Essential Arduino

History

The personal computer revolution of the late 20th century was an accident. Really. Intel and the other silicon valley corporations were not trying to create portable general purpose computers—after all who would buy such a thing? (IBM estimated the total market for computers: banks, governments, and research labs, at a couple of thousand at most.) The Intel 8008 chip that started it all was intended to run a handheld calculator. It was envisioned as the first of a line of "programmable microcontrollers" (also called microprocessors), devices that would make it possible to design circuits for complex operations with a minimum number of chips. Today such chips are everywhere, from toothbrushes to 787s. This definitely includes musical instruments—the Prophet 5, DX 7 and all following digital instruments are built around microcontrollers.

The essence of a microcontroller is it has a single job to do. This can be fairly complex, but the micro is focused on that job—it is not expected to also Skype and check email. You could think of it as a program fixed in silicon. In the early days, the program had to be burned into the silicon at the factory- the code was developed on emulators, which cost a fair amount of money ($10-20,000). Getting final product involved minimum orders in the thousands. Hobbyists and electronic musicians were mostly shut out of the system by the costs. (Not everybody- David Behrman and George Lewis worked extensively with a $300 emulation board called the KIM-1.) In the 90s the Microchip company brought out a line of field programmable microprocessors called PIC. This spawned a series of project boards and kits, of which the most prominent is the Basic Stamp. The Stamp addressed the second barrier to widespread use of microprocessors—the lack of high level programming languages. Early microprocessors did not have enough power to run languages like C or Java—they were programmed in assembly language, or required a proprietary compiler (another expensive item). The PIC can be programmed in Basic. PIC systems cost about $200[1] for a programmer board and $60 per system board (one needed for each project, unless you built your own), and some musicians and artists started using them. (Since Arduino came out, these prices have fallen to be competitive.)

Arduino was developed by a pair of teachers, Massimo Banzi and David Cuartielles, at a short lived Italian Design school. (See http://spectrum.ieee.org/geek-life/hands-on/the-making-of-arduino/0). The key to Arduino is a powerful open source programming language called Wiring written by graduate student Hernando Barraganat, working with professor Casey Reas, who is well known as one of the inventors of the Processing language.

[1] $300 in 1995 was much less money than $300 in 1975.

Arduino is revolutionary for two reasons—a complete board cost about $35, (Less for stripped down versions), and the programing language is free and well supported by on-line documentation and discussion groups. The language can be expanded by third party libraries, and many talented programmers are doing just that. This system obviously works- over a quarter million Arduino boards have been sold.

Overview

Figure 1.

All microcontroller systems have three essential functions:

- Gather data from attached sensors and controls
- Process that data according to the program
- Send data to output devices

Arduino is really good at this because of the Atmel Atmega328 microprocessor at its heart. This chip has 19 pins that can be either inputs or outputs. (You define the function of pins as part of the program.) Some pins have further capabilities. Here's a list of what can be done:

- Input a bit (all pins).
- Output a bit (all pins).
- Output a control voltage via pulse width modulation (PWM) (6 pins)
- Read a control voltage (six pins)
- Produce a servo control signal (two pins)
- Read or write RS-232 serial data (this kind of serial communications enables the chip to talk to computers via USB or other devices via MIDI.)
- Read or write I2C serial data (this is a protocol that communicates with chips on the same circuit board. This includes high tech devices such as GPS units.)
- Read or write TCP serial data for network communications.

In practical terms, here are things that can be used as data sources:

Bit Inputs

Anything that looks electrically like a switch (on-off) can control a bit input. Toggles, push buttons, foot pedals, some kinds of light sensors, thermostats, magnetic sensors are all switches. A cute type of switch called a limit switch can detect the motion of practically anything. Of course, bits can be combined into numbers- this number could indicate the position of a rotary control or (via a diode matrix circuit) which of a large number of keys is pressed.

