing to Arduino

Problem

You want to send binary bytes, integers, or long values from Processing to Arduino. For example, you want to send a message consisting of a message identifier "tag," an index (perhaps indicating a particular device attached to Arduino), and a 16-bit value.

Solution

Use this code:

```
/* SendingBinaryToArduino
 * Language: Processing
 */
import processing.serial.*;

Serial myPort;  // Create object from Serial class
public static final char HEADER = '|';
public static final char MOUSE  = 'M';

void setup()
{
  size(200, 400);
  String portName = Serial.list()[0];
  myPort = new Serial(this, portName, 9600);
}

void draw(){
}

void serialEvent(Serial p) {
  // handle incoming serial data
  String inString = myPort.readStringUntil('\n');
  if(inString != null) {
      println( inString );   // echo text string from Arduino
  }
}
```

When the mouse is clicked in the Processing window, `sendMessage` will be called with `index` equal to the vertical position of the mouse in the window when clicked and `value` equal to the horizontal position. The window size was set to 200,400, so `index` would fit into a single byte and `value` would fit into two bytes:

```
void mousePressed() {
  int index = mouseY;
  int value = mouseX;
  sendMessage(MOUSE, index, value);
}
```

sendMessage sends a header, tag, and index as single bytes. It sends the value as two bytes, with the most significant byte first:

```
void sendMessage(char tag, int index, int value){
  // send the given index and value to the serial port
  myPort.write(HEADER);
  myPort.write(tag);
  myPort.write(index);
  char c = (char)(value / 256); // msb
  myPort.write(c);
  c = (char)(value & 0xff);  // lsb
  myPort.write(c);
}
```

The Arduino code to receive this and echo the results back to Processing is:

```
//BinaryDataFromProcessing
```

The next three defines must mirror the definitions used in the sending program:

```
#define HEADER        '|'
#define MOUSE         'M'
#define MESSAGE_BYTES  5  // the total bytes in a message


void setup()
{
  Serial.begin(9600);
}

void loop(){
```

The check to ensure that at least MESSAGE_BYTES have been received ensures that we don't try to process the message until all the required data is available:

```
  if ( Serial.available() >= MESSAGE_BYTES)
  {
```

Only read the rest of the message if a valid header has been received:

```
    if( Serial.read() == HEADER)
    {
      char tag = Serial.read();
      if(tag == MOUSE)
      {
        int index = Serial.read(); // this was sent as a char
but it's ok to use it as an int
```

The next lines convert the two bytes back to an integer. Serial.read() * 256; restores the most significant byte to its original value. Compare this to Processing code that sent the two bytes comprising the value:

```
        int val = Serial.read() * 256;
        val = val + Serial.read();
        Serial.print("Received mouse msg, index = ");
        Serial.print(index);
        Serial.print(", value  ");
```

```
        Serial.println(val);
      }
      else
      {
```

If the code gets here, the tag was not recognized. This helps you to ignore data that may be incomplete or corrupted:

```
        Serial.print("got message with unknown tag ");
        Serial.println(tag);
      }
    }
  }
}
```

Discussion

This code is similar to the Processing code in the previous recipes, with the addition of a function called `sendMessage`. In this example, the function is called with three parameters: a tag, an index, and a value. The function first sends the header character to identify the start of the message. Then the single byte index is sent, followed by the two bytes that comprise the integer value. You can make your own version of this function to send the combination of values that you need for your application.

4.9 Sending the Value of Multiple Arduino Pins

Problem

You want to send groups of binary bytes, integers, or long values from Arduino. For example, you may want to send the values of the digital and analog pins to Processing.

Solution

This recipe sends a header followed by an integer containing the bit values of digital pins 2 to 13. This is followed by six integers containing the values of analog pins 0 through 5. Chapter 5 has many recipes that set values on the analog and digital pins that you can use to test this sketch:

```
/*
 * SendBinaryFields
 * Sends digital and analog pin values as binary data
 */

const char HEADER = 'H';  // a single character header to indicate the start
of a message
// these are the values that will be sent in binary format

void setup()
{
  Serial.begin(9600);
  for(int i=2; i <= 13; i++)
```

```
  {
    pinMode(i, INPUT);       // set pins 2 through 13 to inputs
    digitalWrite(i, HIGH);   // turn on pull-ups
  }
}

void loop()
{
  Serial.print(HEADER,BYTE); // send the header
  // put the bit values of the pins into an integer
  int values = 0;
  int bit = 0;
  for(int i=2; i <= 13; i++)
  {
    bitWrite(values, bit, digitalRead(i));  // set the bit to 0 or 1 depending
                                            // on value of the given pin
    bit = bit + 1;                          // increment to the next bit
  }
  sendBinary(values); // send the integer

  for(int i=0; i < 6; i++)
  {
    values = analogRead(i);
    sendBinary(values); // send the integer
  }
  delay(1000); //send every second
}

// function to send the given integer value to the serial port
void sendBinary( int value)
{
  // send the two bytes that comprise an integer
  Serial.print(lowByte(value), BYTE);  // send the low byte
  Serial.print(highByte(value), BYTE); // send the high byte
}
```

Discussion

The code sends a header (the character `H`), followed by an integer holding the digital pin values using the `bitRead` function to set a single bit in the integer to correspond to the value of the pin (see Chapter 3). It then sends six integers containing the values read from the six analog ports (see Chapter 5 for more information). All the integer values are sent using `sendBinary`, introduced in Recipe 4.6. The message is 15 bytes long—1 byte for the header, 2 bytes for the digital pin values, and 12 bytes for the six analog integers. The code for the digital and analog inputs is explained in Chapter 5.

