# MIDI
# For TheArduino

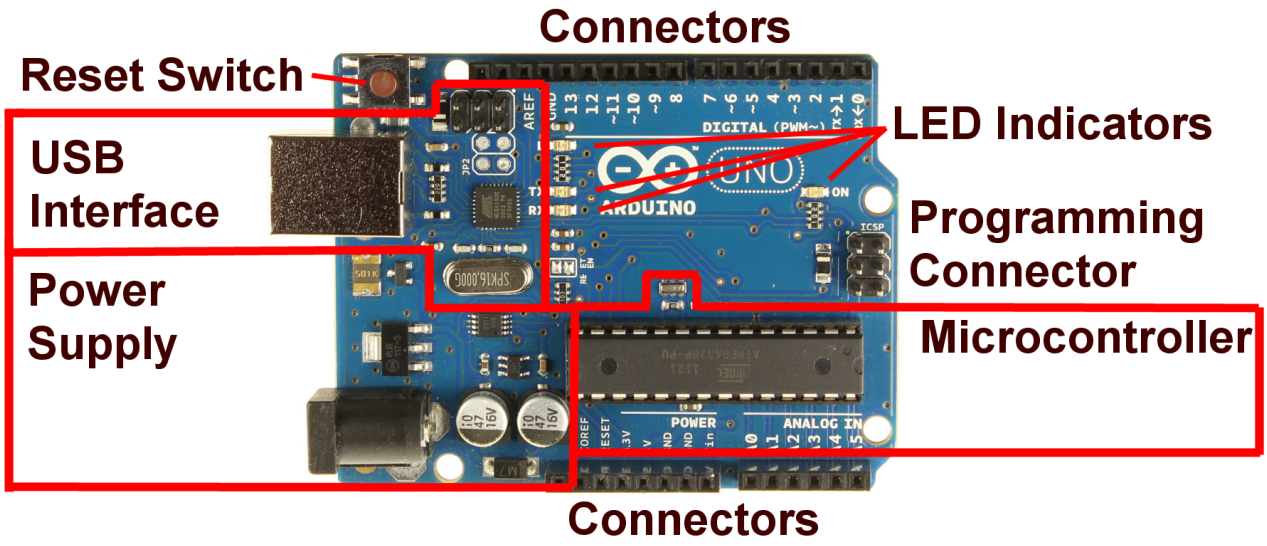
# The Arduino

According to the Arduino web site *"Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments"*

In essence the Arduino is a miniature computer. You can load your own programs onto it, and they can continue to run independently of an attached computer. They are stored even with the power off.

The Arduino has easily accessible connectors that can be wired directly to switches, lights, sensors and so on. It is compact and can run off batteries so it is perfect for building robots, interactive art installations, synthesisers, door entry systems, almost anything that needs a simple computer program to interact with the outside world.

The diagram below shows the main parts of a typical Arduino board. The "miniature computer" (microcontroller chip) is actually just a small part of whats on the board – it's the black rectangular thing with metal legs in the lower right. Everything else on the board is just there to make the microcontroller easier for you to talk to and work with.




The Arduino is based around a specific microcontroller called the Atmel Atmega328. There are many types of microcontroller chips and there is nothing particularly special about the Atmega – it just happens to be a decent mid-range chip that the designers of the Arduino chose to use. There are literally thousands of other types of microcontroller available and you'll find them in huge number of of household appliances and devices. Even your toaster is likely to contain a microcontroller!

Arduino is an open-source project, which means anyone can build and sell "clone" boards. In fact it is very easy to make your own one from scratch. The development tools are free and there is a large global community of people working with Arduino. All this makes it a very good choice for getting started with playing with microcontrollers, embedded programming (meaning writing computer programs that run inside other devices) and as a way into electronics in general.

Its worth mentioning that Arduino is not the only educational microcontroller board out there (PicAXE, Basic Stamp, mbed...etc) but is is certainly one of the most accessible and widely used.

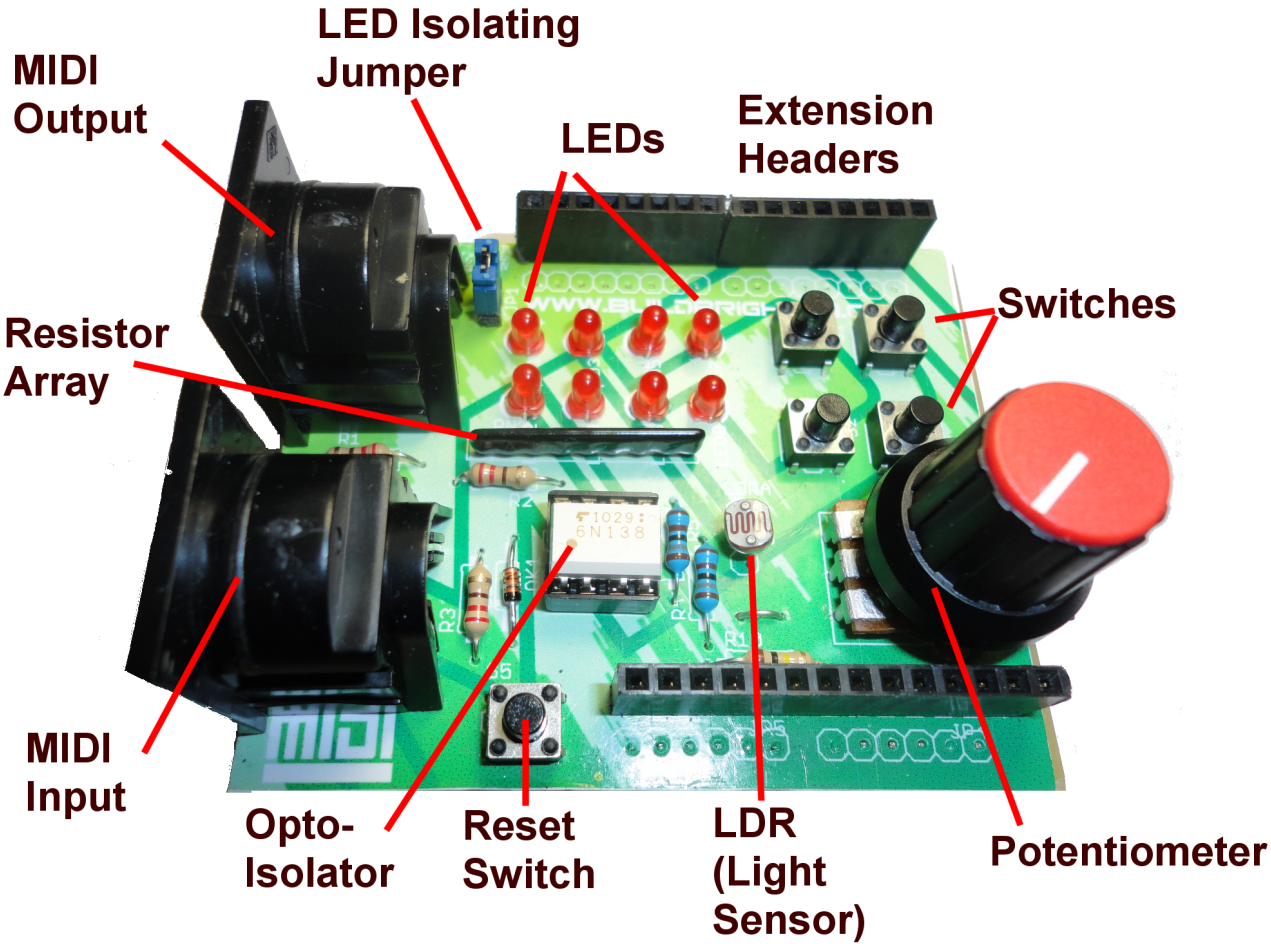
# The MIDI Playground Shield

For this workshop you'll have received a “shield” for the Arduino.

“Shield” is just what the Arduino community calls a circuit board designed to plug in to an Arduino board sit on top of it. Shields add new functions and capabilities to the Arduino – there are many commericallly available shields for different purposes (for example to talk to the internet, or display a picture on a TV set)

BuildBrighton's MIDI Playground shield gives you sockets where you can plug in standard MIDI cables to connect the Arduino with controller keyboards, synthesisers and so on. We've also given you several features which are not specifically MIDI related, but will allow us to create inputs (like pressing buttons) or display outputs (like flashing lights) while we go through the workshop exercises.

Here is a quick tour of your shield.




- The **MIDI Input** and **MIDI Output** sockets allows us to connect the Arduino to MIDI controllers and synthesisers using standard 5-pin MIDI cables.

- The **Opto-isolator** allows the MIDI input to work without a direct electrical connection to the device that is sending the MIDI signals.

  You dont have to understand how this works for you to use the shield, but just in case you want to know - inside this chip there is a tiny LED placed next to a sensitive light operated

switch, so information can passed across the chip via flashes of light. This is necessary because the MIDI standard says devices should not have an electrical connection between their circuit ground points or it can cause problems for some devices, such as “ground loops” (Many devices will actually will work fine without an isolator)

- The eight **LEDs** are connected directly to output pins on the Arduino, so you can easily switch them on and off from a program. For each LED we use a resistor to limit the current that can flow through it (too much current and we could burn it out). The **Resistor Array** contains 8 resistors in a handy small package. You can see some single resistors on the board (they are stripey things with a metal lead coming from each end).

- The four **switches** at the top right are connected to digital inputs of the Arduino. Your programs can check the switches and act when they are pressed.

- The **LDR** (light-dependent resistor) is a light sensor based on a special kind of resistor that gets lower in resistance as more intense light falls on it. The LDR is connected to an input on the Arduino so we can read it from our programs and control sounds by perhaps shading the LDR with a hand. For example we can make a kind of MIDI light theremin this way.

- The **Potentiometer** (“Pot”) is another type of variable resistance, but this time we control it by turning a knob. Pots (and sometimes rotary encoder switches) form the basis of the knobs on MIDI control surfaces.

- Most Arduino shields have **extension headers**, so that all the connectors of the Arduino board are still accessible. The height of our pot and sockets prevents another shield being plugged in on top of the MIDI Playground, but the headers allow you to wire up add other switches, pots etc by poking wires into them.

  Note: The lower extension header adds an addition +5V and ground pin either side of the analog inputs (+5 is next to analog pin 0 and GND is to the right of analog pin 6). These are useful for adding pots and other voltage divider inputs.

- The **reset switch** “reboots” the Arduino. Most shields contain a duplicate Reset switch because they cover up the one on the Arduino board. Hopefully you won't need to make too much use of it :)

- The LEDs are hard-wired to Arduino digital pins 6 through 13. This might prevent you being able to use those pins as inputs. To get around this we provide an **LED Isolation Jumper**. If you remove this jumper the LEDs are disconnected from ground, so they will not light up, and should have minimal impact on the normal function of the digital pins.

# Basic Exercises

Before we look specifically at MIDI, lets try some simple programming with the features of the shield.

## *Basics1.ino*

This sketch simply blinks the LED on pin 13 of the Arduino. All Arduino boards have an on-board LED hooked up to pin 13. This sketch simply flashes it once per second, over and over.

```
//
// BLINK THE LED ON DIGITAL PIN 13
//

void setup()
{
  // set pin 13 to be a digital output
  pinMode(13, OUTPUT);
}

void loop()
{
  // turn the LED on
  digitalWrite(13,HIGH);

  // wait half a second
  delay(500);

  // turn the LED off again
  digitalWrite(13,LOW);

  // turn the LED on
  delay(500);
}
```

The delay command makes the program wait for a certain number of milliseconds (thousandths of a second). Delay(500) is a half-second delay. Without a delay the LED would flash so fast it would appear to be lit continuously.

Note how we say HIGH to switch an output pin “on” and LOW to switch it “off”. HIGH and LOW refer to the voltage levels +5V and 0V. We like to use these terms because they are more accurate.. the pin is never “off” (which implies it is disconnected from any voltage). Rather it is either connected to either the HIGH supply voltage (5 volts) or the LOW supply voltage (0 volts).

Since our LED is connected between the output pin and the LOW supply voltage, a HIGH output on the pin turns the LED on and a LOW output voltage turns is off again. It is equally possible to connect the LED, in reverse, between the output pin and the HIGH supply voltage such that the LED is switched on when we set the output pin LOW!

