Embedded Systems
Making the Sketch Do Your Bidding

Structuring an Arduino Program:

- Programs for Arduino are usually referred to as ***sketches***. The terms sketch and program are interchangeable.
- Sketches contain code—the instructions the board will carry out.
- Code that needs to run only once (such as to set up the board for your application) must be placed in the setup function.
- Code to be run continuously after the initial setup has finished goes into the loop function.

Terminology:

"*sketch*" – a program you write to run on an Arduino board

"*pin*" – an input or output connected to something.

e.g. output to an LED, input from a knob.

"*digital*" – value is either HIGH or LOW.

(aka on/off, one/zero) e.g. switch state

"*analog*" – value ranges, usually from 0-255.

e.g. LED brightness, motor speed, etc.

Structuring an Arduino Program:

- Here is a typical sketch:

```
const int ledPin =  13;     // LED connected to digital pin 13

// The setup() method runs once, when the sketch starts
void setup()
{
  pinMode(ledPin, OUTPUT);      // initialize the digital pin as an output
}

// the loop() method runs over and over again,
void loop()
{
  digitalWrite(ledPin, HIGH);   // turn the LED on
  delay(1000);                  // wait a second
  digitalWrite(ledPin, LOW);    // turn the LED off
  delay(1000);                  // wait a second
}
```

Using Simple Primitive Types (Variables):

Table 2-1. Arduino data types

Numeric types	Bytes	Range	Use
`int`	2	–32768 to 32767	Represents positive and negative integer values.
`unsigned int`	2	0 to 65535	Represents only positive values; otherwise, similar to `int`.
`long`	4	–2147483648 to 2147483647	Represents a very large range of positive and negative values.
`unsigned long`	4	4294967295	Represents a very large range of positive values.
`float`	4	3.4028235E+38 to –3.4028235E+38	Represents numbers with fractions; use to approximate real-world measurements.
`double`	4	Same as `float`	In Arduino, `double` is just another name for `float`.
`boolean`	1	`false` (0) or `true` (1)	Represents true and false values.
`char`	1	–128 to 127	Represents a single character. Can also represent a signed value between –128 and 127.
`byte`	1	0 to 255	Similar to `char`, but for unsigned values.

Other types	Use
`String`	Represents arrays of `chars` (characters) typically used to contain text.
`void`	Used only in function declarations where no value is returned.

Using Floating-Point Numbers:

- **Problem:**
  - Floating-point numbers are used for values expressed with decimal points (this is the way to represent fractional values). You want to calculate and compare these values in your sketch.

- **Solution:**
  - The following code shows how to declare floating-point variables, illustrates problems you can encounter when comparing floating-point values, and demonstrates how to overcome them:

Using Floating-Point Numbers:

```
/*
 * Floating-point example
 * This sketch initialized a float value to 1.1
 * It repeatedly reduces the value by 0.1 until the value is 0
 */


float  value = 1.1;

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

void loop()
{
  value = value - 0.1;   // reduce value by 0.1 each time through the loop
  if( value == 0)
     Serial.println("The value is exactly zero");
  else if(almostEqual(value, 0))
  {
    Serial.print("The value ");
    Serial.print(value,7); // print to 7 decimal places
    Serial.println(" is almost equal to zero");
  }
  else
    Serial.println(value);

  delay(100);
```

```
}

// returns true if the difference between a and b is small
// set value of DELTA to the maximum difference considered to be equal
boolean almostEqual(float a, float b)
{
  const float DELTA = .00001; // max difference to be almost equal
  if (a == 0) return fabs(b) <= DELTA;
  if (b == 0) return fabs(a) <= DELTA;
  return fabs((a - b) / max(fabs(a), fabs(b))) <= DELTA ;
}
```

Using Floating-Point Numbers:

The Serial Monitor output from this sketch is as follows:

```
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
The value -0.0000001 is almost equal to zero
-0.10
-0.20
```

The output continues to produce negative numbers.

Working with Groups of Values:

- **Problem:**
  - You want to create and use a group of values (called arrays). The arrays may be a simple list or they could have two or more dimensions. You want to know how to determine the size of the array and how to access the elements in the array.

- **Solution:**
  - This sketch creates two arrays: an array of integers for pins connected to switches and an array of pins connected to LEDs, as shown in Figure 2-1:

Working with Groups of Values:

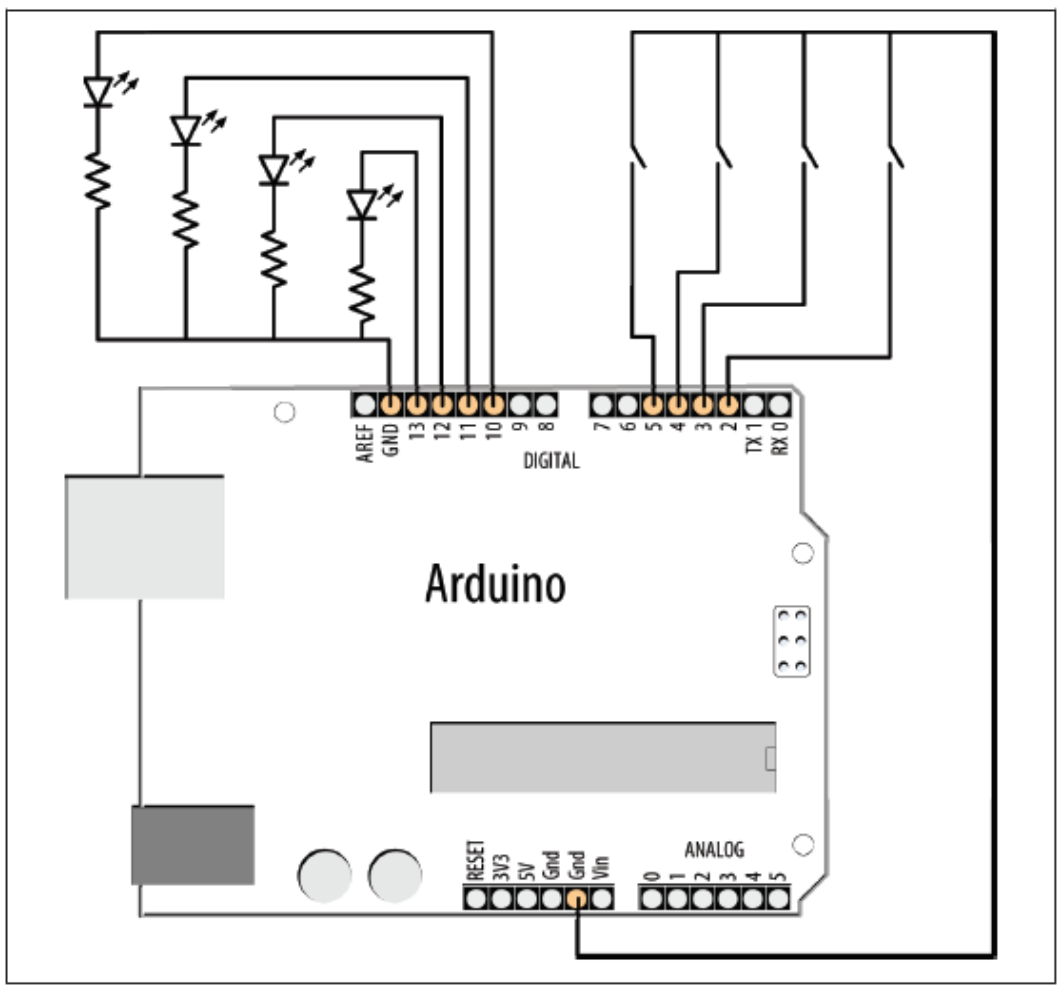


Figure 2-1. Connections for LEDs and switches

Working with Groups of Values:

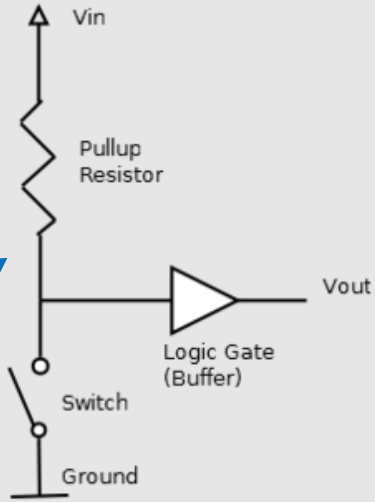
```
/*
 array sketch
 an array of switches controls an array of LEDs
 see Chapter 5 for more on using switches
 see Chapter 7 for information on LEDs
 */

int inputPins[] = {2,3,4,5};  // create an array of pins for switch inputs

int ledPins[] = {10,11,12,13};  // create array of output pins for LEDs

void setup()
{
  for(int index = 0; index < 4; index++)
  {
    pinMode(ledPins[index], OUTPUT);          // declare LED as output
    pinMode(inputPins[index], INPUT);         // declare pushbutton as input

    digitalWrite(inputPins[index],HIGH);      // enable pull-up resistors
    // (see Recipe 5.2)
  }
}
```



Working with Groups of Values:

```
void loop(){
  for(int index = 0; index < 4; index++)
  {
    int val = digitalRead(inputPins[index]);  // read input value
    if (val == LOW)                           // check if the switch is pressed
    {
      digitalWrite(ledPins[index], HIGH); // turn LED on if switch is pressed
    }
    else
    {
      digitalWrite(ledPins[index], LOW);  // turn LED off
    }
  }
}
```

Using Arduino String Functionality:

- **Problem:**
  - You want to manipulate text. You need to copy it, add bits together, and determine the number of characters.

- **Solution:**
  - The previous recipe mentioned how arrays of characters can be used to store text: these character arrays are usually called strings. Arduino has a capability called ***String*** that adds rich functionality for storing and manipulating text.

Using Arduino String Functionality:

```
/*
  Basic_Strings sketch
 */

String text1 = "This string";
String text2 = " has more text";
String text3;  // to be assigned within the sketch

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

  Serial.print( text1);
  Serial.print(" is ");
  Serial.print(text1.length());
  Serial.println(" characters long.");

  Serial.print("text2 is ");
  Serial.print(text2.length());
  Serial.println(" characters long.");

  text1.concat(text2);
  Serial.println("text1 now contains: ");
  Serial.println(text1);
}

void loop()
{
}
```

Using Arduino String Functionality:

The Serial Monitor will display the following:

```
This string is 11 characters long.
text2 is 14 characters long.
text1 now contains:
This string has more text
```

Another way to combine strings is to use the string addition operator. Add these two lines to the end of the setup code:

```
text3 = text1 + " and more";
Serial.println(text3);
```

The new code will result in the Serial Monitor adding the following line to the end of the display:

This string has more text and more

You can use the indexOf and lastIndexOf functions to find an instance of a particular character in a string.