Assuming analog pins have values of 0 on pin 0, 100 on pin 1, and 200 on pin 2 through 500 on pin 5, and digital pins 2 through 7 are high and 8 through 13 are low, this is the decimal value of each byte that gets sent:

```
72   // the character 'H' - this is the header
     // two bytes in low high order containing bits representing pins 2-13
63   // binary 00111111 :  this indicates that pins 2-7 are high
```

```
0    // this indicates that  8-13 are low

     // two bytes for each pin representing the analog value
0    // pin 0 has an integer value of 0 so this is sent as two bytes
0

100  // pin 1 has a value of 100, sent as a byte of 100 and a byte of 0
0
...
     // pin 5 has a value of 500
244  // the remainder when dividing 500 by 256
1    //  the number of times 500 can be divided by 256
```

This Processing code reads this message and prints the values to the Processing console:

```
/*
 * ReceiveMultipleFieldsBinary_P
 *
 * portIndex must be set to the port connected to the Arduino
*/

import processing.serial.*;

Serial myPort;        // Create object from Serial class
short portIndex = 0;  // select the com port, 0 is the first port

char HEADER = 'H';

void setup()
{
  size(200, 200);
  // Open whatever serial port is connected to Arduino.
  String portName = Serial.list()[portIndex];
  println(Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this, portName, 9600);
}

void draw()
{
int val;

  if ( myPort.available() >= 15)  // wait for the entire message to arrive
  {
    if( myPort.read() == HEADER) // is this the header
    {
      println("Message received:");
      // header found
      // get the integer containing the bit values
      val = readArduinoInt();
      // print the value of each bit
      for(int pin=2, bit=1; pin <= 13; pin++){
        print("digital pin " + pin + " = " );
        int isSet = (val & bit);
        if( isSet == 0)
          println("0");
```

```
        else
          println("1");
          bit = bit * 2; // shift the bit
      }
      println();
      // print the six analog values
      for(int i=0; i < 6; i ++){
        val = readArduinoInt();
        println("analog port " + i + "= " + val);
      }
      println("----");
    }
  }
}

// return the integer value from bytes received on the serial port (in low,high
order)
int readArduinoInt()
{
  int val;      // Data received from the serial port

  val = myPort.read();              // read the least significant byte
  val =  myPort.read() * 256 + val; // add the most significant byte
  return val;
}
```

The Processing code waits for 15 characters to arrive. If the first character is the header, it then calls the function named `readArduinoInt` to read two bytes and transform them back into an integer by doing the complementary mathematical operation that was performed by Arduino to get the individual bits representing the digital pins. The six integers are then representing the analog values.

See Also

To send Arduino values back to the computer or drive the pins from the computer (without making decisions on the board), consider using Firmata (*http://www.firmata.org*). The Firmata library is included in the Arduino software, and a library is available to use in Processing. You load the Firmata code onto Arduino, control whether pins are inputs or outputs from the computer, and then set or read those pins.

4.10 How to Move the Mouse Cursor on a PC or Mac

Problem

You want Arduino to interact with an application on your computer by moving the mouse cursor. Perhaps you want to move the mouse position in response to Arduino information. For example, suppose you have connected a Wii nunchuck (see Recipe 13.2) to your Arduino and you want your hand movements to control the position of the mouse cursor in a program running in a PC.

Solution

You can send serial commands that contain the mouse cursor position to a program running on the target computer. Here is a sketch that moves the mouse cursor based on the position of two potentiometers:

```
// SerialMouse sketch
#define potXPin 4
#define potYPin 5

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  int x = analogRead(potXPin);
  int y = analogRead(potYPin);
  Serial.print(x,DEC);
  Serial.print(",");
  Serial.print(y,DEC);
  Serial.println();  // send a cr/lf
  delay(50); // send position 20 times a second
}
```

Figure 4-3 illustrates the wiring for two potentiometers (see Chapter 5 for more details).