## *Basics2.Ino*

Now lets switch all 8 LEDs on and off in sequence. Notice how using the "#define" command lets us replace a pin number like "13" with a more readable name like "LED1PIN"

```
// Define some symbols to use in place of the actual
// pin numbers (this makes our program easier to read)
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

void setup()
{
  // set all the pins connected to the LEDs to work
  // as digital outputs
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);
}

void loop()
{
  // first led on for 0.1 seconds
  digitalWrite(LED1PIN, HIGH);
  delay(100);
  digitalWrite(LED1PIN, LOW);

  // led #2 on for 0.1 seconds
  digitalWrite(LED2PIN, HIGH);
  delay(100);
  digitalWrite(LED2PIN, LOW);

  // led #3 on for 0.1 seconds
  digitalWrite(LED3PIN, HIGH);
  delay(100);
  digitalWrite(LED3PIN, LOW);

  // led #4 on for 0.1 seconds
  digitalWrite(LED4PIN, HIGH);
  delay(100);
  digitalWrite(LED4PIN, LOW);

  // led #5 on for 0.1 seconds
  digitalWrite(LED5PIN, HIGH);
  delay(100);
  digitalWrite(LED5PIN, LOW);

  // led #6 on for 0.1 seconds
  digitalWrite(LED6PIN, HIGH);
  delay(100);
  digitalWrite(LED6PIN, LOW);
```

```
  // led #7 on for 0.1 seconds
  digitalWrite(LED7PIN, HIGH);
  delay(100);
  digitalWrite(LED7PIN, LOW);

  // led #8 on for 0.1 seconds
  digitalWrite(LED8PIN, HIGH);
  delay(100);
  digitalWrite(LED8PIN, LOW);
}
```

## *Basics3.Ino*

Now we'll look at reading inputs. As well as the 8 LEDs on digital pins 6 through 13, the shield has switches connected to pins 2 through 5.

Each time the loop function runs, it checks the position of each switch in turn, and if the switch is pressed it turns on two LEDs. If the switch is not pressed, it turns those two LEDs off.

```
// define names for the pins
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

#define SWITCHAPIN  5
#define SWITCHBPIN  2
#define SWITCHCPIN  4
#define SWITCHDPIN  3

void setup()
{

  // these pins, for the LEDs, are outputs
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);

  // but these pins, for the switches, are inputs
  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);
  pinMode(SWITCHCPIN, INPUT);
  pinMode(SWITCHDPIN, INPUT);

  // these lines turn on the "internal pull-ups" for the switch inputs
  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);
  digitalWrite(SWITCHCPIN, HIGH);
  digitalWrite(SWITCHDPIN, HIGH);

}

void loop()
{
  // if switch A is pressed down right now
  if(digitalRead(SWITCHAPIN) == LOW)
  {
    // turn leds 1+2 on
    digitalWrite(LED1PIN, HIGH);
    digitalWrite(LED2PIN, HIGH);
  }
  else
  {
```

```
    // turn leds 1+2 off
    digitalWrite(LED1PIN, LOW);
    digitalWrite(LED2PIN, LOW);
  }

  // if switch B is pressed down right now
  if(digitalRead(SWITCHBPIN) == LOW)
  {
    // turn leds 3+4 on
    digitalWrite(LED3PIN, HIGH);
    digitalWrite(LED4PIN, HIGH);
  }
  else
  {
    // turn leds 3+4 off
    digitalWrite(LED3PIN, LOW);
    digitalWrite(LED4PIN, LOW);
  }

  // if switch C is pressed down right now
  if(digitalRead(SWITCHCPIN) == LOW)
  {
    // turn leds 5+6 on
    digitalWrite(LED5PIN, HIGH);
    digitalWrite(LED6PIN, HIGH);
  }
  else
  {
    // turn leds 5+6 off
    digitalWrite(LED5PIN, LOW);
    digitalWrite(LED6PIN, LOW);
  }

  // if switch D is pressed down right now
  if(digitalRead(SWITCHDPIN) == LOW)
  {
    // turn leds 7+8 on
    digitalWrite(LED7PIN, HIGH);
    digitalWrite(LED8PIN, HIGH);
  }
  else
  {
    // turn leds 7+8 off
    digitalWrite(LED7PIN, LOW);
    digitalWrite(LED8PIN, LOW);
  }
}
```

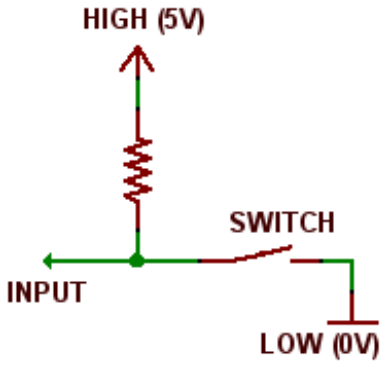
There are a few more things here... you can see that for the switches we set the associated digital pin to be an INPUT using the pinMode command. This means we can use the digitalRead command to check whether the digital input is HIGH or LOW, telling us whether an attached switch is pressed or not.

Remember we are looking for HIGH or LOW inputs, not “on” and “off”. A not-so-obvious thing about digital inputs is that if they are not connected to anything, they do **not** automatically read LOW.

How so? Well these input pins are very sensitive, and if we are reading them while they are connected to nothing, they start picking up stray electrical noise. Suprisingly we're just as likely to get a HIGH reading as a LOW one from an unconnected pin and usually we'll see it randomly

flipping between these values. Kind of interesting... but very confusing and one of the most common pitfalls when you start playing adding switches to your Arduino!
Usually we avoid this “floating” behaviour like the plague and we do this with a simple trick called the “pull up resistor”. Basically we connected our digital pin to the HIGH supply voltage through a high value resistor (usually at least 10,000 ohms). This means that the pin consistently reads a HIGH input and it no longer affected by noise.

Now we put our switch between the pin and the LOW supply voltage. When the switch is pressed, it connects the digital input to LOW and easily overcomes the HIGH voltage (since the high value resistor makes sure the HIGH level it is very weaky applied)




By simply adding the resistor we cure the “floating” input problems. We just need to remember that when our switch is pressed, the input will read LOW (so kind of working in reverse)

The pull up resistor trick is so useful that the Atmega chip used in the Arduino actually has them built in, so we don't need to use external resistances to achieve this. Instead we set the value of an INPUT pin to HIGH, which is a completely different thing to setting an OUTPUT pin to HIGH!

```
pinMode(SWITCHAPIN, INPUT);
digitalWrite(SWITCHAPIN, HIGH);
```

If you think a bit about this you might wonder why “pull down” resistors are not used. The principle would be exactly the same but the value from the switch would be HIGH when pressed, so a bit more user-friendly, right?

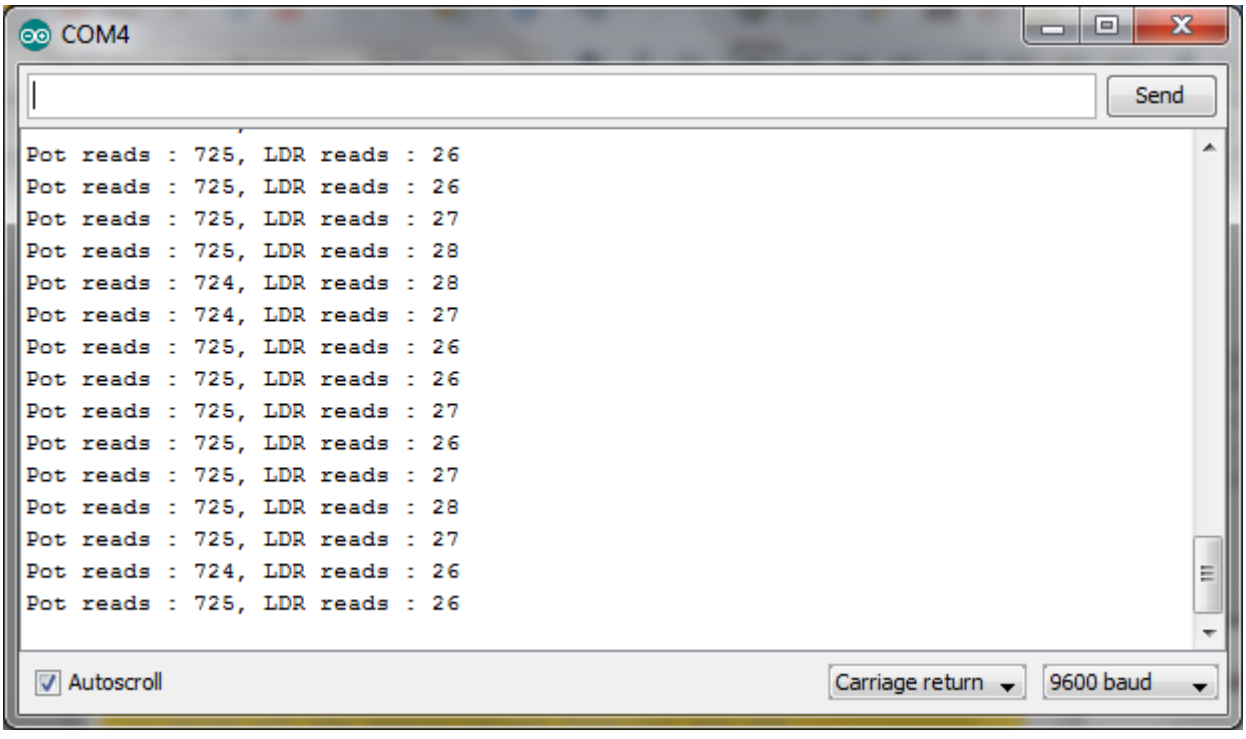
There is a technical reason to choose pull-up rather than pull-down resistors. Basically a small current would flow out of the input pin to LOW through the pull-down resistor, which increases the power draw of the circuit by a very small amount. Its a small loss, but enough to make pull-ups the natural choice all else being equal. Pull down resistors are still used in some situations where a pull-up cannot be used.

## *Basics5.Ino*

This sketch might look small, but it introduces two big concepts. Here we will take a look at analog inputs (those which can read a continuous range of values rather than just HIGH/LOW) and we'll also look at the serial interface, which lets us send information from the Arduino which can be displayed on a computer screen.

```
void setup()
{
  // set up the serial port to 9,600 bits per second
  Serial.begin(9600);
}

void loop()
{
  // write out the analog values from pot and LDR
  Serial.print("Pot reads : ");
  Serial.print(analogRead(0));
  Serial.print(", LDR reads : ");
  Serial.println(analogRead(1));

  delay(100);

}
```

Run the sketch and then in the Arduino IDE, open up the Serial monitor. You should see something like this...




Try rotating the potentiometer knob on the shield, and move the LDR in and out of the light. You should see the values changing (in fact you may find the numbers fluctuate a little bit even when you're not near the board)

Play with the pot, and you should notice the numbers for the pot range between 0 and 1023. The same is true of the LDR, but its output wont reach either extreme.

The numbers you are seeing are the raw values read using the analogRead command. This command reads one of six analog input pins. The number you see is 0 if the analog pin is connected to LOW (0V), or 1023 if the pin is connected to HIGH (5V). In between, the value will refect the fraction of the HIGH voltage which is being read at the pin.

Both the potentiometer and the LDR are variable resistors, not voltage sources, so what clever trick do we use to make them output a voltage between LOW and HIGH? Again the solution is very simple.

Lets start with the potentiometer. A potentiometer has three connections – two of them are to either end of a circular resistive track. The other connection (usually the one in the middle) is to a “wiper” which connects with the track at a position you decide by turning the knob. Lets call the track connections A and B and the wiper W. When the knob is turned completely to the left A and W are connected and the full resistance of the track is between W and B. Conversely when the knob is completely to the right the opposite is true; W is connected to B and at maximum resisitence from A.

Now, if we connect A to LOW and B to HIGH then W will read a value between LOW and HIGH depending on how far it has been turned. Halfway between A and B we'll get half the voltage and therefore half the analog input reading. The pot dials up a voltage between LOW and HIGH – perfect!

However, now we have our power supply shorted across the pot track right? Surely that's bad? Well, as long as the pot track is a high resistance this doesn't matter since minimal current can flow through the track. The pots on your shield are 10,000 ohms and it's common to use pots with track resistences of a million ohms, so it not a problem! However remember this when picking pots for an analog input. A low resistance like 500 ohms would definitely be unsuitable for this application.

So, the pot kind of works out of the box but what about the LDR? It only has 2 connections so can't “divide the voltage” like a pot can right?

Actually, it can - all we need to do is add another (fixed) resistor of a similar resistance to the LDR. On our shield we connect the LDR to a 100,000 ohm resistor  and the pair are connected in series between the HIGH and LOW voltage supplies.

We connect the analog input to the point where the 2 resistors are connected – as the light level changes on the LDR, the resistance of the LDR changes, and the ratio of the LDR to the fixed 100k resisitence changes accordingly. The voltage that we read reflects this ratio. The upshot is that we convert the LDR resistance to a voltage level we can read, however since the LDR never reaches “zero” or “infinite” resistance, the raw input values we get will never quite reach the 0 or 1023 extremes we could hit with the pot.

This sketch also uses the Arduino's “serial” port to write messages from the Arduino that we can are receiving via the USB cable and displaying on the Arduino serial monitor tool. The number 9600 is the rate that information is being sent across the connection, there is nothing very special about the 9600 rate, it just happens to be the default.

