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

floatvalue: 66.00
```

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

Data type	Print (val)	Print (val,DEC)	Print (val,BYTE)	Print (val,HEX)	Print (val,OCT)	Print (val,BIN)
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;
```

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.

NOTE

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.

WARNING

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.

NOTE

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.

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. 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).

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 4.7shows 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

Check the Arduino references
for `lowByte` at http://www.arduino.cc/en/Reference/LowByte and `highByte` a t 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.

4.9. 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 (seeChapter 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.