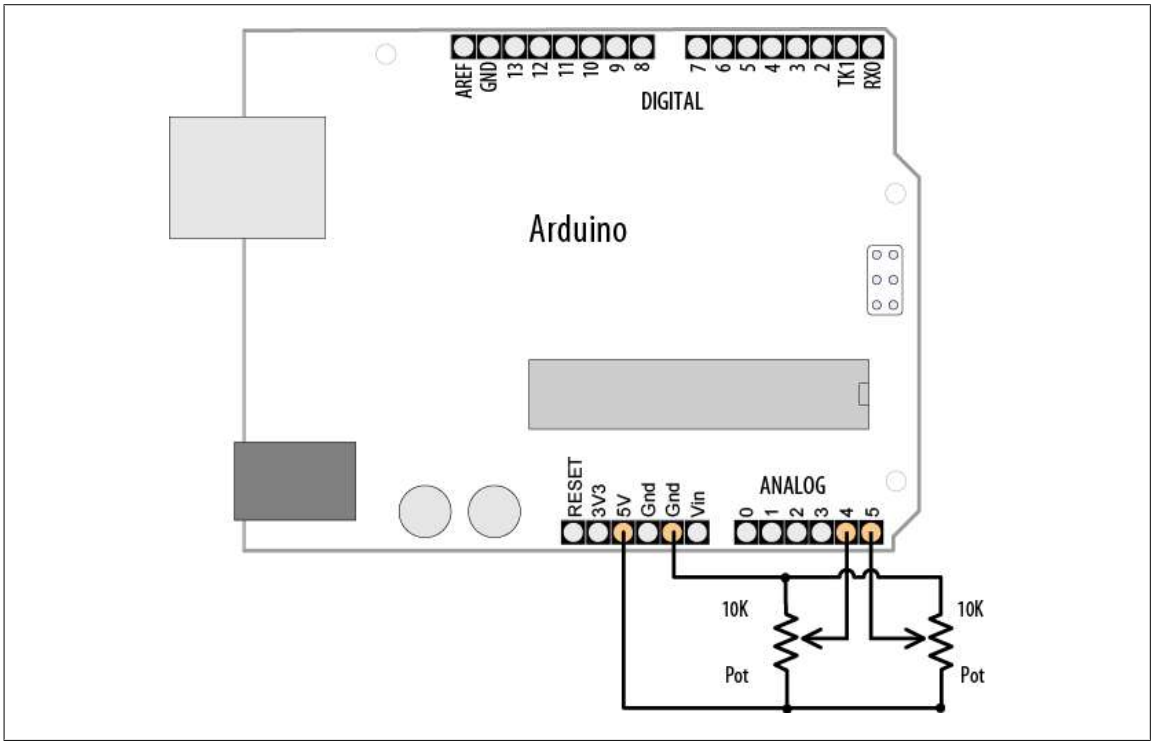

Figure 4-3. Wiring for two potentiometers

The Processing code is based on the code shown in Recipe 4.4:

```
/*
 * ArduinoMouse.pde  (Processing sketch)
 */

/* WARNING: This sketch takes over your mouse
 Press escape to close running sketch */

import processing.serial.*;

Serial myPort;     // Create object from Serial class
Robot myRobot;     // create object from Robot class;

public static final char HEADER = 'M';    // character to identify the start of
a message
public static final short LF = 10;        // ASCII linefeed
public static final short portIndex = 1;  // select the com port, 0 is the first
port


void setup() {
  size(200, 200);
  println(Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this,Serial.list()[portIndex], 9600);
  try {
    myRobot = new Robot(); // the Robot class gives access to the mouse
  }
  catch (AWTException e) {  // this is the Java exception handler
    e.printStackTrace();
  }
}

void draw() {

}

void serialEvent(Serial p) {
  String message = myPort.readStringUntil(LF); // read serial data
  if(message != null)
  {
    print(message);
    String [] data  = message.split(","); // Split the comma-separated message
    if(data[0].charAt(0) == HEADER) // check for header character in the first
field
    {
      if( data.length > 3)
      {
        int x = Integer.parseInt(data[1]);
        int y = Integer.parseInt(data[2]);
        print("x= " + x);
        println(", y= " + y);
        myRobot.mouseMove(x,y); // move mouse to received x and y position
      }
    }
```

```
  }
}
```

The Processing code splits the message containing the *x* and *y* coordinates and sends them to the `mouseMove` method of the Java `Robot` class.

Discussion

This technique for controlling applications running on your computer is easy to implement and should work with any operating system that can run the Processing application.

Some platforms require special privileges or extensions to access low-level input control. If you can't get control of the mouse, check the documentation for your operating system.

If you require Arduino to actually appear as though it were a mouse to the computer, you have to emulate the actual USB protocol real mice use. This protocol, for Human Interface Devices (HID), is complex, but Phillip Lindsay has some useful information and code at *http://code.google.com/p/vusb-for-arduino/*.

A runaway `Robot` object has the ability to remove your control over the mouse and keyboard if used in an endless loop.

See Also

Go to *http://java.sun.com/j2se/1.3/docs/api/java/awt/Robot.html* for more information on the Java `Robot` class.

An article on using the `Robot` class is available at *http://www.developer.com/java/other/article.php/10936_2212401_1*.

If you prefer to use a Windows programming language, the low-level Windows API function to insert keyboard and mouse events into the input stream is called `SendInput`. You can visit *http://msdn.microsoft.com/en-us/library/ms646310(VS.85).aspx* for more information.

4.11 Controlling Google Earth Using Arduino

Problem

You want to control movement in an application such as Google Earth using sensors attached to Arduino. For example, you want sensors to detect hand movements to act

as the control stick for the flight simulator in Google Earth. The sensors could use a joystick (see Recipe 6.17) or a Wii nunchuck (see Recipe 13.2).

Solution

Google Earth lets you "fly" anywhere on Earth to view satellite imagery, maps, terrain, and 3D buildings (see Figure 4-4). It contains a flight simulator that can be controlled by a mouse, and this recipe uses techniques described in Recipe 4.10 combined with a sensor connected to Arduino to provide the joystick input.