Using the serial port like this is very useful for writing out messages from your program to see what is going on (after all, the Arduino has no screen). However as we'll soon see, we send MIDI messages out from the Arduino in exactly the same way as we're sending out these text messages...

## *Basics6.Ino*

Phew, a lot to take in with the last sketch! With this one we'll just be putting together some of the things we've covered up until now to do something fun.

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);
}

void indicate(int value)
{
  digitalWrite(LED1PIN, (value > 128 * 7)? HIGH: LOW);
  digitalWrite(LED2PIN, (value > 128 * 6)? HIGH: LOW);
  digitalWrite(LED3PIN, (value > 128 * 5)? HIGH: LOW);
  digitalWrite(LED4PIN, (value > 128 * 4)? HIGH: LOW);
  digitalWrite(LED5PIN, (value > 128 * 3)? HIGH: LOW);
  digitalWrite(LED6PIN, (value > 128 * 2)? HIGH: LOW);
  digitalWrite(LED7PIN, (value > 128 * 1)? HIGH: LOW);
  digitalWrite(LED8PIN, (value > 128 * 0)? HIGH: LOW);
}
void loop()
{
 indicate(analogRead(0));
}
```

Run this sketch and move the pot. You should see that between 0 and 8 LEDs come on depending on the position of the pot. What we're doing is simply diving the range of analog values from the pot (0-1023) into 9 divisions and lighting the appropriate number of LEDs for the value we read.

You might notice this program is a bit more compact than Basics2, even though it does more. We can do this by defining our own “function” (called “indicate”) in which we use a little trick so save a few lines of code. For example the line

```
digitalWrite(LED1PIN, (value > 128 * 7)? HIGH: LOW);
```

Is exactly equivalent to

```
if(value > 128 * 7)
 digitalWrite(LED1PIN, HIGH);
else
 digitalWrite(LED1PIN, LOW);
```

but saves us a several lines of code and as a result is more readable despite being a bit more cryptic.

## *Basics7.Ino*

Our final “basics” sketch simply repeats Basics6 but using the LDR to control the level of the LEDS instead of the pot. Note that we multiply the LDR reading by 8 to give us a reasonable chance of lighting all the LEDs.

Since the LDR lowers in resistance with a greater light level, the number of LEDs will reduce with higher light levels.

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);
}

void indicate(int value)
{
  digitalWrite(LED1PIN, (value > 128 * 7)? HIGH: LOW);
  digitalWrite(LED2PIN, (value > 128 * 6)? HIGH: LOW);
  digitalWrite(LED3PIN, (value > 128 * 5)? HIGH: LOW);
  digitalWrite(LED4PIN, (value > 128 * 4)? HIGH: LOW);
  digitalWrite(LED5PIN, (value > 128 * 3)? HIGH: LOW);
  digitalWrite(LED6PIN, (value > 128 * 2)? HIGH: LOW);
  digitalWrite(LED7PIN, (value > 128 * 1)? HIGH: LOW);
  digitalWrite(LED8PIN, (value > 128 * 0)? HIGH: LOW);
}
void loop()
{
 indicate(analogRead(1) * 8);
}
```

# Introducing MIDI

MIDI (Musical Instrument Digital Interface) is a way for electronic musical instruments, and controllers such as keyboards, to communicate with each other. It has been around since the 1980's and its still going strong (so it must be good, right?). Well, despite its limitations and frustrations, MIDI has the great advantage of being exteremely simple to implement in a device, as well as being supported by pretty much everything. MIDI really is one of the ways to have the most fun with your Arduino with the least effort.

MIDI works by sending messages down a wire in precisely the same way that we sent text messages over serial in the previous exercise. The trick is not in the wiring, but in the structure of the messages that we send over it.

Each MIDI message is a sequence of characters. Most MIDI messages are made up of 3 characters. You can think of them as being a bit like very short SMS text messages which might say something like

“Start Playing A Note,  Channel Is Number 1,  Note Is Middle C, Volume Is Medium”

MIDI can say all this with 3 characters (or “bytes”, if you prefer). Remember that each character has a numeric value as well as a symbol. Since MIDI messages are not intended to be easy for humans to read, we'll be talking in terms of the numeric byte values rather than the character symbols (So instead of “A” we'll say 65 – if this isn't making sense right now don't worry, it should soon)

The three character values used for the example message above would be

**144, 60, 63**

MIDI gives each musical note a numeric value between 0 and 127, and 60 happens to be the number used for Middle C. Volume, or “velocity” is a number between 0 (silent) and 127 (loudest) so 63 is in the middle of this range. 144 is the code for “Channel 1 note on”.

It is usually easier to see the structure of MIDI data if we use “hexadecimal” instead of normal decimal numbers. Hexadecimal is base 16 and is commonly used in computing.

When we count in hexdecimal we start 0, 1, 2, 3 …. but when we get to ten, we start using letters; A, B, C …. all the way to F (fifteen). Sixteen is 10, Seventeen is 11 and so we continue up to 1F (Thirty-one) then roll over to 20 (Thirty-two)

In hexadecimal every number from zero (00) to two-hundred and fifty-five (FF) can be written using a maximum of two digits. To make it clear that we have a hexadecimal number rather than decimal (for example “21” meaning “thirty-three” and not “twenty-one”) it is common to use a prefix like “#” or “0x” and to insert a leading zero for single digit numbers.

Since the Arduino environment uses the 0x convention well use it here. So hexadecimal counting would go as follows: 0x00, 0x01, 0x02, 0x03 …. 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12 … 0x1f, 0x20 … 0xfe, 0xff.

Lets look at our MIDI message again but this time in hexadecimal.

144, 60, 63 becomes **0x90, 0x3c, 0x3f**

There is a point to this.... the hexadecimal notation now helps us see the structure of the first character. Remember 144 means “note on channel 1”. When this is written as 0x90, the first digit “9” means “note on” and the second digit “0” means “Channel 1”.

In this way we can treat a single character value as if it was two values from 0-F (zero through fifteen) packed together with different meanings. This is a common computer programming trick, and these half-byte values even have the special name “nybble” (geddit?)

MIDI has sixteen channels numbered 0 through F (zero through fifteen) although they are conventionally and confusingly referred to as channels one through sixteen (so the channel numbered 9 in the data is actually called “MIDI channel ten” :o)

Hexadecimal helps us easily see that 0x90 is “note on channel 1” and 0x93 is “note on channel 4”. Other messages are similar, for example 0x8f means “note off channel 16” and 0xb2 means “controller position for channel 3” (We'll discuss these other messages later). The structure of these values is much easier to see than the equivalent decimal numbers 144, 147, 143 and 178 respectively. When you've work with MIDI for a while you'll get to recognise the hexadecimal values for message types much more easily that you would the decimal values.

Before we move on to look at specific MIDI messages, it is useful to know something else about MIDI. Perhaps you were wondering why the characters used for note number and note velocity are restricted to the value range 0-127 when the full range of possible values is 0-255. Or maybe you weren't but you are now...

The reason is a clever trick used by the designers of MIDI. Any character value which has a decimal value greater than 127 is defined to be a “command”. In hexadecimal this would mean any value 0x8_, 0x9_, 0xA_, 0xB_, 0xC_, 0xD_, 0xE_, 0xF_ (For the binary speakers amongst you it means any value with bit 7 set)

So, going back to our example message, 0x90, 0x3c, 0x3f

The “Channel 1 note on” character, 0x90, is a command, but the other characters are just data. Since they are less than 0x80. So, data values can have any value 0x00 through 0x7f. In decimal this is our 0-127 range.

So why is this a clever trick?

Well firstly it enables a device listening to MIDI to get synchronised with the sender very easily. Imagine connecting into a MIDI stream which is already up and running. Without this easy distinction you would not easily know how to handle the numbers you are seeing, since you would not know which numbers represented commands and which represented data. MIDI makes this very easy and does it with no additional overhead (like special message markers). This was very important when MIDI first came out, since data communications were slow at the time and every possible ounce of efficiency was squeezed out of them (MIDI is still slow, so it remains important!)

Another clever trick is that it is possible to insert single byte commands (without any data) inside other commands when it is important to send them without any delay. One such command is a MIDI clock “tick” which keeps MIDI devices in time with each other. This has the code 0xF8. Perhaps a clock tick occurs while we're sending our note on message from a sequencer which is also the MIDI synchronisation source (keeping other instruments in time with it). The device receiving

the message might see

0x90, 0x3c, 0xf8, 0x3f

The note on message is messed up! but its not a problem since we know that 0x90 is a note on command with 2 data bytes (note number and velocity) following it, and since 0xf8 is out of the valid range for data we know it is not the note velocity, so the note velocity must come after it.

MIDI commands which can break into other messages are called “realtime” commands and we'll look at them later.

Another trick used by MIDI is “running status” which saves space on the line. Say we send a sequence of Channel 1 Note On commands for notes middle C, D and E, we might send

0x90, 0x3c, 0x3f, 0x90, 0x3e, 0x3f, 0x90, 0x40, 0x3f

MIDI lets us compress this down to

0x90, 0x3c, 0x3f, 0x3e, 0x3f, 0x40, 0x3f

We only sent the command once and the receiver, having got the note and velocity values for the first command, sees more data bytes and assumes they are for the same command type. The command status has been left “running”. This might only save us one byte for each message, but remember MIDI is slow and every little helps.

It is common practice to send “note off” as a “note on” with zero velocity instead of the specifial MIDI note off command (0x8_). For example we could silence a middle C playing on channel 2 by sending

0x81, 0x3c, 0x00

But we could instead send

0x91, 0x3c, 0x00

This is a “note on” with zero velocity. The advantage? Well it lets us use the running status to save space when we send a batch of note on/off messages (say on a chord change). For example stop playing a C and play a D..

0x81, 0x3c, 0x00, 0x91, 0x3e, 0x7f

or alternatively

0x91, 0x3c, 0x00, 0x3e, 0x7f

One byte saved.. high fives all round!

## Sending MIDI Notes

Lets actually send some MIDI data from our Arduino and get it to play something. We'll be sending notes on MIDI channel 1, so you need a MIDI device to receive and play the notes, which should listen on Channel 1 or Omni (any channel) mode.

If you don't have a hardware sound module or synthesiser available, you can use a “soft synth” program or plugin running on a PC or Mac, and a USB-MIDI lead to connect with the Arduino.

A quick word of warning before we start. Our Arduino MIDI sketches uses the Arduino's serial port to send MIDI. The Arduino is actually programmed (when you “Upload” from the IDE) by sending the program over this same serial connection. This can cause a problem when a synthesiser receives this Arduino program and tries to treat it as MIDI data, and can get in a mess and need to be reset. Alternatively you can disconnect the MIDI Out lead from the shield when you are uploading a new program to the Arduino.

## *SendNotes1.Ino*

Here is a minimal sketch to send a MIDI note. Hopefully some parts of this are now looking familiar to you and you should be able to see to some degree how the sketch works.

```
void setup()
{
  Serial.begin(31250);
}

void loop()
{
  Serial.write((byte)0x90);
  Serial.write((byte)0x30);
  Serial.write((byte)0x7f);

  delay(200);

  Serial.write((byte)0x90);
  Serial.write((byte)0x30);
  Serial.write((byte)0x00);

  delay(200);

}
```

Lets look at the important parts here:

```
Serial.begin(31250);
```

Just like the sketch where we first used the serial port, we need to use the begin command to tell the serial port the data rate that we want to use. 31250 here is a “magic” number, MIDI always uses this data rate, so you will need to too. If you have the wrong data rate, your MIDI messages won't be understood.

```
Serial.write((byte)0x90);
Serial.write((byte)0x30);
Serial.write((byte)0x7f);
```

These three commands are each sending a single character to the serial port. The characters we are sending are

0x90, 0x30, 0x7f

Once you've been looking at enough of these messages you'll instantly recognise this as a “note on” message on MIDI channel 1, with a note value of 0x30 (decimal 48, the “C” an octave below middle C) and a velocity of 0x7f (decimal 127, which is “full velocity”)

When this message is received by your sound module or soft-synth, it should start playing that “C” note.

The note will continue to play until we tell it to stop. MIDI messages do not contain any “note duration” but rather the sender needs to explicitly stop the note by sending another message after the appropriate period of time.

We have a 200 millisecond (one fifth of a second) delay before we send another message to silence the note.

```
Serial.write((byte)0x90);
Serial.write((byte)0x30);
Serial.write((byte)0x00);
```

This message is a “note on” with zero velocity. We could actually have sent the following message instead

```
Serial.write((byte)0x80);
Serial.write((byte)0x30);
Serial.write((byte)0x00);
```

This differs in the first character; 0x80 means “note off, channel 1”. As I said before, it is common practice in MIDI implementations to use zero velocity note on instead of the proper note off message (because using a single message type lets us use the “running status” to reduce the size of our messages)

What might we change about the sketch if we wanted to use “running status” to stop the note ringing using a MIDI message that is just 2 bytes long? (Don't read the next paragraph if you want to work this out yourself)

The answer is that we can remove the first byte from the note off message, so we just send the bytes 0x30 and 0x00. The 0x90 status will carry over (or “run”) from the previous note on message. Give it a try sending this modified message to your synth. Hopefully you're starting to see why most MIDI implementations don't bother with the 0x80 note off message, and we'll forget about it from hereon in.