Voltage inputs

The last six pins on the Arduino can be configured to read a voltage that matches the voltage range of the chip itself. (usually 0-5 volts--be careful not to exceed this limit.) Most rotary knobs and sliders are attached to potentiometers, which output a voltage proportional to their position. Many other devices, like sonar sensors, also produce a voltage output. One thing you can't do with these inputs is record audio. They just aren't fast enough. (They can detect something like a drum hit, though.)

I2C

I2C is a method many microprocessors use to communicate with each other. This is a serial system, which means bits are transferred one at a time, but very quickly. Chip based systems like accelerometers and compasses use I2C communications. I2C is a buss based system, which means many devices can be connected to the same input pin. A second pin is required to talk back or query devices. There is a variation of this system called SPI. Arduino does this too.

RS-232

RS-232 is a serial communications system designed for communications between computers and peripheral devices like modems and printers. This require some extra parts, primarily to protect the system from damage due to mis-connection. (With I2C you just have to be careful.) These parts are included on the Arduino, (connected to pins 0 and 1) and make the Arduino appear as a USB to serial convertor to a computer. (You may see a network connection dialog, which you can just close.)

TCP and beyond

Arduino can connect to the internet with the addition of a circuit board called a "Shield". Shields are boards roughly the size of the Arduino that are designed to plug directly into the Arduino headers. Most shields have headers that duplicate the Arduino, so several shields can be stacked up. In addition to internet, shields are available for high power motor control, relays, LCD displays, joysticks, cellphone network (you need a SIMM), Wifi and several other radio systems.

Here are some things to control, either via shields or homemade circuitry.

Bit output

A single bit from the Arduino is not very powerful electrically, but conceptually it is awesome. It can turn on a light from an LED directly connected to the board to a searchlight via an industrial relay. Relays can control anything electric, as long as the relay is up to the power demand. Low power relays may be made from a single transistor. A solenoid is similar to a relay, but instead of closing a switch, it moves a piston. This could hit a drum, open a window or raise a flag.

A group of bits may control a complex switch which selects which of many sources to read, thus expanding Arduino's capacity well beyond 19 inputs. A group of bits can also control a type of motor called a stepper to point it to a precise position.

PWM

There is no direct analog voltage output, but several pins (3,5,6,9,10 or 11) can be set to produce a pulse wave at 500 Hz. The duty cycle of the pulse wave can be adjusted from 0 to 50%. A light attached to this will behave exactly as if the power were being reduced. So will a motor. In fact, a motor controlled this way has a wider range of speed that it would with variable voltage. It's not much of a trick to convert the pulse to a steady voltage in those cases where it is truly necessary.

Servos

The miniature servo motors developed for radio control airplanes are very handy gadgets. Basically, they will point to a position defined by the pulse rate on a control input. The motors range from tiny rudder controls to heavy duty linear pistons. Two pins on the Arduino support this type of control. These are great for building robots that play drums—the servo lines up the stick and solenoids swing the sticks.

For more about connecting Arduinos to the world, see my Electronics for Contraptions tutorial.

Programming Basics

The Arduino world includes a programming language that is a variant of the C programming language.

Learning how

There are three traditional ways to learn to program with any language:

- Take a course.
- Buy a book.
- Read lots of code examples and experiment with your own.

Only the third method will teach you much. I have nothing against formal programming courses (after all, I teach them) but in my experience the class either holds students back or goes too fast. Lectures are a good way to learn about algorithms (methods for solving problems), but the real learning comes from looking at code and trying things out. (A good teacher will get you to do that.)

I do recommend you get a good book. A good one is full of code examples with understandable explanations. I like Practical Arduino, by Hugh Blemings. This is more than a book—the author maintains an excellent website full of example code and things to try. (http://www.practicalarduino.com/about) I also like the offerings from Makershed, which tend to focus on specific topics. Some of these are available as eBooks or PDFs.