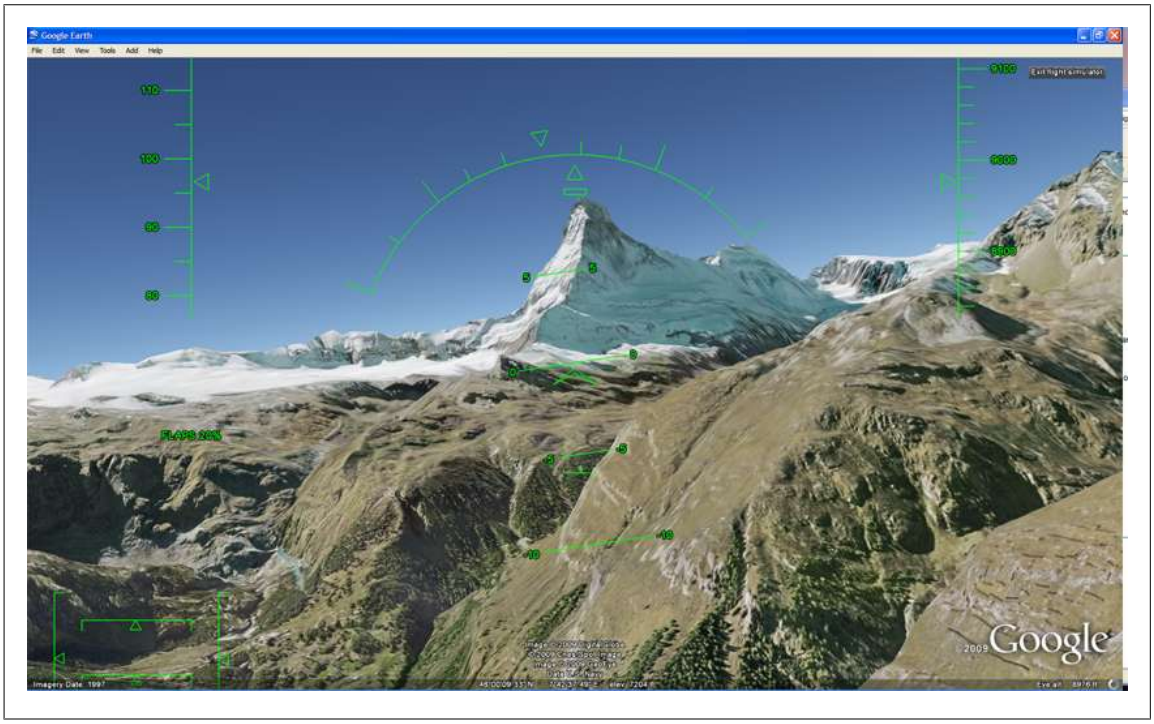

Figure 4-4. Google Earth flight simulator

This recipe's Solution is based on the method used in Recipe 4.10 for emulating a mouse by sending Arduino data to Processing. The Arduino code sends the horizontal and vertical positions determined by reading the joystick values (the joystick code is discussed in Recipe 6.17) from a PlayStation game controller:

```
*
 * GoogleEarthPSX
 *
 * Send joystick data from PSX to Processing
 * uses PSX library discussed in Recipe 6.17
 */


#include <Psx.h>                        // Includes the Psx Library

Psx Psx;                                // Create an instance of the Psx library
const int dataPin = 5;
```

```
const int cmndPin = 4;
const int attPin = 3;
const int clockPin = 2;
const int psxDelay = 10; // this determines the clock delay in microseconds

const byte nudge = 64;   // the amount of movement to be sent when stick is pushed
const byte HEADER = 255; // this value is sent as the header
unsigned int data;
byte x,y, buttons;

void setup()
{
  Serial.begin(9600);
  Psx.setupPins(dataPin, cmndPin, attPin, clockPin, psxDelay);  // initialize Psx
}


void loop()
{

  data = Psx.read();   // get the psx controller button data
  x = y = 127;         // center x & y values, offsets are added if buttons pressed
  buttons = 0;

 if(data & psxLeft || data & psxSqu)
    x = x - nudge;
 if(data & psxDown ||data & psxX)
    y = y + nudge;
 if(data & psxRight ||data & psxO)
    x = x + nudge;
 if(data & psxUp ||data & psxTri)
    y = y - nudge;
  if(data & psxStrt)  // (Z button)
     buttons = buttons + 2;
  if(data & psxSlct)  // (C button)
     buttons = buttons + 1;


 Serial.print(HEADER);
 Serial.print(x);
 Serial.print(y);
 Serial.print(buttons);

 delay(20); // send position 50 times a second
}
```

The Processing sketch reads the header byte and three data bytes for *x* and *y* mouse position and button state. The technique for handling binary data is discussed in Recipe 4.7:

```
/**
 * GoogleEarthFS_P
 *
 * Read Arduino Serial packets
 * and send mouse position to Google Earth
```

```
 *
 */


import processing.serial.*;

Serial myPort;       // Create object from Serial class
int portIndex = 1;   // select the com port, 0 is the first port
int HEADER = 255;
int val;             // Data received from the serial port

GoogleFS myGoogle;

void setup()
{
  size(256, 256);
  println(Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this,Serial.list()[portIndex], 9600);
  myGoogle = new GoogleFS();
  smooth();
  fill(255);
  background(255);
  println("Start Google FS in the center of your screen");
  println("center the mouse pointer in google earth
and press Z button to start");
  do{
    getData();
  }
  while(buttons != 2 ); // wait for data and Z button to start

  println("started");
  myGoogle.mousePress(InputEvent.BUTTON1_MASK); // starts the FS
}

int accx,accy,buttons;
int xOffset, yOffset = 0;

boolean getData(){
  if ( myPort.available() >= 4) {  // If a data packet is available,
    if(myPort.read() == HEADER){  // check for header
      // erase the old markers
      stroke(255);
      ellipse(accx, accy,4,4);

      accx = myPort.read();
      accy = myPort.read();
      buttons = myPort.read();
      if(xOffset == 0){
        // here if first time
        xOffset = accx;
        yOffset = accy;
      }
      if( buttons == 1) {
        println("tbutton: c");
```

```
        xOffset = accx;
        yOffset = accy;
      }
      if(buttons == 3 ){
        exit();
      }
      return true;  // data available
    }
  }
  return false;
}

void draw()
{
  if(getData()){
    if(buttons != 2 ){
      // only activate the mouse when the Z button is not pressed
      myGoogle.move(accx- xOffset, accy - yOffset);
     }

    print("accx: ");     print(accx);
    print("\taccy: ");   print(accy);
    println();

    stroke(255,0,0);     ellipse(accx, accy,4,4);
  }
}

class GoogleFS {
  Robot myRobot;     // create object from Robot class;
  int centerX,centerY;
  GoogleFS(){
    try {
      myRobot = new Robot();
    }
    catch (AWTException e) {
      e.printStackTrace();
    }
    Dimension screen = java.awt.Toolkit.getDefaultToolkit().getScreenSize();
    centerY =  (int)screen.getHeight() / 2 ;
    centerX =  (int)screen.getWidth() / 2;
  }
  // moves mouse from center of screen by given offset
  void move(int offsetX, int offsetY){
    myRobot.mouseMove(centerX + 4 * offsetX,centerY + -4 * offsetY);
  }
  void mousePress( int button){
    myRobot.mousePress(button) ;
  }
}
```

Discussion

Arduino determines the horizontal and vertical positions by reading the joystick value from the PSX (PlayStation game controller). This control does not actually provide a

value proportional to the stick position, so the Arduino code sends an offset from the center when the stick is moved—this is the `nudge` variable and it determines the value of this offset, and therefore the sensitivity of the control. The state of the Select and Start switches is also sent (when pressed, Select has a value of 2 and Start has a value of 1; when both are pressed the value is 3, and when no button is pressed the value is 0).

Follow the instructions for connecting the PSX in Recipe 6.17.

Google Earth is a free download; you can get it from the Google website, *http://earth.google.com/download-earth.html*. Download and run the version for your operating system to install it on your computer. Start Google Earth, and from the Tools menu, select Enter Flight Simulator. Select an aircraft (the SR22 is easier to fly than the F16) and an airport. The Joystick support should be left unchecked—you will be using the mouse to control the aircraft. Click the Start Flight button and immediately press the space bar to pause the simulator so that you can get the Processing sketch running.