## *SendNotes2.Ino*

Lets make this a bit more interactive by using our digital input skills to trigger a MIDI note when a button is pressed

```
// SEND NOTE WHEN A BUTTON IS PRESSED
#define LED1PIN  13
#define SWITCHAPIN  5

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(SWITCHAPIN, INPUT);
  digitalWrite(SWITCHAPIN, HIGH);

  Serial.begin(31250);
}

void sendNote(byte note, byte velocity)
{
  Serial.write((byte)0x90);
  Serial.write(note);
  Serial.write(velocity);
}

int wasPressed = 0;

void loop()
{
  // is switch A currently held down?
  if(digitalRead(SWITCHAPIN) == LOW)
  {
    // has it just been pressed?
    if(wasPressed == 0)
    {
      sendNote(48,127);
      digitalWrite(LED1PIN, HIGH);
      wasPressed = 1;
    }
  }
  else
  {
    // has it just been released?
    if(wasPressed == 1)
    {
      sendNote(48,0);
      digitalWrite(LED1PIN, LOW);
      wasPressed = 0;
    }
  }
}
```

Note how we use a variable (“wasPressed”) to remember whether the button was pressed last time we checked it. It is important that we can identify “changes” in the button state and only send our MIDI  note on or note off message when the state changes.

# The MIDI Scale

We have seen how MIDI represents each note of the musical scale by a number between 0 and 127. Lets have a quick look at what this gives us. The following diagram shows all the available MIDI notes.

| MIDI | | Octave | MIDI | | Octave | MIDI | | Octave |
|---|---|---|---|---|---|---|---|---|
| 0 | C | -1 | 45 | A | 2 | 90 | F# | 6 |
| 1 | C# | -1 | 46 | A# | 2 | 91 | G | 6 |
| 2 | D | -1 | 47 | B | 2 | 92 | G# | 6 |
| 3 | D# | -1 | 48 | C | 3 | 93 | A | 6 |
| 4 | E | -1 | 49 | C# | 3 | 94 | A# | 6 |
| 5 | F | -1 | 50 | D | 3 | 95 | B | 6 |
| 6 | F# | -1 | 51 | D# | 3 | 96 | C | 7 |
| 7 | G | -1 | 52 | E | 3 | 97 | C# | 7 |
| 8 | G# | -1 | 53 | F | 3 | 98 | D | 7 |
| 9 | A | -1 | 54 | F# | 3 | 99 | D# | 7 |
| 10 | A# | -1 | 55 | G | 3 | 100 | E | 7 |
| 11 | B | -1 | 56 | G# | 3 | 101 | F | 7 |
| 12 | C | 0 | 57 | A | 3 | 102 | F# | 7 |
| 13 | C# | 0 | 58 | A# | 3 | 103 | G | 7 |
| 14 | D | 0 | 59 | B | 3 | 104 | G# | 7 |
| 15 | D# | 0 | 60 | C | 4 | 105 | A | 7 |
| 16 | E | 0 | 61 | C# | 4 | 106 | A# | 7 |
| 17 | F | 0 | 62 | D | 4 | 107 | B | 7 |
| 18 | F# | 0 | 63 | D# | 4 | 108 | C | 8 |
| 19 | G | 0 | 64 | E | 4 | 109 | C# | 8 |
| 20 | G# | 0 | 65 | F | 4 | 110 | D | 8 |
| 21 | A | 0 | 66 | F# | 4 | 111 | D# | 8 |
| 22 | A# | 0 | 67 | G | 4 | 112 | E | 8 |
| 23 | B | 0 | 68 | G# | 4 | 113 | F | 8 |
| 24 | C | 1 | 69 | A | 4 | 114 | F# | 8 |
| 25 | C# | 1 | 70 | A# | 4 | 115 | G | 8 |
| 26 | D | 1 | 71 | B | 4 | 116 | G# | 8 |
| 27 | D# | 1 | 72 | C | 5 | 117 | A | 8 |
| 28 | E | 1 | 73 | C# | 5 | 118 | A# | 8 |
| 29 | F | 1 | 74 | D | 5 | 119 | B | 8 |
| 30 | F# | 1 | 75 | D# | 5 | 120 | C | 9 |
| 31 | G | 1 | 76 | E | 5 | 121 | C# | 9 |
| 32 | G# | 1 | 77 | F | 5 | 122 | D | 9 |
| 33 | A | 1 | 78 | F# | 5 | 123 | D# | 9 |
| 34 | A# | 1 | 79 | G | 5 | 124 | E | 9 |
| 35 | B | 1 | 80 | G# | 5 | 125 | F | 9 |
| 36 | C | 2 | 81 | A | 5 | 126 | F# | 9 |
| 37 | C# | 2 | 82 | A# | 5 | 127 | G | 9 |
| 38 | D | 2 | 83 | B | 5 | | | |
| 39 | D# | 2 | 84 | C | 6 | | | |
| 40 | E | 2 | 85 | C# | 6 | | | |
| 41 | F | 2 | 86 | D | 6 | | | |
| 42 | F# | 2 | 87 | D# | 6 | | | |
| 43 | G | 2 | 88 | E | 6 | | | |
| 44 | G# | 2 | 89 | F | 6 | | | |

We can see this scale covers almost nine octaves, which is more than enough for most purposes (especially given that most MIDI instruments can be transposed up and down by a number octaves).

Middle C is note 60 (C4) and we can see that there are 12 MIDI notes per octave, so adding or subtracting 12 from a MIDI value transposes us by an octave.

The “granularity” of the MIDI scale is the semitone, just like a piano keyboard. MIDI is not designed to accommodate microtonal scales – although there are plenty of workarounds such that this does not need to be a limit to creativity.

## *SendNotes3.Ino*

This sketch is extension to the previous one - a one note MIDI keyboard isn't much use.. but four....

Lets extend the last sketch to make a four note MIDI input device using the first 4 notes of a C major scale and all four of our buttons.

This is much better – four times better in fact - you can even play the first bar of “three blind mice” on it!

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7

#define SWITCHAPIN  5
#define SWITCHBPIN  2
#define SWITCHCPIN  4
#define SWITCHDPIN  3

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);
  pinMode(SWITCHCPIN, INPUT);
  pinMode(SWITCHDPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);
  digitalWrite(SWITCHCPIN, HIGH);
  digitalWrite(SWITCHDPIN, HIGH);

  Serial.begin(31250);
}

void sendNote(byte note, byte velocity)
{
  Serial.write((byte)0x90);
  Serial.write(note);
  Serial.write(velocity);
}


byte wasPressed[4];

void handleButton(byte which, byte inputPin, byte outputPin, byte note)
{
  if(digitalRead(inputPin) == LOW)
  {
    digitalWrite(outputPin, HIGH);
    if(wasPressed[which] == 0)
    {
      sendNote(note,127);
      wasPressed[which] = 1;
    }
  }
```

```
  else
  {
    digitalWrite(outputPin, LOW);
    if(wasPressed[which] == 1)
    {
      sendNote(note,0);
      wasPressed[which] = 0;
    }
  }

}

void loop()
{
  handleButton(0, SWITCHAPIN, LED1PIN, 48);
  handleButton(1, SWITCHBPIN, LED2PIN, 50);
  handleButton(2, SWITCHCPIN, LED3PIN, 52);
  handleButton(3, SWITCHDPIN, LED4PIN, 53);
}
```

This sketch shows a bit of tidying up of the code using a function to handle the buttons. This is far preferable to a copy/paste job, since the resulting code is more compact and easier to make changes to.

## *SendNotes4.Ino*

This next sketch is a very small addition to the last one... it demonstrates how to add control over note velocity using the pot.

Velocity is important for adding dynamics to MIDI playing. MIDI sound sources might change not only the volume, but the whole character of a sound, based on the MIDI note velocity. Most MIDI keyboards have keys that actually detected how hard they are pressed, and transmit this velocity information over MIDI. This is essential to the “feel” of a velocity-sensitive instrument sound such as a piano.

We don't have velocity-sensitive switches, but now we have the next best thing! Try playing notes while changing the pot position between notes....

```
//
// SET VELOCITY USING POT
//

#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7

#define SWITCHAPIN  5
#define SWITCHBPIN  2
#define SWITCHCPIN  4
#define SWITCHDPIN  3

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);
  pinMode(SWITCHCPIN, INPUT);
  pinMode(SWITCHDPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);
  digitalWrite(SWITCHCPIN, HIGH);
  digitalWrite(SWITCHDPIN, HIGH);

  Serial.begin(31250);
}

void sendNote(byte note, byte velocity)
{
  Serial.write((byte)0x90);
  Serial.write(note);
  Serial.write(velocity);
}


byte wasPressed[4];

void handleButton(byte which, byte inputPin, byte outputPin, byte note)
{
  if(digitalRead(inputPin) == LOW)
```

```
  {
    digitalWrite(outputPin, HIGH);
    if(wasPressed[which] == 0)
    {
      sendNote(note,analogRead(0) / 8);
      wasPressed[which] = 1;
    }
  }
  else
  {
    digitalWrite(outputPin, LOW);
    if(wasPressed[which] == 1)
    {
      sendNote(note,0);
      wasPressed[which] = 0;
    }
  }

}

void loop()
{
  handleButton(0, SWITCHAPIN, LED1PIN, 48);
  handleButton(1, SWITCHBPIN, LED2PIN, 49);
  handleButton(2, SWITCHCPIN, LED3PIN, 50);
  handleButton(3, SWITCHDPIN, LED4PIN, 51);
}
```

## *SendNotes5.Ino*

Now we are comfortable sending notes, lets do something a bit more fun..

This sketch is an example of generating a stream of notes programmatically without any interaction. In essence it is a very simple pattern sequencer. Press button B to start and A to stop the pattern

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

#define SWITCHAPIN  5
#define SWITCHBPIN  2

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);

  Serial.begin(31250);
}

void sendNote(byte note, byte velocity)
{
  Serial.write((byte)0x90);
  Serial.write(note);
  Serial.write(velocity);
}

byte notes[8] = {
  48,
  48,
  47,
  48,
  48,
  48,
  51,
  48
};


byte playIndex = 0;
byte lastNote = 0;
byte isRunning = 0;
```

```
unsigned long nextNoteTime = 0;

void loop()
{
  if(isRunning)
  {
    unsigned long thisTime = millis();
    if(thisTime > nextNoteTime)
    {
      if(lastNote > 0)
        sendNote(lastNote, 0);

      byte thisNote = notes[playIndex];
      sendNote(thisNote, 127);
      lastNote = thisNote;
      nextNoteTime = thisTime + 200;

      digitalWrite(LED1PIN, (playIndex == 0) ? HIGH: LOW);
      digitalWrite(LED2PIN, (playIndex == 1) ? HIGH: LOW);
      digitalWrite(LED3PIN, (playIndex == 2) ? HIGH: LOW);
      digitalWrite(LED4PIN, (playIndex == 3) ? HIGH: LOW);
      digitalWrite(LED5PIN, (playIndex == 4) ? HIGH: LOW);
      digitalWrite(LED6PIN, (playIndex == 5) ? HIGH: LOW);
      digitalWrite(LED7PIN, (playIndex == 6) ? HIGH: LOW);
      digitalWrite(LED8PIN, (playIndex == 7) ? HIGH: LOW);

      playIndex = playIndex + 1;
      if(playIndex >= 8)
        playIndex = 0;
    }
  }

  if(digitalRead(SWITCHBPIN) == LOW)
  {
    isRunning= 1;
  }
  else if(digitalRead(SWITCHAPIN) == LOW && isRunning == 1)
  {
    isRunning = 0;
    if(lastNote > 0)
      sendNote(lastNote, 0);
  }
}
```

This sketch introduces an important concept for MIDI programming – that of timing. Our sequencer needs to be able to play notes at a constant rate while it is running, but it also need to remain responsive to the stop/start buttons being pressed (so we dont want to use the delay command, since this would stop the program reading buttons or doing anything else while it is waiting)

Here we use the millis function to fetch the current time (a count of milliseconds since the Arduino was powered on). We can use this for simple timing, withough having to have our program “hang” at a delay.

The code waits 200 milliseconds between each step, but at the same time it is able to respond to buttons being pressed.

The sequence of 8 notes is held in the notes[] array. Have a play changing these note values.