Using Arduino String Functionality:

Table 2-2. Brief overview of Arduino String functions

`charAt(n)`	Returns the *n*th character of the `String`
`compareTo(S2)`	Compares the `String` to the given `String` S2
`concat(S2)`	Returns a new `String` that is the combination of the `String` and S2
`endsWith(S2)`	Returns true if the `String` ends with the characters of S2
`equals(S2)`	Returns true if the `String` is an exact match for S2 (case-sensitive)
`equalsIgnoreCase(S2)`	Same as `equals` but is not case-sensitive
`getBytes(buffer,len)`	Copies `len`(gth) characters into the supplied byte buffer
`indexOf(S)`	Returns the index of the supplied `String` (or character) or –1 if not found
`lastIndexOf(S)`	Same as `indexOf` but starts from the end of the `String`
`length()`	Returns the number of characters in the `String`
`replace(A,B)`	Replaces all instances of `String` (or character) A with B
`setCharAt(index,c)`	Stores the character c in the `String` at the given index
`startsWith(S2)`	Returns true if the `String` starts with the characters of S2
`substring(index)`	Returns a `String` with the characters starting from index to the end of the `String`
`substring(index,to)`	Same as above, but the substring ends at the character location before the 'to' position
`toCharArray(buffer,len)`	Copies upo `len` characters of the `String` to the supplied buffer
`toInt()`	Returns the integer value of the numeric digits in the `String`
`toLowerCase()`	Returns a `String` with all characters converted to lowercase
`toUpperCase()`	Returns a `String` with all characters converted to uppercase
`trim()`	Returns a `String` with all leading and trailing whitespace removed

Using C Character Strings:

- **Problem:**
  - You want to understand how to use raw character strings: you want to know how to create a string, find its length, and compare, copy, or append strings. The core C language does not support the Arduino-style String capability, so you want to understand code from other platforms written to operate with primitive character arrays.

- **Solution:**
  - Arrays of characters are sometimes called *character strings* (or simply *strings* for short).

Using C Character Strings:

You declare strings like this:

```
char stringA[8]; // declare a string of up to 7 chars plus terminating null
char stringB[8]  = "Arduino"; // as above and init(ialize) the string to "Arduino"
char stringC[16] = "Arduino"; // as above, but string has room to grow
char stringD[ ]  = "Arduino"; // the compiler inits the string and calculates size
```

Use strlen (short for *string length*) to determine the number of characters before the terminating null:

```
int length = strlen(string);  // return the number of characters in the string
```

length will be 0 for stringA and 7 for the other strings shown in the preceding code. The null that indicates the end of the string is not counted by strlen.

Use strcpy (short for *string copy*) to copy one string to another:

```
strcpy(destination, source);   // copy string source to destination
```

Use strncpy to limit the number of characters to copy (useful to prevent writing more characters than the destination string can hold). You can see this used in Recipe 2.7:

```
// copy up to 6 characters from source to destination
strncpy(destination, source, 6);
```

Using C Character Strings:

Use strcat (short for *string concatenate*) to append one string to the end of another:

```
// append source string to the end of the destination string
strcat(destination, source);
```

Use strcmp (short for *string compare*) to compare two strings. You can see this used in Recipe 2.7:

```
if(strcmp(str, "Arduino") == 0)
   // do something if the variable str is equal to "Arduino"
```

Splitting Comma-Separated Text into Groups:

- **Problem:**
  - You have a string that contains two or more pieces of data separated by commas (or any other separator). You want to split the string so that you can use each individual part.

- **Solution:**
  - This sketch prints the text found between each comma:

Splitting Comma-Separated Text into Groups:

```
/*
 * SplitSplit sketch
 * split a comma-separated string
 */

String  text = "Peter,Paul,Mary";  // an example string
String  message = text; // holds text not yet split
int     commaPosition;  // the position of the next comma in the string

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

  Serial.println(message);          // show the source string
  do
  {
```

Splitting Comma-Separated Text into Groups:

```
      commaPosition = message.indexOf(',');
      if(commaPosition != -1)
      {
          Serial.println( message.substring(0,commaPosition));
          message = message.substring(commaPosition+1, message.length());
      }
      else
      {  // here after the last comma is found
         if(message.length() > 0)
           Serial.println(message);  // if there is text after the last comma,
                                     // print it
      }
   }
   while(commaPosition >=0);
}

void loop()
{
}
```

Splitting Comma-Separated Text into Groups:

The Serial Monitor will display the following:

```
Peter,Paul,Mary
Peter
Paul
Mary
```

Converting a Number to a String:

- **Problem:**
  - You need to convert a number to a string, perhaps to show the number on an LCD or other display.