Run the *GoogleEarthFS_P* sketch and press the Start button on the PSX controller. You will see a dot in the Processing draw window showing the joystick position, and you should see this move as you press the PSX controller joystick buttons. At this point, the mouse position is under the control of Arduino, but control returns to your computer mouse when you hold the PSX Start button. Make Google Earth the Active window by holding the PSX Start button and clicking on Google Earth.

You are now ready to fly. Release the PSX Start button, press Page Up on your keyboard a few times to increase the throttle, and then press the space bar on your keyboard to unpause the simulator. When the SR22 reaches an air speed that is a little over 100 knots, you can "pull back" on the stick and fly. Information explaining the simulator controls is available in the Help menu.

Here is another variation that sends a similar message to the Processing sketch. This one uses the Wii nunchuck code from Recipe 13.2:

```
/*
 * WiichuckSerial
 *
 * based on code from Tod E. Kurt, http://thingm.com/
 * Modified to send serial packets to Processing
 *
 */


#include <Wire.h>
#include "nunchuck_funcs.h"

int loop_cnt=0;
const byte header = 254;              // a value to indicate start of message

byte accx,accy,joyx,joyy,buttons;

// set the current coordinates as the center points
```

```
void setup()
{
    Serial.begin(9600);
    nunchuck_setpowerpins();
    nunchuck_init(); // send the initialization handshake
    nunchuck_get_data(); // ignore the first time
    delay(50);
}

void loop()
{
    if( loop_cnt > 50 ) { // every 50 msecs get new data
        loop_cnt = 0;

        nunchuck_get_data();

        accx  = nunchuck_accelx();
        accy  = nunchuck_accely();
        buttons = nunchuck_zbutton() * 2;
        buttons = buttons + nunchuck_cbutton(); // cbutton is least significant bit

        Serial.print((byte)HEADER);          // value indicating start of message
        Serial.print((byte)accx);
        Serial.print((byte)accy);
        Serial.print((byte)buttons);

    }
    loop_cnt++;
    delay(1);
}
```

The connections are shown in Recipe 13.2. Operation is similar to the PSX version, except you use the Z button on the nunchuck instead of the Start button.

See Also

The Google Earth website contains the downloadable code and instructions needed to get this going on your computer: *http://earth.google.com/*.

4.12 Logging Arduino Data to a File on Your Computer

Problem

You want to create a file containing information received over the serial port from Arduino. For example, you want to save the values of the digital and analog pins at regular intervals to a logfile.

Solution

We covered sending information from Arduino to your computer in previous recipes. This solution uses the same Arduino code explained in Recipe 4.9. The Processing

sketch that handles file logging is based on the Processing sketch also described in that recipe.

This Processing sketch creates a file (using the current date and time as the filename) in a directory called *Arduino*. Messages received from Arduino are added to the file. Pressing any key saves the file and exits the program:

```
/*
 * ReceiveMultipleFieldsBinaryToFile_P
 *
 * portIndex must be set to the port connected to the Arduino
 * based on ReceiveMultipleFieldsBinary, this version saves data to file
 * Press any key to stop logging and save file
 */

import processing.serial.*;


PrintWriter output;
DateFormat fnameFormat= new SimpleDateFormat("yyMMdd_HHmm");
DateFormat  timeFormat = new SimpleDateFormat("hh:mm:ss");
String fileName;

Serial myPort;        // Create object from Serial class
short portIndex = 0;  // select the com port, 0 is the first port
char HEADER = 'H';

void setup()
{
  size(200, 200);
  // Open whatever serial port is connected to Arduino.
  String portName = Serial.list()[portIndex];
  println(Serial.list());
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this, portName, 9600);
  Date now = new Date();
  fileName = fnameFormat.format(now);
  output = createWriter(fileName + ".txt"); // save the file in the sketch folder
}

void draw()
{
  int val;
  String time;

  if ( myPort.available() >= 15)  // wait for the entire message to arrive
  {
    if( myPort.read() == HEADER) // is this the header
    {
      String timeString = timeFormat.format(new Date());
      println("Message received at " + timeString);
      output.println(timeString);
      // header found
      // get the integer containing the bit values
      val = readArduinoInt();
```

```
      // print the value of each bit
      for(int pin=2, bit=1; pin <= 13; pin++){
        print("digital pin " + pin + " = " );
        output.print("digital pin " + pin + " = " );
        int isSet = (val & bit);
        if( isSet == 0){
           println("0");
           output.println("0");
        }
        else  {
          println("1");
          output.println("0");
        }
        bit = bit * 2; // shift the bit
      }
      // print the six analog values
      for(int i=0; i < 6; i ++){
        val = readArduinoInt();
        println("analog port " + i + "= " + val);
        output.println("analog port " + i + "= " + val);
      }
      println("----");
      output.println("----");
    }
  }
}

void keyPressed() {
  output.flush(); // Writes the remaining data to the file
  output.close(); // Finishes the file
  exit(); // Stops the program
}

// return the integer value from bytes received on the serial port (in low,high
order)
int readArduinoInt()
{
  int val;      // Data received from the serial port

  val = myPort.read();          // read the least significant byte
  val =  myPort.read() * 256 + val; // add the most significant byte
  return val;
}
```

Don't forget that you need to set `portIndex` to the serial port connected to Arduino.