# MIDI Continuous Controllers

Up until now we've just seen how MIDI is used to transmit note information between a controller and a sound-making device. This is obviously one of MIDI's most important abilities, but MIDI is certainly not a one-trick pony and there is plenty else it can do.

On of MIDI's other fundamental features is the ability to send “Continuous Controllers”, which are often abbreviated to “CCs”.

They are “continuous” because instead of just being “on” or “off”, like notes, each CC can have any of a continous range of values, within the now-familiar range 0-127.

They are “controllers” because they are used to control the device which is being played by MIDI. For example a MIDI synthesiser may have many internal parameters that can be controlled by MIDI and which affect the sound. For example a filter cutoff frequency, or a delay time parameter, might be controllable by MIDI.

The simplest way for device parameters to be controlled via MIDI is by using CC messages. Like MIDI note messages, CC information is sent to a specic MIDI channel. The message is sent to a specific controller number on that channel (between 0 and 127) and sets that controller to a value between 0 and 127.

An example of a CC message is

0xb5, 0x01, 0x20

The first byte has a high nybble of “B”. This defines it as a Continuous Controller message. Just like with the Note messages, the second nybble (“5”) defines the MIDI channel. So here we've got a CC message for channel #6 (remember to add one to get the channel “name”)

0x01 is the controller number, and 0x20 is the new value we're setting that controller to. When a synthesiser receives this message, it will set its internal value of controller #1 on channel #6 to this new value (0x20, or 32 deciman)

MIDI actually defines special meanings for most Ccs. Controller #1 is actually defined to be the “Modulation Wheel Position” (so our example message has the effect of signalling movement of the Mod Wheel to position 32 out of 127 (so just under a quarter of the way from minimum modulation to maximum)

Some standard CC numbers are

- Controller #1 – Modulatiom wheel position
- Controller #7 – Channel volume
- Controller #10 – Channel left-right pan
- Controller #11 – Expression pedal position
- Controller #64 – Sustain pedal position

Most MIDI hardware and software has its own set of CC#s that can be used to control it, often these are remappable. We'll see how to use these shortly, but first lets take a look at an example of how to send CC messages from the Arduino.

## *Controllers1.lno*

This sketch is based on the sequencer example. However, we'll add control over the modulation wheel of our synthesiser, which we will link to the pot. Try moving the pot when the sequencer is playing (Remember to use a program on your synth which has a modulation wheel assignment)

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

#define SWITCHAPIN  5
#define SWITCHBPIN  2

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);

  Serial.begin(31250);
}

void sendNote(byte note, byte velocity)
{
  Serial.write((byte)0x90);
  Serial.write(note);
  Serial.write(velocity);
}

void sendController(byte cc, byte value)
{
  Serial.write((byte)0xB0);
  Serial.write(cc);
  Serial.write(value);
}

byte notes[8] = {
  48,
  48,
  47,
  48,
  48,
  48,
  51,
  48
};
```

```
byte playIndex = 0;
byte lastNote = 0;
byte isRunning = 0;
int lastModWheel = 0;

unsigned long nextNoteTime = 0;

void loop()
{
  if(isRunning)
  {
    unsigned long thisTime = millis();
    if(thisTime > nextNoteTime)
    {
      if(lastNote > 0)
        sendNote(lastNote, 0);

      byte thisNote = notes[playIndex];
      sendNote(thisNote, 127);
      lastNote = thisNote;
      nextNoteTime = thisTime + 200;

      digitalWrite(LED1PIN, (playIndex == 0) ? HIGH: LOW);
      digitalWrite(LED2PIN, (playIndex == 1) ? HIGH: LOW);
      digitalWrite(LED3PIN, (playIndex == 2) ? HIGH: LOW);
      digitalWrite(LED4PIN, (playIndex == 3) ? HIGH: LOW);
      digitalWrite(LED5PIN, (playIndex == 4) ? HIGH: LOW);
      digitalWrite(LED6PIN, (playIndex == 5) ? HIGH: LOW);
      digitalWrite(LED7PIN, (playIndex == 6) ? HIGH: LOW);
      digitalWrite(LED8PIN, (playIndex == 7) ? HIGH: LOW);

      playIndex = playIndex + 1;
      if(playIndex >= 8)
        playIndex = 0;
    }
  }

  if(digitalRead(SWITCHBPIN) == LOW)
  {
    isRunning= 1;
  }
  else if(digitalRead(SWITCHAPIN) == LOW && isRunning == 1)
  {
    isRunning = 0;
    if(lastNote > 0)
      sendNote(lastNote, 0);
  }

  int thisModWheel = analogRead(0)/8;
  if(thisModWheel < lastModWheel - 1 || thisModWheel > lastModWheel + 1)
  {
    sendController(1, thisModWheel) ;
    lastModWheel = thisModWheel;
  }

}
```

# Pitch Bend

The Pitch Bend wheel might seem like a of continuous controller, just like the Mod Wheel. However the designers of MIDI obviously felt that it deserved a special message type. Every pitch bend message actually has **two** numbers from 0-127 to define its value.

Here is a pitch bend message

0xe4, 0x33, 0x44

Once again it is directed to a specific MIDI channel. The first byte defines both the message type ("E" for pitch bend) and the channel ("4" for Channel#5)

The pitch bend value is the 0x33 (51 decimal) and 0x44 bytes (68 decimal), which we treat together as one value, equal to the first number plus 128 * the second.

The actual pitchbend position in the example message is

128 * 68 + 51 = 8755

The maximum possible pitchbend value is 128 * 127 + 127 = 16383 and the lowest value is 0. However pitchbend is usually a two-way thing (i.e. bend in either direction)

MIDI defines the center of the possible pitch bend range as "no bend". This is the value 8192 in decimal, so 8755 is a small bend upwards. The actual pitch bend range, in terms of musical notes, is set up in the synthesiser.

Lets try something with pitch bend...

## *Controllers2.lno*

This sketch adds Pitch Bend, controlled by the LDR

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

#define SWITCHAPIN  5
#define SWITCHBPIN  2

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);

  Serial.begin(31250);
}

void sendNote(byte note, byte velocity)
{
  Serial.write((byte)0x90);
  Serial.write(note);
  Serial.write(velocity);
}

void sendController(byte cc, byte value)
{
  Serial.write((byte)0xB0);
  Serial.write(cc);
  Serial.write(value);
}

void sendPitchBend(int value)
{
  Serial.write((byte)0xE0);
  Serial.write(value % 128);
  Serial.write(value / 128);
}

byte notes[8] = {
  48,
  48,
  47,
  48,
  48,
```

```
  48,
  51,
  48
};


byte playIndex = 0;
byte lastNote = 0;
byte isRunning = 0;
int lastModWheel = 0;
int lastPitchBend = 0;

unsigned long nextNoteTime = 0;

void loop()
{
  if(isRunning)
  {
    unsigned long thisTime = millis();
    if(thisTime > nextNoteTime)
    {
      if(lastNote > 0)
        sendNote(lastNote, 0);

      byte thisNote = notes[playIndex];
      sendNote(thisNote, 127);
      lastNote = thisNote;
      nextNoteTime = thisTime + 200;

      digitalWrite(LED1PIN, (playIndex == 0) ? HIGH: LOW);
      digitalWrite(LED2PIN, (playIndex == 1) ? HIGH: LOW);
      digitalWrite(LED3PIN, (playIndex == 2) ? HIGH: LOW);
      digitalWrite(LED4PIN, (playIndex == 3) ? HIGH: LOW);
      digitalWrite(LED5PIN, (playIndex == 4) ? HIGH: LOW);
      digitalWrite(LED6PIN, (playIndex == 5) ? HIGH: LOW);
      digitalWrite(LED7PIN, (playIndex == 6) ? HIGH: LOW);
      digitalWrite(LED8PIN, (playIndex == 7) ? HIGH: LOW);

      playIndex = playIndex + 1;
      if(playIndex >= 8)
        playIndex = 0;
    }
  }

  if(digitalRead(SWITCHBPIN) == LOW)
  {
    isRunning= 1;
  }
  else if(digitalRead(SWITCHAPIN) == LOW && isRunning == 1)
  {
    isRunning = 0;
    if(lastNote > 0)
      sendNote(lastNote, 0);
  }

  int thisModWheel = analogRead(0)/8;
  if(thisModWheel < lastModWheel - 1 || thisModWheel > lastModWheel + 1)
  {
    sendController(1, thisModWheel) ;
    lastModWheel = thisModWheel;
  }

  int thisPitchBend = analogRead(1)*16;
```

```
  thisPitchBend = constrain(thisPitchBend, 0, 16383);
  if(thisPitchBend < lastPitchBend - 100 || thisPitchBend > lastPitchBend +
100)
  {
    sendPitchBend(thisPitchBend) ;
    lastPitchBend = thisPitchBend;
  }

}
```

# Real World Use Of MDI Continuous Controller Messages

As I said above, most MIDI devices and software will have special CC numbers that can be used to control them. The manufacturers of these devices or programs will usually make available a document called a “MIDI Implementation Chart” which shows which CC#s the device will respond to. For example here is part of the midi implementation documentation for the Dave Smith Mopho Synthesiser (a hardware synth)

| Parameter | CC# |
|---|---|
| Osc 1 Frequency | 20 |
| Osc 1 Freq Fine | 21 |
| Osc 1 Shape | 22 |
| Glide 1 | 23 |
| Osc 2 Frequency | 24 |
| Osc 2 Freq Fine | 25 |
| Osc 2 Shape | 26 |
| Glide 2 | 27 |
| Osc Mix | 28 |
| Noise Level | 29 |
| | |
| Filter Frequency | 102 |
| Resonance | 103 |
| Filter Key Amt | 104 |
| Filter Audio Mod | 105 |
| Filter Env Amt | 106 |
| Filter Env Vel Amt | 107 |

| Parameter | CC# |
|---|---|
| VCA Level | 113 |
| Amp Env Amt | 115 |
| Amp Velocity Amt | 116 |
| Amp Delay | 117 |
| Amp Attack | 118 |
| Amp Decay | 119 |
| Amp Sustain | 75 |
| Amp Release | 76 |
| | |
| Env 3 Destination | 85 |
| Env 3 Amt | 86 |
| Env 3 Velocity Amt | 87 |
| Env 3 Delay | 88 |
| Env 3 Attack | 89 |
| Env 3 Decay | 90 |
| Env 3 Sustain | 77 |
| Env 3 Release | 78 |
| | |
| BPM | 14 |

And here is part of the MIDI implementation chart for Propellerheads Reason (A software synth)

| MIDI Contr. # | MIDI Controller Name | Subtractor | Thor | Malström |
|---|---|---|---|---|
| 0 | *ControlBankSelect* | | | |
| 1 | *ControlModulationWheel* | Mod Wheel | Mod Wheel | Mod Wheel |
| 2 | ControlBreathController | | | |
| 3 | ControlUndefined3 | | | |
| 4 | ControlFootController | | Osc 1+2 Level | |
| 5 | ControlPortamentoTime | Portamento | Portamento | Portamento |
| 6 | *ControlDataEntryMSB* | | | |
| 7 | ControlChannelVolume | Master Level | Master Level | Master Level |
| 8 | ControlBalance | | Mixer 1-2 Balance | |
| 9 | ControlUndefined9 | Amp Env Decay | Amp Env Decay | Oscillator B |
| 10 | ControlPan | | Amplifier Pan | |
| 11 | *ControlExpressionController* | | | |
| 12 | ControlEffectControl1 | Amp Env Sustain | Delay Dry/wet | Oscillator B |
| 13 | ControlEffectControl2 | | Chorus Dry/Wet | |
| 14 | ControlUndefined14 | Filter Env Attack | Filter Env Attack | Filter Env Att |
| 15 | ControlUndefined15 | Filter Env Decay | Filter Env Decay | Filter Env De |
| 16 | ControlGeneralPurpose1 | Filter Env Sustain | Filter Env Sustain | Filter Env Su |
| 17 | ControlGeneralPurpose2 | Filter Env Release | Filter Env Release | Filter Env Re |
| 18 | ControlGeneralPurpose3 | Filter Env Amount | Delay Time | Filter Env Am |
| 19 | ControlGeneralPurpose4 | Filter Env Invert | Osc 3 Level | Filter Env Inv |
| 20 | ControlUndefined20 | Osc1 Wave | Osc1 Mod Amount | |
| 21 | ControlUndefined21 | Osc1 Octave | Osc1 Octave | Oscillator B |
| 22 | ControlUndefined22 | Osc1 Semitone | Osc1 Semitone | Oscillator B |
| 23 | ControlUndefined23 | Osc1 Fine Tune | Osc1 Fine Tune | Oscillator B |
| 24 | ControlUndefined24 | | Osc1 Type | |
| 25 | ControlUndefined25 | Osc1 Kbd Track | Delay Feedback | Modulator A |
| 26 | ControlUndefined26 | LFO1 Rate | LFO1 Rate | Modulator A |
| 27 | ControlUndefined27 | LFO1 Amount | LFO1 Delay | Modulator A |
| 28 | ControlUndefined28 | LFO1 Wave | LFO1 Wave | Modulator A |
| 29 | ControlUndefined29 | LFO1 Dest | LFO1 Kbd track | Modulator A |

## *Controllers3.Ino*

Armed with a MIDI implemenation chart, lets turn the MIDI Playground shield into a MIDI control surface! This example connects the pot and 4 switches to a rotary controller and 4 switches on the Combinator module.