- **Solution:**
  - The String variable will convert numbers to strings of characters automatically. You can use literal values, or the contents of a variable. For example, the following code will work:

Converting a Number to a String:

```
String myNumber = "1234“;
```

As will this:

```
int value = 127;
String myReadout = "The reading was ";
myReadout.concat(value);
```

Or this:

```
int value = 127;
String myReadout = "The reading was ";
myReadout += value;
```

Converting a Number to a String:

compound assignments

Compound assignments combine an arithmetic operation with a variable assignment. These are commonly found in for loops as described later. The most common compound assignments include:

```
x ++      // same as x = x + 1, or increments x by +1
x --      // same as x = x - 1, or decrements x by -1
x += y    // same as x = x + y, or increments x by +y
x -= y    // same as x = x - y, or decrements x by -y
x *= y    // same as x = x * y, or multiplies x by y
x /= y    // same as x = x / y, or divides x by y
```

Converting a String to a Number:

- **Problem:**
  - You need to convert a string to a number. Perhaps you have received a value as a string over a communication link and you need to use this as an integer or floating-point value.
- **Solution:**
  - There are a number of ways to solve this. If the string is received as serial data, it can be converted on the fly as each character is received.
  - 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). This code fragment terminates the incoming digits on any character that is not a digit (or if the buffer is full). For this to work, though, you'll need to enable the newline option in the Serial Monitor or type some other terminating character:

Converting a String to a Number:

```
/*
 * StringToNumber
 * Creates a number from a string
 */

const int ledPin = 13; // pin the LED is connected to

int   blinkDelay;     // blink rate determined by this variable
char strValue[6];     // must be big enough to hold all the digits and the
                      // 0 that terminates the string
int index = 0;        // the index into the array storing the received digits

void setup()
{
 Serial.begin(9600);
 pinMode(ledPin,OUTPUT); // enable LED pin as output
}
```

Converting a String to a Number:

```
void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();
    if(index < 5 && isDigit(ch) ){
      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
      blinkDelay = atoi(strValue);  // use atoi to convert the string to an int
      index = 0;
    }
```

Converting a String to a Number:

```
  }
  blink();
}

void blink()
{
   digitalWrite(ledPin, HIGH);
   delay(blinkDelay/2);  // wait for half the blink period
   digitalWrite(ledPin, LOW);
   delay(blinkDelay/2);  // wait for the other half
}
```

Structuring Your Code into Functional Blocks:

- **Problem:**
  - You want to know how to add functions to a sketch, and the correct amount of functionality to go into your functions. You also want to understand how to plan the overall structure of the sketch.
- **Solution:**
  - Functions are used to organize the actions performed by your sketch into functional blocks. Functions package functionality into well-defined inputs (information given to a function) and outputs (information provided by a function) that make it easier to structure, maintain, and reuse your code. You are already familiar with the two functions that are in every Arduino sketch: **setup and loop**. You create a function by declaring its return type (the information it provides), its name, and any optional parameters (values) that the function will receive when it is called.

Structuring Your Code into Functional Blocks:

Here is a simple function that just blinks an LED. It has no parameters and doesn't return anything (the void preceding the function indicates that nothing will be returned):

```
// blink an LED once
void blink1()
{
   digitalWrite(13,HIGH);   // turn the LED on
   delay(500);              // wait 500 milliseconds
   digitalWrite(13,LOW);    // turn the LED off
   delay(500);              // wait 500 milliseconds
}
```

Structuring Your Code into Functional Blocks:

The following version has a parameter (the integer named count) that determines how many times the LED will flash:

```
// blink an LED the number of times given in the count parameter
void blink2(int count)
{
  while(count > 0 )  // repeat until count is no longer greater than zero
  {
    digitalWrite(13,HIGH);
    delay(500);
    digitalWrite(13,LOW);
    delay(500);
    count = count -1; // decrement count
  }
}
```

Structuring Your Code into Functional Blocks:

- Here is an example sketch that takes a parameter and returns a value. The parameter determines the length of the LED on and off times (in milliseconds).

- The function continues to flash the LED until a button is pressed, and the number of times the LED flashed is returned from the function:

Structuring Your Code into Functional Blocks:

```
/*
  blink3 sketch
  Demonstrates calling a function with a parameter and returning a value.
  Uses the same wiring as the pull-up sketch from
  Recipe 5.2

  The LED flashes when the program starts and stops when a switch connected
  to digital pin 2 is pressed.
  The program prints the number of times that the LED flashes.
 */

const int ledPin = 13;          // output pin for the LED
const int inputPin = 2;         // input pin for the switch

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(inputPin, INPUT);
  digitalWrite(inputPin,HIGH); // use internal pull-up resistor (Recipe 5.2)
  Serial.begin(9600);
}
```

Structuring Your Code into Functional Blocks:

```
void loop(){
  Serial.println("Press and hold the switch to stop blinking");
  int count = blink3(250); // blink the LED 250ms on and 250ms off
  Serial.print("The number of times the switch blinked was ");
  Serial.println(count);
}

// blink an LED using the given delay period
// return the number of times the LED flashed
int blink3(int period)
{
  int result = 0;
  int switchVal = HIGH; //with pull-ups, this will be high when switch is up

  while(switchVal == HIGH)  // repeat this loop until switch is pressed
                            // (it will go low when pressed)
  {
    digitalWrite(13,HIGH);
    delay(period);
    digitalWrite(13,LOW);
    delay(period);
    result = result + 1; // increment the count
```

Structuring Your Code into Functional Blocks:

```
    switchVal = digitalRead(inputPin);  // read input value
  }
  // here when switchVal is no longer HIGH because the switch is pressed
  return result;  // this value will be returned
}
```

Structuring Your Code into Functional Blocks:

Discussion

The code in this recipe's Solution illustrates the three forms of function call that you will come across. blink1 has no parameter and no return value. Its form is:

```
void blink1()
{
   // implementation code goes here...
}
```

blink2 takes a single parameter but does not return a value:

```
void blink2(int count)
{
   // implementation code goes here...
}
```

blink3 has a single parameter and returns a value:

```
int blink3(int period)
{
   // implementation code goes here...
}
```

Structuring Your Code into Functional Blocks:

Most of the functions you come across will be some variation on these forms. For example, here is a function that takes a parameter and returns a value:

```
int sensorPercent(int pin)
{
int percent;

  int val = analogRead(pin);  // read the sensor (ranges from 0 to 1023)
  percent = map(val,0,1023,0,100); // percent will range from 0 to 100.
  return percent;
}
```

The same functionality can be achieved without using the percent temporary variable:

```
int sensorPercent(int pin)
{
  int val = analogRead(pin);  // read the sensor (ranges from 0 to 1023)
  return map(val,0,1023,0,100); // percent will ranges from 0 to 100.
}
```

Structuring Your Code into Functional Blocks:

Here is how the function can be called:

```
// print the percent value of 6 analog pins
for(int sensorPin = 0; sensorPin < 6; sensorPin++)
{
  Serial.print("Percent of sensor on pin ");
  Serial.print(sensorPin);
  Serial.print(" is ");
  int val = sensorPercent(sensorPin);
  Serial.print(val);
}
```

Returning More Than One Value from a Function:

- **Problem:**
  - You want to return two or more values from a function. Recipe 2.10 provided examples for the most common form of a function, one that returns just one value or none at all. But sometimes you need to modify or return more than one value.

- **Solution:**
  - There are various ways to solve this. The easiest to understand is to have the function change some global variables and not actually return anything from the function:

Returning More Than One Value from a Function:

```
/*
  swap sketch
  demonstrates changing two values using global variables
 */

int x; // x and y are global variables
int y;

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

void loop(){
  x = random(10); // pick some random numbers
  y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swap();

  Serial.print("The value of x and y after swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();


  delay(1000);
}
```

Returning More Than One Value from a Function:

```
// swap the two global values
void swap()
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

Returning More Than One Value from a Function:

```
/*
  functionReferences sketch
  demonstrates returning more than one value by passing references
 */
```

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

void loop(){
  int x = random(10); // pick some random numbers
  int y = random(10);

  Serial.print("The value of x and y before swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);
  swap(x,y);

  Serial.print("The value of x and y after swapping are: ");
  Serial.print(x); Serial.print(","); Serial.println(y);Serial.println();

  delay(1000);
}
```

Returning More Than One Value from a Function:

```
// swap the two given values
void swap(int &value1, int &value2)
{
  int temp;
  temp = value1;
  value1 = value2;
  value2 = temp;
}
```

Taking Actions Based on Conditions:

- **Problem:**
  - You want to execute a block of code only if a particular condition is true. For example, you may want to light an LED if a switch is pressed or if an analog value is greater than some threshold.

- **Solution:**
  - The following code uses the wiring shown in Recipe 5.1:

Taking Actions Based on Conditions:

```
/*
   Pushbutton sketch
   a switch connected to digital pin 2 lights the LED on pin 13
*/

const int ledPin = 13;              // choose the pin for the LED
const int inputPin = 2;             // choose the input pin (for a pushbutton)

void setup() {
  pinMode(ledPin, OUTPUT);          // declare LED pin as output
  pinMode(inputPin, INPUT);         // declare pushbutton pin as input
}

void loop(){
  int val = digitalRead(inputPin);  // read input value
  if (val == HIGH)                  // check if the input is HIGH
  {
    digitalWrite(ledPin, HIGH);     // turn LED on if switch is pressed
  }
}
```

Taking Actions Based on Conditions:

If you want to do one thing if a statement is true and another if it is false, use the if...else statement:

```
/*
   Pushbutton sketch
   a switch connected to pin 2 lights the LED on pin 13
*/

const int ledPin = 13;            // choose the pin for the LED
const int inputPin = 2;           // choose the input pin (for a pushbutton)

void setup() {
  pinMode(ledPin, OUTPUT);         // declare LED pin as output
  pinMode(inputPin, INPUT);        // declare pushbutton pin as input
}

void loop(){
  int val = digitalRead(inputPin);  // read input value
  if (val == HIGH)                   // check if the input is HIGH
  {
    // do this if val is HIGH
    digitalWrite(ledPin, HIGH);     // turn LED on if switch is pressed
  }
  else
  {
    // else do this if val is not HIGH
    digitalWrite(ledPin, LOW);      // turn LED off
  }
}
```

Repeating a Sequence of Statements:

- **Problem:**
- You want to repeat a block of statements while an expression is true.

- **Solution:**
- A while loop repeats one or more instructions while an expression is true:

Repeating a Sequence of Statements:

```
/*
 * Repeat
 * blinks while a condition is true
 */

const int ledPin    = 13; // digital pin the LED is connected to
const int sensorPin = 0; // analog input 0

void setup()
{
  Serial.begin(9600);
  pinMode(ledPin,OUTPUT); // enable LED pin as output
}

void loop()
{
  while(analogRead(sensorPin) > 100)
  {
    blink();    // call a function to turn an LED on and off
    Serial.print(".");
  }
  Serial.println(analogRead(sensorPin)); // this is not executed until after
                                         // the while loop finishes!!!
}
```

Repeating a Sequence of Statements:

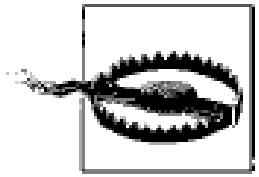
```
void blink()
{
   digitalWrite(ledPin, HIGH);
   delay(100);
   digitalWrite(ledPin, LOW);
   delay(100);
}
```

Repeating a Sequence of Statements:

The do...while loop is similar to the while loop, but the instructions in the code block are executed before the condition is checked. Use this form when you must have the code executed at least once, even if the expression is false:

```
do
{
   blink();  // call a function to turn an LED on and off
}
while (analogRead(sensorPin) > 100);
```

Repeating a Sequence of Statements:



Only the code within a while or do loop will run until the conditions permit exit. If your sketch needs to break out of a loop in response to some other condition such as a timeout, sensor state, or other input, you can use break:

```
while(analogRead(sensorPin) > 100)
{
   blink();
   if(Serial.available())
     break; // any serial input breaks out of the while loop
}
```

Repeating Statements with a Counter:

- **Problem:**
  - You want to repeat one or more statements a certain number of times. The for loop is similar to the while loop, but you have more control over the starting and ending conditions.