Discussion

The base name for the logfile is formed using the `DateFormat` function in Processing:

```
DateFormat fnameFormat= new SimpleDateFormat("yyMMdd_HHmm");
```

The full filename is created with code that adds a directory and file extension:

```
  output = createWriter(fileName + ".txt");
```

The file will be created in the same directory as the Processing sketch (the sketch needs to be saved at least once to ensure that the directory exists). `createWriter` is the Processing function that opens the file; this creates an object (a unit of runtime functionality) called `output` that handles the actual file output. The text written to the file is the same as what is printed to the console in Recipe 4.9, but you can format the file contents as required by using the standard string handling capabilities of Processing. For example, the following variation on the draw routine produces a comma-separated file that can be read by a spreadsheet or database. The rest of the Processing sketch can be the same, although you may want to change the extension from *.txt* to *.csv*:

```
void draw()
{
  int val;
  String time;

  if ( myPort.available() >= 15)  // wait for the entire message to arrive
  {
    if( myPort.read() == HEADER) // is this the header
    {
       String timeString = timeFormat.format(new Date());
       output.print(timeString);
       val = readArduinoInt(); // read but don't output the digital values

      // output the six analog values delimited by a comma
      for(int i=0; i < 6; i ++){
        val = readArduinoInt();
        output.print("," + val);
      }
      output.println();
    }
  }
}
```

See Also

For more on `createWriter`, see *http://processing.org/reference/createWriter_.html*.

4.13 Sending Data to Two Serial Devices at the Same Time

Problem

You want to send data to a serial device such as a serial LCD, but you are already using the built-in serial port to communicate with your computer.

Solution

On a Mega this is not a problem, as it has four hardware serial ports; just create two serial objects and use one for the LCD and one for the computer:

```
void setup() {
  // initialize two serial ports on a megaL
  Serial.begin(9600);
  Serial1.begin(9600);
}
```

On a standard Arduino board (such as the Uno or Duemilanove) that only has one hardware serial port, you will need to create an emulated or "soft" serial port.

You can use Mikal Hart's NewSoftSerial, a serial port emulation library, available at *http://arduiniana.org/libraries/newsoftserial*. Download and install NewSoftSerial.

The Arduino team is planning to provide NewSoftSerial with future Arduino downloads. Check the release notes for your Arduino version to see if this software is already included.

Select two available digital pins, one each for transmit and receive, and connect your serial device to them. It is convenient to use the hardware serial port for communication with the computer because this has a USB adapter on the board. Connect the device's transmit line to the receive pin and the receive line to the transmit pin. In Figure 4-5, we have selected pin 2 as the receive pin and pin 3 as the transmit pin.

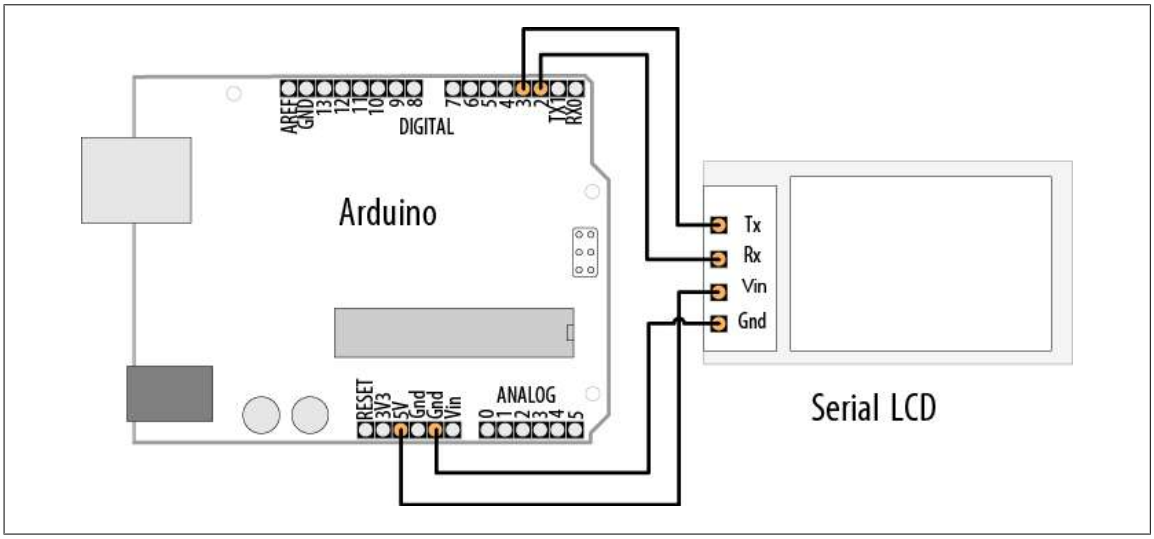

Figure 4-5. Connecting a serial device to a "soft" serial port

In your sketch, create a `NewSoftSerial` object and tell it which pins you chose as your emulated serial port. In this example, we're creating an object named `serial_lcd`, which we instruct to use pins 2 and 3:

```
/*
 * NewSoftSerialOutput sketch
 * Output data to a software serial port
*/
```

```
#include <NewSoftSerial.h>
...
const int rxpin = 2;           // pin used to receive from LCD
const int txpin = 3;           // pin used to send to LCD
NewSoftSerial serial_lcd(txpin, rxpin); // new serial port on pins 2 and 3

void setup()
{
  Serial.begin(9600); // 9600 baud for the built-in serial port
  serial_lcd.begin(9600); //initialize the software serial port also for 9600
}

int number = 0;

void loop()
{
  serial_lcd.print("The number is ");  // send text to the LCD
  serial_lcd.println(number);    // print the number on the LCD
  Serial.print("The number is ");
  Serial.println(number);        // print the number on the PC console

  delay(500); // delay half second between numbers
  number++;   // to the next number
}
```

This sketch assumes that a serial LCD has been connected to pins 2 and 3 as shown in Figure 4-5, and that a serial console is connected to the built-in port. The loop will repeatedly display the same message on each:

```
The number is 0
The number is 1
...
```

Discussion

Every Arduino microcontroller contains at least one built-in serial port. This special piece of hardware is responsible for generating the series of precisely timed pulses its partner device sees as data and for interpreting the similar stream that it receives in return. Although the Mega has four such ports, most Arduino flavors have only one. For projects that require connections to two or more serial devices, you'll need a software library that emulates the additional ports. A "software serial" library effectively turns an arbitrary pair of digital I/O pins into a new serial port.