**Reason 6 MIDI Controller Chart**

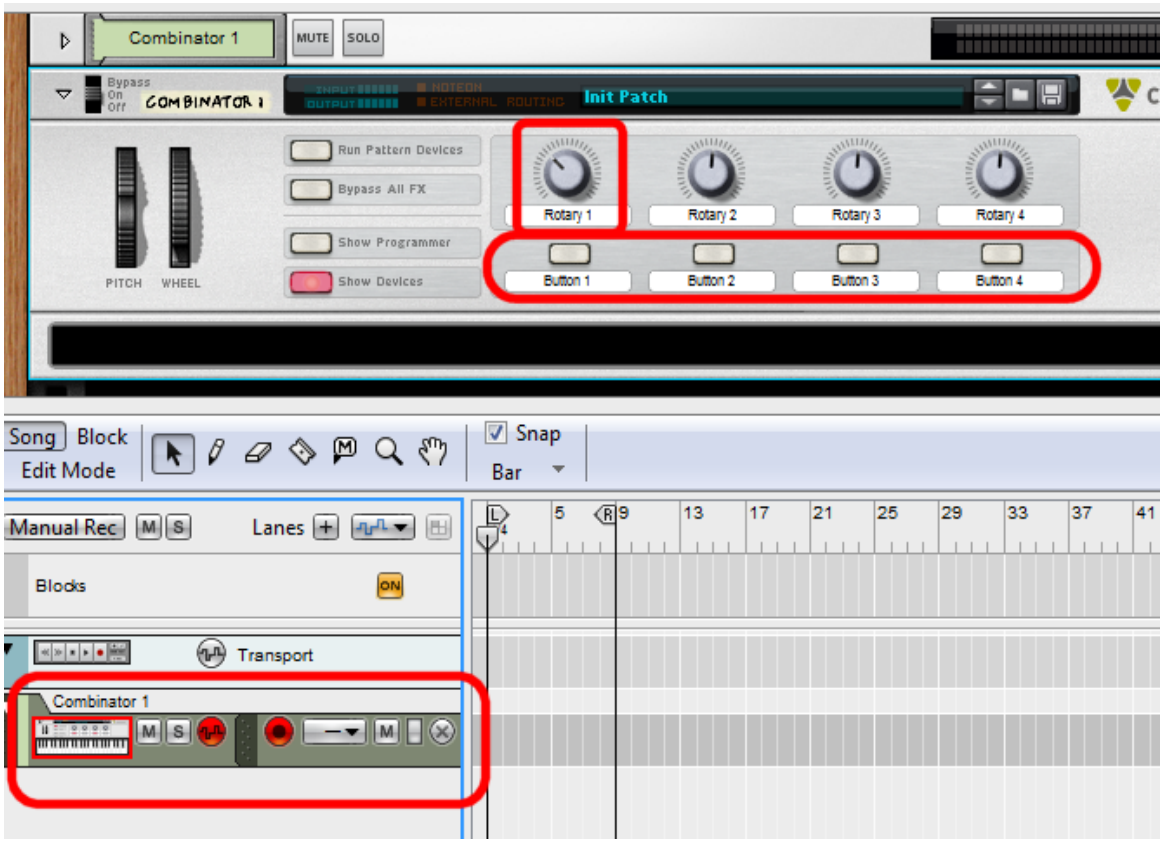
| MIDI Contr. # | MIDI Controller Name | Mixer Master Section | Mixer Channel Strip | Combinator | Mixer 14:2 |
|---|---|---|---|---|---|
| 0 | ControlBankSelect | | | | |
| 1 | ControlModulationWheel | | | Mod Wheel | |
| 2 | ControlBreathController | | | | |
| 3 | ControlUndefined3 | | | | |
| 4 | ControlFootController | | | | |
| 68 | ControlLegatoFootSwitch | | Bypass Insert FX (Insert FX) | | Channel 1 |
| 69 | ControlHold2 | | | | Channel 2 |
| [illegible] | ControlSoundController1 (Sound Variation) | | | | Channel 3 |
| 71 | ControlSoundController2 (Harmonic Intensity) | Rotary 1 (Insert FX) | Rotary 1 (Insert FX) | Rotary 1 | Channel 4 |
| 72 | ControlSoundController3 (Release time) | Rotary 2 (Insert FX) | Rotary 2 (Insert FX) | Rotary 2 | Channel 5 |
| 73 | ControlSoundController4 (Attack time) | Rotary 3 (Insert FX) | Rotary 3 (Insert FX) | Rotary 3 | Channel 6 |
| 74 | ControlSoundController5 (Brightness) | Rotary 4 (Insert FX) | Rotary 4 (Insert FX) | Rotary 4 | Channel 7 |
| 75 | ControlSoundController6 | Button 1 (Insert FX) | Button 1 (Insert FX) | Button 1 | Channel 8 |
| 76 | ControlSoundController7 | Button 2 (Insert FX) | Button 2 (Insert FX) | Button 2 | Channel 9 |
| 77 | ControlSoundController8 | Button 3 (Insert FX) | Button 3 (Insert FX) | Button 3 | Channel 10 |
| 78 | ControlSoundController9 | Button 4 (Insert FX) | Button 4 (Insert FX) | Button 4 | Channel 11 |
| [illegible] | ControlSoundController10 | | | | Channel 12 |

Using the chart (which we can dowload from Propllerheads site) we identify that Combinator Rotary 1 is cc#71, and the four combinator switches are cc#75, 76, 77, 78 respectively. Now we can create an Arduino sketch to link these to the controls on our shield...

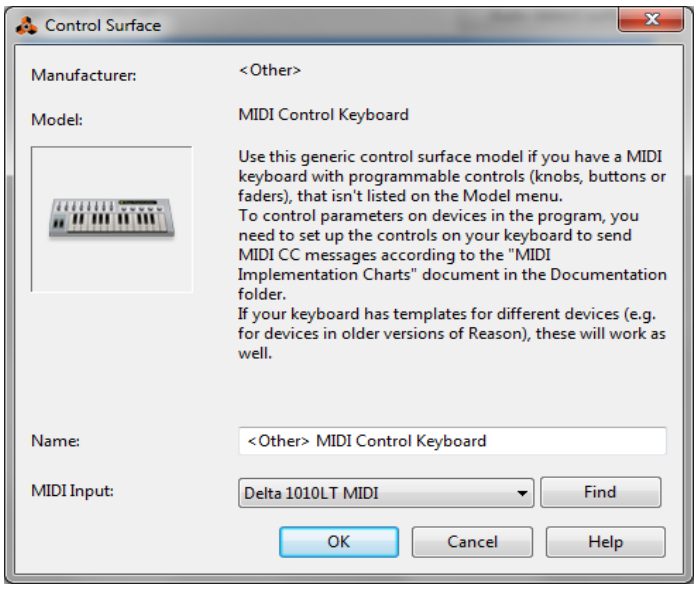
```
#define SWITCHAPIN  5
#define SWITCHBPIN  2
#define SWITCHCPIN  4
#define SWITCHDPIN  3

void setup()
{
  Serial.begin(31250);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);
  pinMode(SWITCHCPIN, INPUT);
  pinMode(SWITCHDPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);
  digitalWrite(SWITCHCPIN, HIGH);
  digitalWrite(SWITCHDPIN, HIGH);
}

void sendController(byte cc, byte value)
{
  Serial.write((byte)0xB0);
  Serial.write(cc);
  Serial.write(value);
}

int lastValue[5] = {0};
void checkController(byte index, byte cc, int value)
```

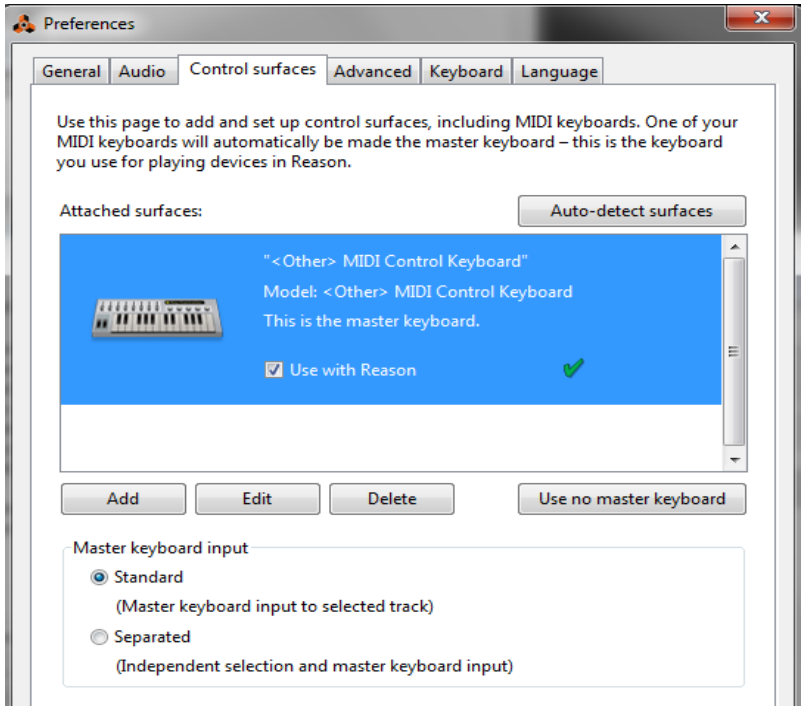
```
{
  value = constrain(value,0,127);
  if(value < lastValue[index] - 1 || value > lastValue[index] + 1)
  {
    sendController(cc, value) ;
    lastValue[index] = value;
  }
}


void loop()
{
  checkController(0, 71, analogRead(0)/8);
  checkController(1, 75, digitalRead(SWITCHAPIN) == LOW ? 127 : 0);
  checkController(2, 76, digitalRead(SWITCHBPIN) == LOW ? 127 : 0);
  checkController(3, 77, digitalRead(SWITCHCPIN) == LOW ? 127 : 0);
  checkController(4, 78, digitalRead(SWITCHDPIN) == LOW ? 127 : 0);

}
```

Armed with a MIDI implemenation chart, lets turn the MIDI Playground shield into a MIDI control surface! This example connects the pot and 4 switches to a rotary controller and 4 switches on the Combinator module.




To make this work, Reason needs to be configured to receive MIDI from the interface that we have the Arduino connected to, as shown blow. If you don't use Reason, don't worry, you can still more than likely get this working – you'll just need to find the correct settings yourself.




Now twiddle the pot and press buttons and you should see the controls on your connected device being worked. Hopefully you can now see how the same principle might be extended into a full MIDI control surface for your synth or DAW software.

We're not going to dwell on Reason too much, nor get too deep into electronics right now, however if you want to take this further, a a couple of points are:

1 – The above example uses the simplest and quickest way to link a Reason device to MIDI. For a real control surface you'd want to mess with the “Advanced MIDI” settings so that you dont need to keep the devices selected in the sequencer for them to receive MIDI. Other software might have similar things to think about.

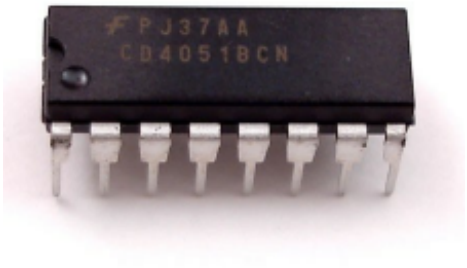
2 – We've only got one pot and four switches here , so we have limited scope to make a useful control surface from the MIDI playground shield. However this is just the start... the Arduino has 6 Analog inputs and 11 usable digital inputs, so we can do 6 pots and 11 switches right off the bat. However there are straighforward ways to extend these ranges using external electronic

components. Some ideas are:

- **Multiplexer Chip** – this chip acts a bit like a "telephone exchange" between a set of incoming input lines and a single outgoing line. For example the 4051 chip (to be had for less than a pound) lets you "dial up" one of 8 inputs to connect to a single input on your Arduino.

  You need to use 3 digital outputs to "do the dialling" but these can be shared between multiple 4051 chips, switching them all together. For example 6 of these chips (one for each of your analog inputs) can be used to read 48 pots in 6 steps, at the expense of 3 digital outputs.