Using the Reference

I've been writing code since 1968[2], and I always keep a reference open, no matter what language I am working in. Computer languages are supposed to be structured and methodical, but there are always little inconsistencies that will trip you up. For instance if you use the string data type to create a sentence, you need to add a \0 (the value 0) to the end of the string, but if you use the String class you don't. The reference makes good reading, too. Think of it as a mystery novel. Among other tidbits you will find this information:

The reference includes a list of all *operators* you can use in Arduino code. Operators are simple math actions like + and / along with some more obscure things such as &&. They will usually turn up in code between two operands, for example 2+2, although some operators are unary like ++x (increment the value in variable x.) Follow the links to get details.

The reference includes a list of built-in *functions*. Functions perform more complex operations—maybe math, maybe getting data or sending it out. More functions can be added to your code by importing *libraries*, and you can write your own.

The reference includes a list of *constants*. Constants are words that many functions can accept instead of a value. These are provided to make the code easier to understand. HIGH and LOW might make more sense than 1 or 0.\, at least when you are talking about output pin states.

Make Comments

The most important lines in the code are not part of the program at all. Any line that begins with // or is enclosed in /* and */ is ignored when the code is sent to the board. Use this feature to write notes to yourself. Describe what each section of code is supposed to do. Comments help in two ways:

They make the organization of the code clear. Just like the headings in this paper, comments in the code identify the important sections. This ensures they are in order and lets you find them quickly when things go wrong. The easiest way to design a program is to write the comments first. Describe each step in the process, then when the descriptions all make sense, go back and add the code to implement the descriptions.

They make it possible to maintain the code. When you come back to a program after a few months, you won't have the slightest idea what the code means. If you don't believe me, look at some code written by a stranger:

```
void loop()  {
        for(int fV = 0 ; fV <= 255; fV+=5) {
                analogWrite(13, fV);
        delay(30);
        }
```

[2] Yes, in FORTRAN.

```
        for(int fV = 255 ; fV >= 0; fV -=5) {
        analogWrite(13, fV);
        delay(30);
 }
}
```

Listing 1.
Now look at this:

```
void loop()  {
  // fade in from min to max in increments of 5 points:
  for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
  // fade out from max to min in increments of 5 points:
  for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -=5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
}
```

Listing 2.

Easier to understand isn't it? Even if you don't know what the functions mean, you can tell this is about turning something on, then off. Always write comments that can be understood by a stranger, because in six months that stranger will be you.

Hidden Code: the structure of an Arduino Program

One of the things that makes Arduino attractive is that a lot of the code is already written, and you never have to look at it. This is stuff that initializes registers, sets up timers, looks at the serial port for new instructions, sets up a run loop and a hundred other things. This is complicated, and has to be customized for each version of the Arduino board. But you and I never have to look at it. Arduino is running a program right out of the box. (You can tell by watching the lights as a virgin board boots up.) This program has two places reserved for our code. A function called setup() is called once when the program is starting up, and a function called loop() is called over and over after that.

[Digression for definition—a function is a chunk of code with a name. Programs tend to repeat certain operations over and over. If we give a name to a section of code, we can include that code anywhere by just typing the name. Function names include a pair of parentheses() stuck on the end. We can put data for the function to use in the parentheses. Functions can return a value. This value will be used in the surrounding code as if it had been placed exactly where the function call was.]

So, any Arduino program has two parts. Here's an example:

```
/*
  Blink
  Turns on an LED on for one second, then off for one second,
repeatedly.
  This example code is in the public domain.
 */
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Listing 3.

Let's deconstruct this. The top section is between /* and */ so it does nothing. Extended comments at the top of the page are called a header, and usually explain what the code is for (Including the name of the program is a nice touch—things get confusing if you have a lot of windows open. You can put your own name here too.)

Setup is the first function to define. (Details about function definition follow) One thing that must be done in the setup is set the modes of pins that will be used, and the blink program uses pin 13. pinMode is a built-in function that sets the specified pin to the specified mode. When we call pinMode, we must include the pin number and a constant to indicate the mode in the parentheses. Values in the parenthesis are called arguments. When there is more than one argument, they must be in the right order and separated by commas.

The action is in the loop function. The comments tell the tale—a series of built-in functions to turn the LED on, wait, turn the LED off, and wait. This is the end of the function, but the loop function is repeated until the power is turned off, so the LED will blink on and off, and on and off....

Experienced coders will realize that the built-in Arduino code is something like this:

```
Main(){
/*pages and pages of initialization*/
setup();
while(1)
      loop();
}
```

Listing 4.

More about Function Definitions

We will be defining more functions than just setup() and loop(). We'll use a function for any code that happens twice, and sometimes just to keep the program structure clear. Functions are defined by a magic incantation:

```
returntype  functionName(argType argument){
The code;
}
```

Listing 5.

- Return type is the data type of the value the function returns. If it does not return a value, put *void*.

[digression—data types tell the complier how much space to reserve for chunks of data and how to treat that data in math operations. Common types are integers and floats and chars (the number represents a letter). More types define the size of the data—an int in Arduino has 16 bits and a long has 32. There are 16 types in Arduino, look them over in the reference.]

- The function name will be used to call this function. Make the name descriptive of what it does: *now*() is not a good function name. *getElapsedTime*() is.
- The arguments are values the function can use in its calculations. An argument is specified by a type followed by a name. When the function code sees this name, the value used in the function call will be used.
- The braces surround the code in the function. Braces are used throughout a program to organize the code into sections. Sections often contain subsections, which contain sub-subsections, so the brace pattern can get complicated. Luckily the editor will help keep braces straight. Put the cursor after a closing brace and the matching open brace will be highlighted with a box. Double click after any brace and the enclosed code will be selected.

The code consists of a list of operations that are executed in order. Lines usually have a structure like this:

```
goesHere = something;
```

goesHere represents a variable, a place to put data. (More about variables to follow) The equal sign represents assignment, or putting a value in the place the variable refers to. The *something* here can be a different variable or more code. The semicolon marks the end of the line for the compiler. Leaving semicolons off is a very common error. (The compiler does not notice spaces, tabs and carriage returns. They simply make code easy for humans to read.)

An expanded something might be

```
goesHere = 4*5+2;
```

In this, goesHere winds up holding the result of the calculation. The value is 22, because multiplications (or divisions) are performed before additions. The formula 2+4*5 will produce the same value. You can change the order of operation with parentheses. 4 * (5 +2) produces 28.

Not all lines make assignments:

analogWrite(11, 2+4*5);

This line sends 22 to pin 11 (setting the pulse width). When a calculation is performed, the result is used wherever the calculation was, in this case as an argument to analogWrite.

Here's an example of a function in action:

```
void flash(pinNumber){
      digitalWrite(pinNumber,HIGH);
      delay(200);
      digitalWrite(pinNumber,LOW);
      delay(800);
}

void loop(){
      flash(13);
      delay(1000);
}
Listing 6.
```

Can you work out what this does?[3]

More about Variables

Variables are places in memory where data may be stored for later use. When you write the variable name in code, the value the variable contains is used there. You must define a variable before using it. You define a variable with this incantation:

```
int goesHere = 0;
```

The equal sign and value are optional. They just set the initial value of the variable, which otherwise is unpredictable. You can define a variable anywhere, but the location of the definition will have ramifications;

- A variable defined inside a function definition is temporary, and only works in the function. These are called local variables.
- A variable defined outside all functions is global, and can be used everywhere. It is traditional to put all global variable definitions at the top of the page, just after the header.

[3] The LED on pin 13 will flash briefly every two seconds.

- You can define the same variable name in different functions. They won't conflict.
- If you define a local variable with a global variable name, that function will use the local value. The global value is hidden.

Naming variables is an art. If you look back at the code example I used for comments, you will notice that the variable fV became fadeValue. A well named variable can work as well as comments in keeping the meaning of code clear. Most programmers use a handful of terse names like X or i as local variables and more explicit names in global variables. When I do this, I am rigorously consistent- to me i is always a loop index.

Branching Code

Decisions

Code is generally executed in strict order, but there are exceptions. Sometimes a line is only called in certain circumstances. This allows the program to adapt to conditions. Suppose you wanted to turn on an LED when a button connected to pin 3 was pressed. Then you would include this code in the loop() function:

```
if(digitalRead(3) == HIGH)
      digitalWrite(13,HIGH);
else
      digitalWrite(13,LOW);
```

Listing 7.

The if statement requires the result of a test as an argument. Usually we put the entire test in there, but a variable is sometimes used. Tests compare two values and return 1 if the conditions are true[4] and 0 if they are not. These are the available tests:

A==B returns 1 if A equals B. Note that there are two equal signs.
A<B returns 1 of A is less than B.
A>B returns 1 if A is greater than B.
A<=B returns 1 if A is less than or equal to B.
A>=B returns 1 if A is greater than or equal to B.
A!=B returns 1 if A is not equal to B.

You can build complex tests with logic operators && and ||, which represent Boolean logic AND and OR.

testA()&&testB() is 1 only if both functions return 1.
testA()||testB() is 1 if either function returns 1.

[4] The Arduino language defines constants true as 1 and false as 0 if you prefer to use words for your logic. Actually, any non zero value is true, only 0 is not, at least as far as if() is concerned.

The if() statement controls execution of the following line. The else that follows that is optional. Note that there are no semicolons after if or else. If if() must control more than one line, enclose them in braces:

```
if(digitalRead(3) == HIGH){
      digitalWrite(12,HIGH);
      digitalWrite(13,LOW);
}else{
      digitalWrite(12,LOW);
      digitalWrite(13,HIGH);
}
```

Listing 8.

Again, notice how the semicolons are used. Only the code lines have them. If, else and that end brace do not.

There are other ways to make choices—look at the switch() statement to choose one of three or more options.

Repeating Yourself

You might want to repeat an action within the main loop. For instance, the following will flash[5] the LED 3 times, then wait 10 seconds before blinking again:

```
Loop(){
      for(int i = 0; i<3,++i){
            flash(13);
      }
      delay(10000);
}
```

Listing 9.

The for statement is the usual way to repeat actions. It takes three arguments, each of which is itself a bit of code. Note the arguments are separated by semicolons, not commas.

- The first argument declares a variable to use for counting repetitions. You can use an existing variable or like this example, define one on the spot. The variable is usually initialized to 0, but there are exceptions.
- The second argument is a test to find if the loop is finished. More precisely, the test is run at the end of each loop, and if it is still true the loop goes around again.
- The third argument is used to change the loop variable each time around. Usually, we just increment the variable by 1, but some programmers get clever here. The test is performed just after this operation.

Here's what happens:

[5] I defined a flash routine a couple of pages ago. You have to include that function definition in this sketch.

```
i = 0;  // index variable is initialized
flash(13);
delay(10000);
++i;
//test—if i < 3. it is now 1 so go again
flash(13);
delay(10000);
++i;
//test—if i < 3 now it's 2, one more time
flash(13);
delay(10000);
++i;
//test—if i < 3 now it's 3, we're done
```

Listing 10.

Note that the loop variable counts up from 0. That is very useful in some kinds of code. When we are done it is equal to the number of times the loop executed.

Most programs are chock full of loops. They can get quite complex, with functions such as *break* to end a loop early, or *continue* which means start again without finishing this cycle. There are also other loop controls, such as *while*, which will loop as long as some test is true. (Which explains why while(1) will loop forever.)

At Least One Practical Example

This is obviously not going to be an exhaustive manual for Arduino. You have a whole career ahead of you learning the board. But here's a program that does something useful—it reads the analog inputs and reports their value to the USB connection, where Max could read it via the serial object. What's connected to the inputs is up to you, but I suggest simple pots to test it. The data is reduced to the MIDI value range (0-127) and each number is preceded by a key in the range 128-133. A Max patch to interpret this is shown in the next page. This is not the most efficient or elegant code in the world, but it gets the job done and leaves plenty of time for the important thing—music!