Although one such library, SoftwareSerial, is included in every Arduino distribution, most programmers prefer to use the more powerful and feature-laden NewSoftSerial library. NewSoftSerial supports a wider range of baud rates and uses some advanced features of the Arduino processor to ensure more reliable data reception. It also supports multiple simultaneous emulated ports. You can read more about NewSoftSerial on Mikal Hart's website.

Table 4-3 compares the features of the NewSoftSerial and SoftwareSerial libraries.

Table 4-3. Feature comparison of two emulation libraries

Feature	NewSoftSerial	SoftwareSerial
Distributed with Arduino software	No[a]	Yes
Max transmit baud rate	115.2K	To about 9,600
Max receive baud rate	38.4K	To about 9,600
Supports 8 MHz processors	Yes	No
Reliable interrupt-driven receives	Yes	No
Supports multiple emulated ports	Yes	No
`.available()` and `.overflow()` methods	Yes	N/A

[a] Note that as of Arduino release 22, NewSoftSerial was available only as a third-party library, but future releases may include it with the base distribution.

To build your software serial port, you select a pair of pins that will act as the port's transmit and receive lines in much the same way that pins 1 and 0 are controlled by Arduino's built-in port. In Figure 4-5, pins 3 and 2 are shown, but any available digital pins can be used. It's wise to avoid using 0 and 1, because these are already being driven by the built-in port.

The syntax for writing to the soft port is identical to that for the hardware port. In the example sketch, data is sent to both the "real" and emulated ports using `print()` and `println()`:

```
serial_lcd.print("The number is ");  // send text to the LCD
serial_lcd.println(number);          // send the number on the LCD

Serial.print("The number is ");      // send text to the hardware port
Serial.println(number);              // to output on Arduino Serial Monitor
```

If you are using a *unidirectional* serial device—that is, one that only sends or receives—you can conserve resources by specifying a nonexistent pin number in the `NewSoftSerial` constructor for the line you don't need. For example, a serial LCD is fundamentally an output-only device. If you don't expect (or want) to receive data from it, you can tell NewSoftSerial using this syntax:

```
#include <NewSoftSerial.h>
...
const int no_such_pin = 255;
const int txpin = 3;
NewSoftSerial serial_lcd(txpin, no_such_pin); // TX-only on pin 3
```

In this case, we would only physically connect a single pin (3) to the serial LCD's "input" or "RX" line.

4.14 Receiving Serial Data from Two Devices at the Same Time

Problem

You want to receive data from a serial device such as a serial GPS, but you are already using the built-in serial port to communicate with your computer.

Solution

This problem is similar to the preceding one, and indeed the solution is much the same. If your Arduino's serial port is connected to the console and you want to attach a second serial device, you must create an emulated port using a software serial library such as NewSoftSerial. In this case, we will be receiving data from the emulated port instead of writing to it, but the basic solution is very similar.

Download NewSoftSerial from Mikal Hart's website. Select two pins to use as your transmit and receive lines.

Connect your GPS as shown in Figure 4-6. Rx (receive) is not used in this example, so you can ignore the Rx connection to pin 3 if your GPS does not have a receive pin.

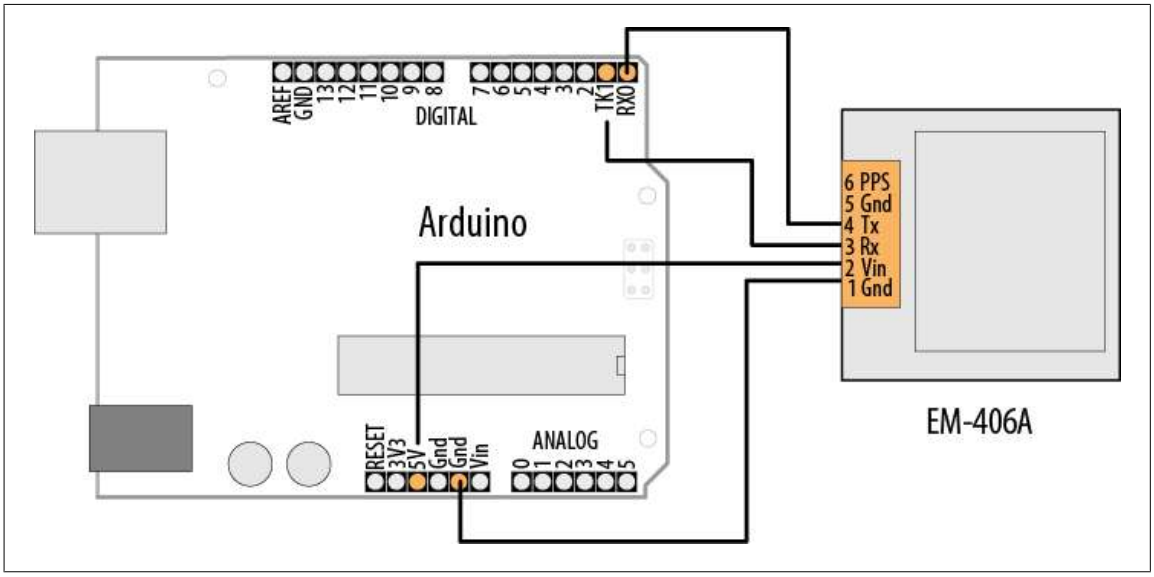

Figure 4-6. Connecting a serial GPS device to a "soft" serial port

As you did in Recipe 4.13, create a `NewSoftSerial` object in your sketch and tell it which pins to control. In the following example, we define a soft serial port called `serial_gps`, using pins 2 and 3 for receive and transmit, respectively:

```
/*
 * NewSoftSerialInput sketch
 * Read data from a software serial port
*/
```

```
#include <NewSoftSerial.h>
const int rxpin = 2;                    // pin used to receive from GPS
const int txpin = 3;                    // pin used to send to GPS
NewSoftSerial serial_gps(txpin, rxpin); // new serial port on pins 2 and 3

void setup()
{
  Serial.begin(9600); // 9600 baud for the built-in serial port
  serial_gps.begin(4800); // initialize the port, most GPS devices use 4800 baud
}

void loop()
{
  if (serial_gps.available() > 0) // any character arrived yet?
  {
    char c = serial_gps.read();   // if so, read it from the GPS
    Serial.print(c, BYTE);        // and echo it to the serial console
  }
}
```

This short sketch simply forwards all incoming data from the GPS to the Arduino Serial Monitor. If the GPS is functioning and your wiring is correct, you should see GPS data displayed on the Serial Monitor.