- **Solution:**
  - This sketch counts from zero to three by printing the value of the variable *i* in a for loop:

Repeating Statements with a Counter:

```
/*
   ForLoop sketch
   demonstrates for loop
*/

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

void loop(){
  Serial.println("for(int i=0; i < 4; i++)");
  for(int i=0; i < 4; i++)
  {
    Serial.println(i);
  }
}
```

The Serial Monitor output from this is as follows (it will be displayed over and over):

```
for(int i=0; i < 4; i++)
0
1
2
3
```

Repeating Statements with a Counter:

A for loop can use an existing variable, or it can create a variable for exclusive use inside the loop. This version uses the value of the variable j created earlier in the sketch:

```
int j;

Serial.println("for(j=0; j < 4; j++ )");
for(j=0; j < 4; j++ )
{
  Serial.println(j);
}
```

This is almost the same as the earlier example, but it does not have the int keyword in the initialization part because the variable j was already defined. The output of this version is similar to the output of the earlier version:

```
for(j=0; j < 4; j++)
0
1
2
3
```

Repeating Statements with a Counter:

You can leave out the initialization part completely if you want the loop to use the value of a variable defined earlier. This code starts the loop with j equal to 1:

```
int j = 1;

Serial.println("for(    ; j < 4; j++ )");
for(  ; j < 4; j++ )
{
  Serial.println(j);
}
```

The preceding code prints the following:

```
for(  ; j < 4; j++)
1
2
3
```

Repeating Statements with a Counter:

The following code tests if the value of the loop variable is less than or equal to 4. It will print the digits from 0 to 4:

```
Serial.println("for(int i=0; i <= 4; i++)");
for(int i=0; i <= 4; i++)
{
  Serial.println(i);
}
```

The third part of a for loop is the iterator statement that gets executed at the end of each pass through the loop. This can be any valid C/C++ statement. The following increases the value of i by two on each pass:

```
Serial.println("for(int i=0; i < 4; i+= 2)");
for(int i=0; i < 4; i+=2)
{
  Serial.println(i);
}
```

Repeating Statements with a Counter:

The *iterator* expression can be used to cause the loop to count from high to low, in this case from 3 to 0:

```
Serial.println("for(int i=3; i > = 0 ; i--)");
for(int i=3; i >= 0 ; i--)
{
  Serial.println(i);
}
```

Repeating Statements with a Counter:

Like the other parts of a for loop, the iterator expression can be left blank (you must always have the two semicolons separating the three parts even if they are blank).

This version only increments i when an input pin is high. The for loop does not change the value of i; it is only changed by the if statement after Serial.print—you'll need to define inPin and set it to INPUT with pinMode():

```
Serial.println("for(int i=0; i < 4; )");
for(int i=0; i < 4; )
{
  Serial.println(i);
  if(digitalRead(inPin) == HIGH) {
    i++;  // only increment the value if the input is high
  }
}
```

Breaking Out of Loops:

- **Problem:**
  - You want to terminate a loop early based on some condition you are testing.

- **Solution:**
  - Use the following code:

Breaking Out of Loops:

```
while(analogRead(sensorPin) > 100)
{
   if(digitalRead(switchPin) == HIGH)
   {
      break;   //exit the loop if the switch is pressed
   }
   flashLED();              // call a function to turn an LED on and off
}
```

Taking a Variety of Actions Based on a Single Variable:

- **Problem:**
  - You need to do different things depending on some value. You could use multiple if and else if statements, but the code soon gets complex and difficult to understand or modify. Additionally, you may want to test for a range of values.
- **Solution:**
  - The switch statement provides for selection of a number of alternatives. It is functionally similar to multiple if/else if statements but is more concise:

Taking a Variety of Actions Based on a Single Variable:

```
/*
 * SwitchCase sketch
 * example showing switch statement by switching on chars from the serial port
 *
 * sending the character 1 blinks the LED once, sending 2 blinks twice
 * sending + turns the LED on, sending - turns it off
 * any other character prints a message to the Serial Monitor
 */
const int ledPin = 13; // the pin the LED is connected to

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

Taking a Variety of Actions Based on a Single Variable:

```
void loop()
{
  if ( Serial.available()) // Check to see if at least one
                           // character is available
  {
    char ch = Serial.read();
    switch(ch)
    {
    case '1':
      blink();
      break;
    case '2':
      blink();
      blink();
      break;
    case '+':
      digitalWrite(ledPin,HIGH);
      break;
```

Taking a Variety of Actions Based on a Single Variable:

```
    case '-':
      digitalWrite(ledPin,LOW);
      break;
    default :
      Serial.print(ch);
      Serial.println(" was received but not expected");
      break;
    }
  }
}

void blink()
{
  digitalWrite(ledPin,HIGH);
  delay(500);
  digitalWrite(ledPin,LOW);
  delay(500);
}
```

Taking a Variety of Actions Based on a Single Variable:

```
case '1':
  blink();   // no break statement before the next label
case '2':
  blink();   // case '1' will continue here
  blink();
  break;     // break statement will exit the switch expression
```

If the break statement at the end of case '1': was removed (as shown in the preceding code), when ch is equal to the character 1 the blink function will be called three times. Accidentally forgetting the break is a common mistake. Intentionally leaving out the break is sometimes handy; it can be confusing to others reading your code, so it's a good practice to clearly indicate your intentions with comments in the code.

If your switch statement is misbehaving, check to ensure that you have not forgotten the break statements.

The default: label is used to catch values that don't match any of the case labels. If there is no default label, the switch expression will not do anything if there is no match.

Comparing Character and Numeric Values:

- **Problem:**
  - You want to determine the relationship between values.

- **Solution:**
  - Compare integer values using the relational operators shown in Table 2-3.

Comparing Character and Numeric Values:

Table 2-3. Relational and equality operators

Operator	Test for	Example
==	Equal to	`2 == 3 // evaluates to false`
!=	Not equal to	`2 != 3 // evaluates to true`
>	Greater than	`2 > 3 // evaluates to false`
<	Less than	`2 < 3 // evaluates to true`
>=	Greater than or equal to	`2 >= 3 // evaluates to false`
<=	Less than or equal to	`2 <= 3 // evaluates to true`

Comparing Character and Numeric Values:

The following sketch demonstrates the results of using the comparison operators:

```
/*
 * RelationalExpressions sketch
 * demonstrates comparing values
 */

int i = 1;  // some values to start with
int j = 2;

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

void loop(){
  Serial.print("i = ");
  Serial.print(i);
  Serial.print(" and j = ");
  Serial.println(j);

  if(i < j)
    Serial.println(" i is less than j");
  if(i <= j)
    Serial.println(" i is less than or equal to j");
```

Comparing Character and Numeric Values:

```
  if(i != j)
    Serial.println(" i is not equal to j");
  if(i == j)
    Serial.println(" i is equal to j");
  if(i >= j)
    Serial.println(" i is greater than or equal to j");
  if(i > j)
    Serial.println(" i is greater than j");

  Serial.println();
  i = i + 1;
  if(i > j + 1)
    delay(10000);  // long delay after i is no longer close to j
}
```

Comparing Character and Numeric Values:

Here is the output:

```
i = 1 and j = 2
 i is less than j
 i is less than or equal to j
 i is not equal to j

i = 2 and j = 2
 i is less than or equal to j
 i is equal to j
 i is greater than or equal to j

i = 3 and j = 2
 i is not equal to j
 i is greater than or equal to j
 i is greater than j
```

Comparing Character and Numeric Values:

Discussion

Note that the equality operator is the double equals sign, ==. One of the most common programming mistakes is to confuse this with the assignment operator, which uses a single equals sign.

The following expression will compare the value of i to 3. The programmer intended this:

```
if(i == 3)  // test if i equals 3
```

But he put this in the sketch:

```
if(i = 3)  // single equals sign used by mistake!!!!
```

This will always return true, because i will be set to 3, so they will be equal when compared.

Comparing Strings:

- **Problem:**
  - You want to see if two character strings are identical.

- **Solution:**
  - There is a function to compare strings, called strcmp (short for *string compare*). Here is a fragment showing its use:

Comparing Strings:

```
char string1[ ] = "left";
char string2[ ] = "right";

if(strcmp(string1, string2) == 0)
   Serial.print("strings are equal)
```

Discussion

`strcmp` returns the value `0` if the strings are equal and a value greater than zero if the first character that does not match has a greater value in the first string than in the second. It returns a value less than zero if the first nonmatching character in the first string is less than in the second. Usually you only want to know if they are equal, and although the test for zero may seem unintuitive at first, you'll soon get used to it.

Comparing Strings:

Bear in mind that strings of unequal length will not be evaluated as equal even if the shorter string is contained in the longer one. So:

```
strcmp("left", "leftcenter") == 0)  // this will evaluate to false
```

You can compare strings up to a given number of characters by using the strncmp function. You give strncmp the maximum number of characters to compare and it will stop comparing after that many characters:

```
strncmp("left", "leftcenter", 4) == 0)  // this will evaluate to true
```

Unlike character strings, Arduino Strings can be directly compared as follows:

```
String stringOne = String("this");
if (stringOne == "this")
  Serial.println("this will be true");
if (stringOne == "that")
  Serial.println("this will be false");
```

Performing Logical Comparisons:

- **Problem:**
  - You want to evaluate the logical relationship between two or more expressions. For example, you want to take a different action depending on the conditions of an if statement.

- **Solution:**
  - Use the logical operators as outlined in Table 2-4.

Performing Logical Comparisons:

Table 2-4. Logical operators

Symbol	Function	Comments
&&	Logical And	Evaluates as `true` if the conditions on both sides of the && operator are true
||	Logical Or	Evaluates as `true` if the condition on at least one side of the || operator is true
!	Not	Evaluates as `true` if the expression is false, and `false` if the expression is true

Performing Logical Comparisons:

Discussion

Logical operators return true or false values based on the logical relationship. The examples that follow assume you have sensors wired to digital pins 2 and 3 as discussed in Chapter 5.

The logical And operator && will return true if both its two operands are true, and false otherwise:

```
if( digitalRead(2) && digitalRead(3) )
  blink(); // blink of both pins are HIGH
```

The logical Or operator || will return true if either of its two operands are true, and false if both operands are false:

```
if( digitalRead(2) || digitalRead(3) )
  blink(); // blink of either pins is HIGH
```

The Not operator ! has only one operand, whose value is inverted—it results in false if its operand is true and true if its operand is false:

```
if( !digitalRead(2)  )
  blink(); // blink of the pin is not HIGH
```

Performing Bitwise Operations:

- **Problem:**
  - You want to set or clear certain bits in a value.

- **Solution:**
  - Use the bit operators as outlined in Table 2-5.

Performing Bitwise Operations:

Table 2-5. Bit operators

Symbol	Function	Result	Example
&	Bitwise And	Sets bits in each place to 1 if both bits are 1; otherwise, bits are set to 0.	3 & 1 equals 1 (11 & 01 equals 01)
|	Bitwise Or	Sets bits in each place to 1 if either bit is 1.	3 | 1 equals 3 (11 | 01 equals 11)
^	Bitwise Exclusive Or	Sets bits in each place to 1 only if one of the two bits is 1.	3 ^ 1 equals 2 (11 ^ 01 equals 10)
~	Bitwise Negation	Inverts the value of each bit. The result depends on the number of bits in the data type.	~1 equals 254 (~00000001 equals 11111110)

Performing Bitwise Operations:

```
/*
 * bits sketch
 * demonstrates bitwise operators
 */

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

void loop(){
  Serial.print("3 & 1 equals "); // bitwise And 3 and 1
  Serial.print(3 & 1);          // print the result
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 & 1 , BIN);   // print the binary representation of the result

  Serial.print("3 | 1 equals "); // bitwise Or 3 and 1
  Serial.print(3 | 1 );
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 | 1 , BIN);   // print the binary representation of the result

  Serial.print("3 ^ 1 equals "); // bitwise exclusive or 3 and 1
  Serial.print(3 ^ 1);
  Serial.print(" decimal, or in binary: ");
  Serial.println(3 ^ 1 , BIN);   // print the binary representation of the result
```

Performing Bitwise Operations:

```
    byte byteVal = 1;
    int intVal = 1;

    byteVal = ~byteVal;  // do the bitwise negate
    intVal = ~intVal;

    Serial.print("~byteVal (1) equals "); // bitwise negate an 8 bit value
    Serial.println(byteVal, BIN);  // print the binary representation of the result
    Serial.print("~intVal (1) equals "); // bitwise negate a 16 bit value
    Serial.println(intVal, BIN);   // print the binary representation of the result

    delay(10000);
}
```

Determine what will be displayed on the serial monitor!!??

2:00

Performing Bitwise Operations:

```
    byte byteVal = 1;
    int intVal = 1;

    byteVal = ~byteVal;  // do the bitwise negate
    intVal = ~intVal;

    Serial.print("~byteVal (1) equals "); // bitwise negate an 8 bit value
    Serial.println(byteVal, BIN);  // print the binary representation of the result
    Serial.print("~intVal (1) equals "); // bitwise negate a 16 bit value
    Serial.println(intVal, BIN);   // print the binary representation of the result

    delay(10000);
  }
```

This is what is displayed on the Serial Monitor:

```
3 & 1 equals 1 decimal, or in binary: 1
3 | 1 equals 3 decimal, or in binary: 11
3 ^ 1 equals 2 decimal, or in binary: 10
~byteVal (1) equals 11111110
~intVal (1) equals 11111111111111111111111111111110
```

Performing Bitwise Operations:

Discussion

Bitwise operators are used to set or test bits. When you "And" or "Or" two values, the operator works on each individual bit. It is easier to see how this works by looking at the binary representation of the values.

Table 2-6. Bitwise And

Bit 1	Bit 2	Bit 1 and Bit 2
0	0	0
0	1	0
1	0	0
1	1	1

Table 2-7. Bitwise Or

Bit 1	Bit 2	Bit 1 or Bit 2
0	0	0
0	1	1
1	0	1
1	1	1

Table 2-8. Bitwise Exclusive Or

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
0	1	1
1	0	1
1	1	0

Combining Operations and Assignment:

- **Problem:**
  - You want to understand and use compound operators. It is not uncommon to see published code that uses expressions that do more than one thing in a single statement. You want to understand a += b, a >>= b, and a &= b.

- **Solution:**
  - Table 2-9 shows the compound assignment operators and their equivalent full expression.

Combining Operations and Assignment:

Table 2-9. Compound operators

Operator	Example	Equivalent expression
+=	value += 5;	value = value + 5; // add 5 to value
-=	value -= 4;	value = value - 4; // subtract 4 from value
*=	value *= 3;	value = value * 3; // multiply value by 3
/=	value /= 2;	value = value / 2; // divide value by 2
>>=	value >>= 2;	value = value >> 2; // shift value right two places
<<=	value <<= 2;	value = value << 2; // shift value left two places
&=	mask &= 2;	mask = mask & 2; // binary-and mask with 2
|=	mask |= 2;	mask = mask | 2; // binary-or mask with 2

Combining Operations and Assignment:

Discussion

These compound statements are no more efficient at runtime than the equivalent full expression, and if you are new to programming, using the full expression is clearer. Experienced coders often use the shorter form, so it is helpful to be able to recognize the expressions when you run across them.