```
/*
  AnalogRead6
 Reads an analog input on 6 pins, prints the result to the serial
monitor
 */
 int pot0 =0;
 int pot1 =1;
 int pot2 =2;
 int pot3 =3;
 int pot4 =4;
 int pot5 =5;

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

```
}

void loop() {
  pot0 = analogRead(A0); // read all pots
  pot1 = analogRead(A1);
  pot2 = analogRead(A2);
  pot3 = analogRead(A3);
  pot4 = analogRead(A4);
  pot5 = analogRead(A5);
    if(Serial.available()){ // be polite, speak when you are
spoken to.
    while (Serial.available()){
      int throwaway = Serial.read();
  }  // clear the buffer
    Serial.write(128);  // use numbers above 127 as keys
    Serial.write(pot0/8); // use MIDI style values
    Serial.write(129);
    Serial.write(pot1/8);
    Serial.write(130);
    Serial.write(pot2/8);
    Serial.write(131);
    Serial.write(pot3/8);
    Serial.write(132);
    Serial.write(pot4/8);
    Serial.write(133);
    Serial.write(pot5/8);
    }
  delay(20);  // fifty times a second is fast enough
}
```

Listing 11.

The only thing tricky about this code is it does not send data unless it has received a message. If it sent constantly, the receiving program could be swamped—in fact, it can crash a computer. This slows things a bit, but even at 9600 baud it only takes 12 ms to send this data. We can go 10 times faster if we want to.

I could make this code easier to read by using arrays, but it wouldn't run any faster or more accurately.

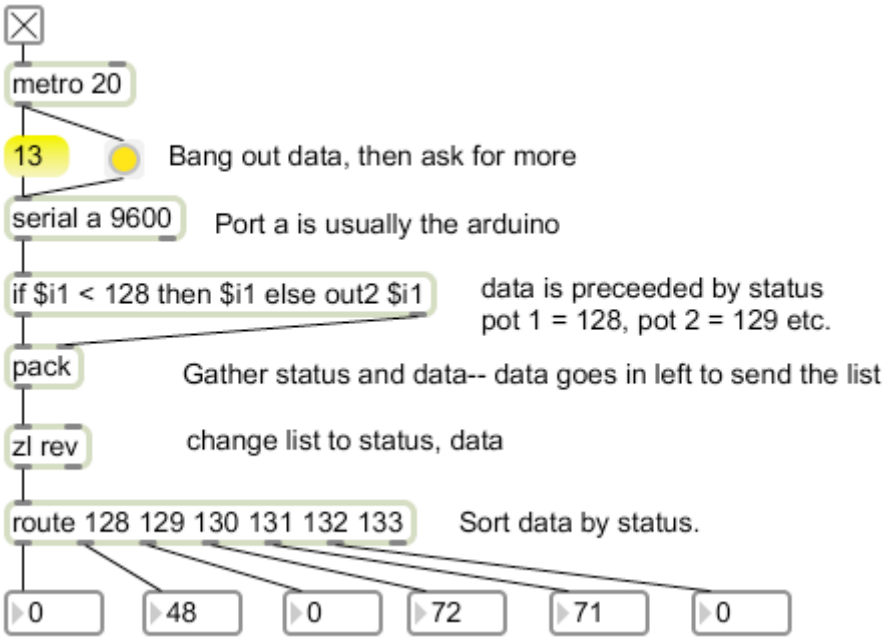

Figure 2

Figure 2 shows a Max patch that will read the data from the arduino. This is what happens:

- Metro bangs serial object
- Serial object sends all data received since last bang (one byte at a time).
- If statement sends status out right and data out left.
- Pack makes a list of data, status
- Zl rev turns this around
- Route shows the data where we want it.
- Metro bang is not complete- a one byte message is sent to arduino, which will respond by sending new data.