Discussion

You initialize an emulated NewSoftSerial port by providing pin numbers for transmit and receive. The following code will set up the port to send on pin 2 and receive on pin 3:

```
const int rxpin = 2;                    // pin used to receive from GPS
const int txpin = 3;                    // pin used to send to GPS
NewSoftSerial serial_gps(txpin, rxpin); // new serial port on pins 2 and 3
```

The syntax for reading an emulated port is very similar to that for reading from a built-in port. First check to make sure a character has arrived from the GPS with `available()`, and then read it with `read()`.

It's important to remember that software serial ports consume time and resources. An emulated serial port must do everything that a hardware port does, using the same processor your sketch is trying to do "real work" with. Whenever a new character arrives, the processor must interrupt whatever it was doing to handle it. This can be time-consuming. At 4,800 baud, for example, it takes the Arduino about two milliseconds to process a single character. While two milliseconds may not sound like much, consider that if your peer device—say, the GPS unit shown earlier—transmits 200 to 250 characters per second, your sketch is spending 40 to 50 percent of its time trying to keep up with the serial input. This leaves very little time to actually *process* all that data. The lesson is that if you have two serial devices, when possible connect the one with the higher bandwidth consumption to the built-in (hardware) port. If you must connect a high-bandwidth device to a software serial port, make sure the rest of your sketch's loop is very efficient.

Receiving data from multiple NewSoftSerial ports

With NewSoftSerial (but not SoftwareSerial), it is possible to create multiple "soft" serial ports in the same sketch. This is a useful way to control, say, several XBee radios in the same project. The caveat is that at any given time, only one of these ports can actively receive data. Reliable reception on a software port requires the processor's undivided attention. That's why NewSoftSerial can only activate one port for data reception at a given time. (This restriction does not apply to *sending* data, only receiving it. See the NewSoftSerial documentation for specifics.)

It *is* possible to receive on two different NewSoftSerial ports in the same sketch. You just have to take some care that you aren't trying to receive from both at the same time. There are many successful designs which, say, monitor a serial GPS device for a while, then later in the sketch accept input from an XBee. The key is to alternate between them. NewSoftSerial considers the "active" port to be whichever port you have most recently accessed using the `read`, `print`, `println`, or `available` method. The following code fragment illustrates how you design a sketch to read first from one port and then from another:

```
/*
 * MultiRX sketch
 * Receive data from two software serial ports
*/

#include <NewSoftSerial.h>
const int rxpin1 = 2;
const int txpin1 = 3;
const int rxpin2 = 4;
const int txpin2 = 5;
NewSoftSerial gps(txpin1, rxpin1);  // gps device connected to pins 2 and 3
NewSoftSerial xbee(txpin2, rxpin2); // gps device connected to pins 2 and 3

void setup()
{
  gps.begin(4800);
  xbee.begin(9600);
}

void loop()
{
  if (xbee.available() > 0) // xbee is active.  Any characters available?
  {
    if (xbee.read() == 'y') // if xbee received a 'y' character
    {
      unsigned long start = millis(); // begin listening to the GPS
      while (start + 100000 > millis())
      // listen for 10 seconds
```

```
        {
          if (gps.available() > 0) // now gps device is active
          {
            char c = gps.read();
            // *** process gps data here
          }
        }
      }
    }
}
```

This sketch is designed to treat the XBee radio as the active port until it receives a `y` character, at which point the GPS becomes active. After processing GPS data for 10 seconds, the sketch once again returns to listening to the XBee port. Data that arrives on an inactive port is simply discarded.

Note that the "active port" restriction only applies to multiple *soft* ports. If your design really must receive data from more than one serial device simultaneously, consider attaching one of these to the built-in hardware port. Alternatively, it is perfectly possible to add additional hardware ports to your projects using external chips, devices called *UARTs*.

4.15 Setting Up Processing on Your Computer to Send and Receive Serial Data

Problem

You want to use the Processing development environment to send and receive serial data.

Solution

You can get the Processing application from the Downloads section of the Processing website, *http://processing.org*. Files are available for each major operating system. Download the appropriate one for your operating system and unzip the file to somewhere that you normally store applications. On a Windows computer, this might be a location like *C:\Program Files\Processing*. On a Mac, it might be something like */Applications/Processing/*.

If you installed Processing on the same computer that is running the Arduino IDE, the only other thing you need to do is identify the serial port in Processing. The following Processing sketch prints the serial ports available:

```
/**
 * GettingStarted
 *
 * A sketch to list the available serial ports
 * and display characters received
 */
```

```
import processing.serial.*;

Serial myPort;      // Create object from Serial class
int portIndex = 0;  // set this to the port connected to Arduino
int val;            // Data received from the serial port

void setup()
{
  size(200, 200);
  println(Serial.list()); // print the list of all the ports
  println(" Connecting to -> " + Serial.list()[portIndex]);
  myPort = new Serial(this, Serial.list()[portIndex], 9600);
}

void draw()
{
  if ( myPort.available() > 0) // If data is available,
  {
    val = myPort.read();         // read it and store it in val
    print(val);
  }
}
```

If you are running Processing on a computer that is not running the Arduino development environment, you need to install the Arduino USB drivers (Chapter 1 describes how to do this).

Set the variable `portIndex` to match the port used by Arduino. You can see the port numbers printed in the Processing text window (the area below the source code, not the separate Display window; see *http://processing.org/reference/environment*). Recipe 1.4 describes how to find out which serial port your Arduino board is using.