  You can use the 4051 to read digital inputs like switches, too. For more information take a look at **http://www.arduino.cc/playground/Learning/4051**




- **Shift Register** – if you have a lot of **digital** inputs to read, you might want to look at shift registers. These handy chips let you grab a set of digital (HIGH/LOW) inputs, then pass the values out one by one over a single wire.

  For example the 74HC165 has 8 inputs and needs 3 of the Arduino digital pins to work it. This means you get 8 digital inputs for the price of 3, but its even better as you can chain these things together. In fact with the same 3 digital pins, the number of inputs you can read this way is unlimted!

  The trade-off is speed - you need to read the inputs one-by-on in sequence, so 8 inputs take something like 8 times longer to read than 8 separate input pins. However unless you go mad on quantity or need to read things *really* fast, the slowness of shift registers is not a problem. http://arduino.cc/en/Tutorial/ShiftIn gives a tutorial for input shift registers.

  You get output shift registers too (74HC595) which can be used to extend the number of digital outputs. http://arduino.cc/en/Tutorial/ShiftOut

  Shift registers look just like multiplexers, so you dont get another picture :)

# Receiving MIDI Note Information

All our exercises so far have concentrated on sending MIDI information from the Arduino. However it is just as possible to receive and process MIDI. A great example of why you might want to do this is to create a new musical instrument!

For example a BuildBrighton member recently built a robotic xylophone, because it is controlled by MIDI we can just download MIDI song files from the internet and play them on the xylophone.

Another reason you might want to receive MIDI is so that you can make some kind of MIDI processor, such as a MIDI effect or an arpeggiator. Here you would receive MIDI from a keyboard, process it in some way, then send the processed MIDI information to a synthesiser to be played.

Receiving MIDI is a little bit more complicated than sending it, because to do it properly we have to be prepared for the full spectrum of MIDI messages to come in on the port and we don't want any of them to confuse our program (even if it ignores the majority of the information)

Something else we might need to think about is “MIDI thru”. MIDI was designed so that multiple instruments can be controlled via a single physical MIDI output – this is after all the reason behind having sixteen MIDI channels, it mean we can send notes to 16 different MIDI instruments via a single cable.

The idea was that the receiving devices would be connected in a chain, with each of them having three sockets on the back; MIDI Input, MIDI Output and MIDI Through (or “Thru”). To play multiple devices we'd connect the MIDI Input of the first device to our controller, then we'd connect the MIDI Thru port of the first device to the second device, and so on. So MIDI Thru would resend anything received on the MIDI Input. A dedicated MIDI Thru is usually connected to the MIDI Input via some electronics, so there would be no need for a program to resend the data, and it would be very fast.

These days it is common for hardware to have a combined MIDI Out/Thru port, where the MIDI input can be echoed (or “thru'd”) to the output by the microcontroller in the device, combined with any MIDI output messages from the device itself. This is probably because the improved speed of modern processors makes this less of an overhead, plus it saves on cost.

Therefore when we build a MIDI device around an Arduino, we need to think about whether we need to support a MIDI thru function (basically meaning we send every message we receive to the output, as well as acting on anything intended for our own device). This is something else to be aware of when you create a device which receives MIDI input, but as we'll see shortly, its really no biggie.

## *ReceiveNotes1.Ino*

This is our first sketch to read MIDI inputs. It simply turns on our 8 LEDs in response to Note On messages for MIDI notes 48 – 55. When a corresponding Note Off message is received, the LED is switched off. See how you can hold chords and have many lights on at the same time.

```
#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6

void setup(void)
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);

  Serial.begin(31250);

}

void setLED(byte note, byte value)
{
  switch(note)
  {
    case 48: digitalWrite(LED1PIN, value); break;
    case 49: digitalWrite(LED2PIN, value); break;
    case 50: digitalWrite(LED3PIN, value); break;
    case 51: digitalWrite(LED4PIN, value); break;
    case 52: digitalWrite(LED5PIN, value); break;
    case 53: digitalWrite(LED6PIN, value); break;
    case 54: digitalWrite(LED7PIN, value); break;
    case 55: digitalWrite(LED8PIN, value); break;
  }
}

byte command;
byte param1;
byte param2;
byte state = 0;
void loop()
{
  if(Serial.available())
  {
    byte ch = Serial.read();
    if(ch >= 0x80)
    {
      command = ch;
      state = 1;
    }
    else if(state == 1)
    {
      param1 = ch;
```

```
      state = 2;
    }
    else if(state == 2)
    {
      param2 = ch;
      state = 1;

      if(command == 0x90 && param2 > 0)
        setLED(param1, HIGH);
      else if(command == 0x90 || command == 0x80)
        setLED(param1, LOW);
    }
  }
}
```

Switching LEDs on might not be the most useful application of MIDI, but this sketch shows the basics of something much more practical. The digital outputs that currently drive the LEDs could instead be connected to other electronics or electromechanical components (e.g. a buzzer, or a solenoid that hits a drum), so a sketch rather like this could become the “brain” of an electronic musical instrument!

As another side note: It really is perfectly possible to use the digital outputs for this exciting kind of real work mechanical stuff, but before you leap into it and start attaching motors and such to the digital outputs of your Arduino, you'll need to learn a bit about the electronics for driving loads.

Basically the digital output pin of an Arduino can only supply a very small amount of current (a few tens of milliamps – thousandths of an amp). However a motor might try to draw an amp or more of current – it might twitch a bit if you're lucky, but thats it. Worse, motors and solenoids (basically anything with big coils of wire in) can produce a “flyback” pulse of high-ish voltage when the current is switched off – this could actually damage the Arduino digital pin.

The answer is to use transistors (and sometimes relays) to do your switching, controlled by the Arduino digital output. Its not too complicated and is nothing to be scared of. If you want to try something like this, google the ULN2803 transistor array chip. This chip lets you switch 8 medium loads, like small motors or solenoids, using 8 digital outputs.

## *The Arduino MIDI Library*

So far all our exercises have used the Serial library to directly work with MIDI data. This has let us see the very guts of MIDI, in the actual bytes of data that are being sent over the line. I hope this gave you a “feel” for how it works, however you may be pleased (or angry) to learn that the Arduino has a way to make this much easier.

Remember how in some of our sketches we created reusable functions that we called again and again (such as the one to send a MIDI note on message)? Well, a “library” is rather like a set of functions that someone else has already written to make our life easier. The Arduino has lots of libraries available for it – in fact you can often create a sketch by simply “glueing” libraries together without really knowing how they work. I guess you could consider this to be a good, or a bad thing.

One such library is the MIDI library. This has functions for sending notes and controllers (but of course we don't need a library to do that for us :o) but also it makes receiving MIDI very easy, and takes care of MIDI Thru and intelligently skips MIDI messages we're not interested in.

From now on we'll be using the MIDI library in the sketches.

### *ReceiveNotes2.Ino*

This sketch is a modificaton of the last one, but using the MIDI library.

```
#include <MIDI.h>

#define LED1PIN  13
#define LED2PIN  11
#define LED3PIN  9
#define LED4PIN  7
#define LED5PIN  12
#define LED6PIN  10
#define LED7PIN  8
#define LED8PIN  6


void setLED(byte note, byte value)
{
  switch(note)
  {
    case 48: digitalWrite(LED1PIN, value); break;
    case 49: digitalWrite(LED2PIN, value); break;
    case 50: digitalWrite(LED3PIN, value); break;
    case 51: digitalWrite(LED4PIN, value); break;
    case 52: digitalWrite(LED5PIN, value); break;
    case 53: digitalWrite(LED6PIN, value); break;
    case 54: digitalWrite(LED7PIN, value); break;
    case 55: digitalWrite(LED8PIN, value); break;
  }
}

void HandleNoteOn(byte channel, byte pitch, byte velocity) {
  setLED(pitch, velocity);
}

void HandleNoteOff(byte channel, byte pitch, byte velocity) {
  setLED(pitch, 0);
}

void setup(void)
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED2PIN, OUTPUT);
  pinMode(LED3PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);
  pinMode(LED5PIN, OUTPUT);
  pinMode(LED6PIN, OUTPUT);
  pinMode(LED7PIN, OUTPUT);
  pinMode(LED8PIN, OUTPUT);

  MIDI.begin(MIDI_CHANNEL_OMNI);
  MIDI.setHandleNoteOn(HandleNoteOn);
  MIDI.setHandleNoteOff(HandleNoteOff);
}

void loop()
{
  MIDI.read();
}
```

In this sketch we create our own function for handlng the note on and note off messages (by switching the corresponding LEDs on and off), then we tell the MIDI library to call these functions when the corresponding MIDI messages are received.

The MIDI library allows us to use “call back” functions like this so all we need to do is decide what we want to do when the message arrives. We don't need to worry about reading the serial port and recognising the byte values in the messages. The library does this for us.

The other important things we need to do are call the MIDI.begin function to start the library (this will set up the serial port), then we need to repeatedly call MIDI.read. The MIDI.read function cause the library to check the MIDI input port and call our handler functions when matching messages arrive.

Now lets combine both the sending and receiving of MIDI notes to make a MIDI effect...

## *ReceiveNotes3.Ino*

This sketch runs a MIDI effect to take any note and turn it into a Major triad (Major chord). The Major triad is made up of the root note, the major third (4 semitones above the root) and the fifth (7 semitones above the root)

For example when a C is received (Note #48) it will send out C (note #48) to the MIDI output, but will also send E (#52) and G (#55). These notes together make up the basic “triad” for a C major chord.

We can simply repeat the same calculation when a note off message is received, and silence the same 3 notes we started playing when we received the note on message

```
// MAJOR TRIAD
#include <MIDI.h>

#define LED1PIN  13

void HandleNoteOn(byte channel, byte pitch, byte velocity) {
  digitalWrite(LED1PIN,HIGH);
  MIDI.sendNoteOn(pitch, velocity, channel);
  MIDI.sendNoteOn(pitch + 4, velocity, channel);
  MIDI.sendNoteOn(pitch + 7, velocity, channel);
  digitalWrite(LED1PIN,LOW);
}

void HandleNoteOff(byte channel, byte pitch, byte velocity) {
  digitalWrite(LED1PIN,HIGH);
  MIDI.sendNoteOff(pitch, velocity, channel);
  MIDI.sendNoteOff(pitch + 4, velocity, channel);
  MIDI.sendNoteOff(pitch + 7, velocity, channel);
  digitalWrite(LED1PIN,LOW);
}

void setup(void)
{
  pinMode(LED1PIN, OUTPUT);

  MIDI.begin(MIDI_CHANNEL_OMNI);
  MIDI.turnThruOff();
  MIDI.setHandleNoteOn(HandleNoteOn);
  MIDI.setHandleNoteOff(HandleNoteOff);
}

void loop()
{
  MIDI.read();
}
```

The MIDI library is used in the same way as before, with handler functions set up for note on and note off messages. We also call MIDI.sendNoteOn and MIDI.sendNoteOff to send the output messages.

Notice how we also call MIDI.turnThruOff. This stops the MIDI library automatically handling MIDI Thru. In this case we don't want that to happen since this would duplicate the root note (We could have left MIDI thru running and just sent the two additional notes – letting the root note get thru'd by the MIDI library, but it might not be quite so clear what was going on)

## *ReceiveNotes4.Ino*

Now something fun... this is a very simple arpeggiator, which receives MIDI note on/off information and repeatedly cycles through all the notes that are held down at any one time, sending those notes to the output and hence "arpeggiating" the chord (i.e. playing the notes of the chord individually in sequence)

```
// ARPEGGIATOR
#include <MIDI.h>
byte noteOn[128] = {0};

void HandleNoteOn(byte channel, byte pitch, byte velocity)
{
  noteOn[pitch] = velocity;
}

void HandleNoteOff(byte channel, byte pitch, byte velocity)
{
  noteOn[pitch] = 0;
}

void setup(void)
{
  MIDI.begin(MIDI_CHANNEL_OMNI);
  MIDI.turnThruOff();
  MIDI.setHandleNoteOn(HandleNoteOn);
  MIDI.setHandleNoteOff(HandleNoteOff);
}

unsigned long nextNoteTime = 0;
byte lastNotePlayed = 0;

void loop()
{
  MIDI.read();
  if(millis() > nextNoteTime)
  {
    nextNoteTime = millis() + 100;
    int thisNote = lastNotePlayed;
    if(lastNotePlayed > 0)
    {
      MIDI.sendNoteOff(lastNotePlayed, 0, 1);
      lastNotePlayed = 0;
    }
    for(int i=0; i<255; ++i)
    {
      thisNote = thisNote + 1;
      if(thisNote > 127)
        thisNote = 1;
      if(noteOn[thisNote] > 0)
      {
        MIDI.sendNoteOn(thisNote, 127, 1);
        lastNotePlayed = thisNote;
        break;
      }
   }
  }
}
```

# Timing And Synchronisation

As I mentioned before, there are many types of MIDI message other than notes and controllers. Some of these messages have the function of synchronising MIDI devices so they keep synchronised in time with each other.

MIDI has a couple of methods for synchronisation. One method works by sending time position information within the structure of a song, and might be used to lock devices together while a song is playing, or while you scroll through a song while editing.

The other method (Beat Clock) simply uses messages for the following actions

- "Start playing at the beginning"
- "Stop playing"
- "Continue playing from where you last stopped"

Once playing, a "clock tick" message is sent to synchronise the devices. The tick message is sent by the "clock source" device at the rate of 24 ticks per quarter note ("Beat"). The clock source would typically be something like a sequencer or DAW program which is controlling all the other devices on the MIDI chain (There can be only one clock source on a given MIDI chain)

These messages are being sent pretty rapidly... say our clock source is running at the tempo of 120 beats per minute (bpm) – so 2 beats per second – this would mean that 48 of these clock tick messages are coming over the MIDI link each and every second.

A "slave" device, such as a drum machine, stays in synch with the clock source by counting the tick messages. Note that there is no "position in the song" conveyed by this method... it assumes that all slave devices are somehow sequencing their own patterns (such as drum machine) and have a concept of what "Start playing at the beginning" means.

Therefore a slave device just needs to obey the Start/Stop/Continue messages and count the tick messages. It's beautifully simple!

A slave device should not even need to perform any time calculations to time things to take place "between ticks" for beat lengths such as triplets. This is because the MIDI beat clock rate has been chosen to divide up nicely for standard musical beat divisions.

| | | |
|---|---|---|
| **Full Note**<br>96 ticks | **Dotted half-note**<br>72 ticks | **Half-note**<br>48 ticks |
| **Dotted quarter-note**<br>36 ticks | **Quarter-note**<br>24 ticks | **Dotted eighth note**<br>18 ticks |
| **Half note triplet**<br>16 ticks | **Eighth note**<br>12 ticks | **Quarter-note triplet**<br>8 ticks |
| **Eighth note triplet**<br>4 ticks | **Dotted sixteenth**<br>9 ticks | **Sixteenth note**<br>6 ticks |
| **Thirty second note**<br>3 ticks | **Sixteenth triplet**<br>2 ticks | |

Lovely jubby – someone really thought about this!

## *BeatClock1.Ino*

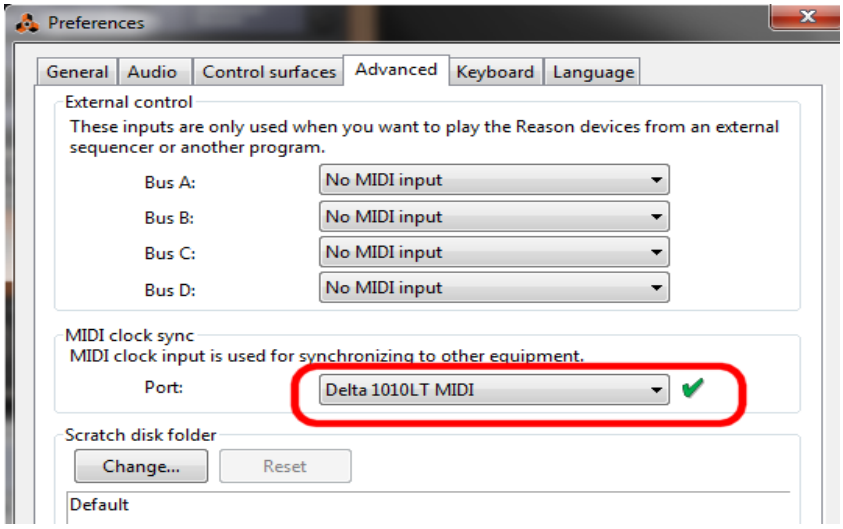
In this sketch we'll make a MIDI beat clock source. The tempo is controlled with the pot, and Start/Stop are controlled by buttons A and B

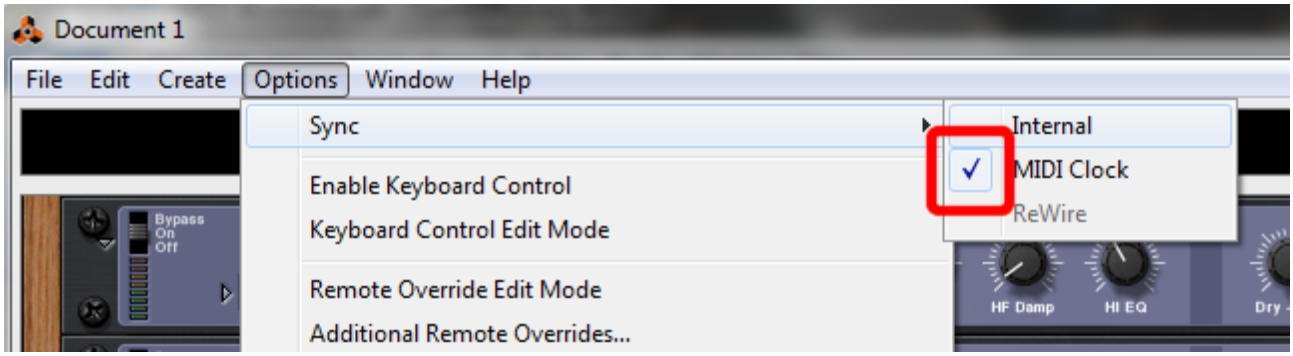
```
// METRONOME
#include <MIDI.h>

#define LED1PIN  13
#define LED4PIN  7

#define SWITCHAPIN  5
#define SWITCHBPIN  2

void setup()
{
  pinMode(LED1PIN, OUTPUT);
  pinMode(LED4PIN, OUTPUT);

  pinMode(SWITCHAPIN, INPUT);
  pinMode(SWITCHBPIN, INPUT);

  digitalWrite(SWITCHAPIN, HIGH);
  digitalWrite(SWITCHBPIN, HIGH);

  MIDI.begin(MIDI_CHANNEL_OMNI);
}

unsigned long nextClockTime = 0;
int clockDelay = 20;
int tickCount = 0;
byte running = 0;
void loop()
{
  MIDI.read();
  unsigned long milliseconds = millis();
  if(milliseconds > nextClockTime)
  {
    if(running)
      MIDI.sendRealTime(Clock);

    nextClockTime = milliseconds + clockDelay;
    tickCount = tickCount + 1;
    if(tickCount > 23)
    {
      tickCount = 0;
      digitalWrite(LED1PIN,HIGH);
    }
    else
    {
      digitalWrite(LED1PIN,LOW);
    }
  }
  else if(digitalRead(SWITCHAPIN) == LOW)
  {
      if(!running)
      {
        MIDI.sendRealTime(Start);
        tickCount = 0;
        running = 1;
        digitalWrite(LED4PIN,HIGH);
      }
  }
  else if(digitalRead(SWITCHBPIN) == LOW)
```

```
  {
      if(running)
      {
        MIDI.sendRealTime(Stop);
        running = 0;
        digitalWrite(LED4PIN,LOW);
      }
  }
  else
  {
    clockDelay = analogRead(0) / 10;
  }
}
```

See how we use the MIDI.sendRealTime function to send out the MIDI beat clock messages.

For the sketch to control a sequencer device or software, it must be configured to accept external MIDI synch. Here are the associated options in Reason.



## Other MIDI Features

There are many things we didn't have time to discuss in this workshop, but the following second gives you some insight and pointers if you want to know more,

**Aftertouch** – This refers to changes in the pressure on a key while the key is held down and allows an extra bit of expression in playing, assuming the synthesiser acts on the messages. There are two types of aftertouch message – Polyphonic aftertouch supports individual aftertouch messages per held note, Channel Pressure applies the same aftertouch to all the held notes. Not all synthesisers respond to aftertouch and not all controllers are capable of sending it..

**Program Change** – Change the "program" (sound setting) for a channel. For example we might select a Piano sound by requesting "Program #1".

Programs are referenced by numbers, so it is important to know which number refers to which sound – particularly if we are playing a pre-made MIDI sequence file (for example an orchestral piece) which really needs a specific instrument type on each channel for it to sound correct, and we don't end up with the Violin part being played on a Gunshot sound, for example. The "General MIDI" (GM) standard helps to ensure this – among other things, the GM standard specifies standard program numbers for specific standard instrument sounds.

GM also defines the specific notes to which specific drum sounds (e.g. kick, snare, hi-hat) should be mapped. Conventionally, drums are played on MIDI channel 10.

**System Exclusive** – The designers of MIDI generously included a feature to allow manufacturers of equipment to devise and send their own messages, completely outside of the MIDI standard. For example it is usually possible to take a "dump" of the data stored in a device like a sequencer or a sampler, or even the patch settings in a synthesiser, and back them up over MIDI to a computer or (especially in past years) a special "MIDI librarian" device. This is achieved by a special feature called "System Exclusive" (SysEx) messages. The sender starts with a special "Start of SysEx" message, then sends all the required data (which the receiver obviously must be able to understand) then finally sends the "End of SysEx" message. In order that all this nonstandard data cannot be confused as MIDI messages, no data byte can be greater than 0x7f (Magic number 127 again). This usually means that the real data needs to be manipulated by both the sender and receiver so that all data values fit into the 0-127 range (in technical terms the 8 bit bytes are broken down into a string of 7 bit values which are reassembled into the 8-bit values at the receiver). Realtime messages like MIDI clock tick can "break into" the stream of SysEx data, but they are easily recognised by being values greater than 127.

When we met **Continuous Controller** messages earlier they might have looked easy – and they can be – but MIDI does plays some special tricks with these messages which can at times seem quite mind blowing. Some examples are

- Certain CC#s define Channel Mode messages, with special functions. For example sending a MIDI message to CC#123 actually means "Turn all notes off".

- Some controllers are used to give a finer level of control of other CC#s which would otherwise just have a range of 0-127. For example CC#1 is the modulation wheel. We can set it to one of 128 values (0-127) by sending to CC#1. However if we send a value to CC#33 we can "fine" set the modulation wheel to 128 positions between each of the positions we can access from CC#1. For example sending 0x23 to CC#1 and 0x3f to CC#33

will sent the mod wheel halfway between positions 0x23 and 0x24.

- CC#100 and 101 together define the Registered Parameters (RPN) to which data entry (#6 together with #38) messages are applied. For example RPN 0x0002 is “Channel Coarse Tuning”. If we send 0x00 to CC#100 and 0x02 to CC#101 we can then send 0x00 to CC#6 and 0x3f to CC#38 to sent the Channel Coarse Tuning to 0x003f. Phew!

- CC#98 and 99 together defined Non-registered Parameters (NRPN) which are messaged just like RPNs. However, unlike RPNs, NRPNs are manufacturer specific and are not defined by the MIDI specification. For example a specific model of synthesiser might use NRPN 0x0123 to be “Filter 3 resonance”. Using NRPN messages it is possible to dynamically modify this internal parameter via MIDI

## *MIDI Over USB*

In this workshop we have concentrated on using “raw” MIDI. That is we've been looking at MIDI messages that are transmitted on a serial cable wth 5-Pin DIN connectors at the same old 31250 bits-per-second, just like the very first MIDI devices of the 1980s.

The MIDI protocol (message formats) might not have changed in that time, but there are now different ways in which those messages can be sent, other than the old serial cables. The most interesting and useful is probably the ability to send MIDI messages over USB.

This is particularly useful because every modern computer has USB ports, but pretty much none have  MIDI DIN ports as standard. Consumer level sound cards tend not to have MIDI support now either (once upon a time every soundcard had MIDI and joystick ports). Even MIDI devices themselves, like keyboards and control surfaces, are now sold which only have USB connections..

When you plug these devices into your computer, they show up just like a normal MIDI device attached to a real MIDI port, and the computer talks to them the same way with the same types of MIDI message we've been learning about (so the MIDI protocol is alive and well, it just uses different wires!)

This works pretty seamlessly because each USB MIDI device supports a standard USB “MIDI Device Class” as defined in the USB standards. Modern operating systems can use them as MIDI interfaces, usually without special drivers. An example of such a device is a MIDI-USB cable which converts MIDI DIN signals into USB MIDI messages so that a MIDI DIN device can be connected to a computer without a dedicated MIDI interface.

USB is actually a pretty complex beast. You can implement a USB device in an Arduino using a shield, but that would be beyond the scope of this workshop. A much simpler way of doing this is using a nice Arduino-like board called the “Teensy”. This board comes talking USB with libraries that implement the MIDI Device Class for you (plus several others) so you can pretty much talk MIDI from the get-go.

We did consider using Teensy's for this workshop, but felt that using USB for MIDI really masks the core of what MIDI is all about, and takes aways the “hands-on” ness of sending real MIDI messages over real MIDI hardware, without the need to involve a computer.

However Teensy is defintiely worth a look, particularly if you wanted to build control surfaces for operating DAW software that will only ever be used connected with a computer.