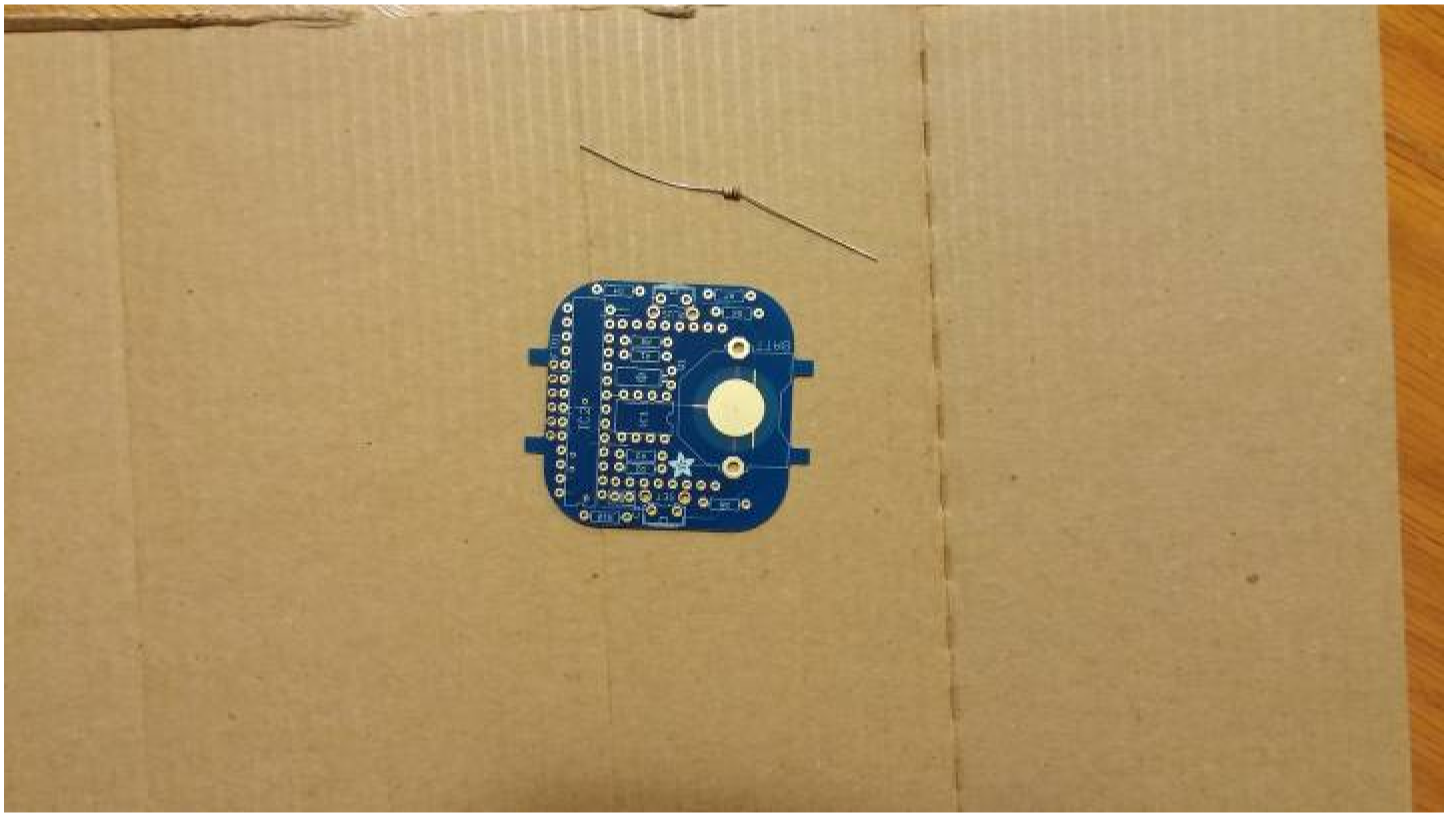
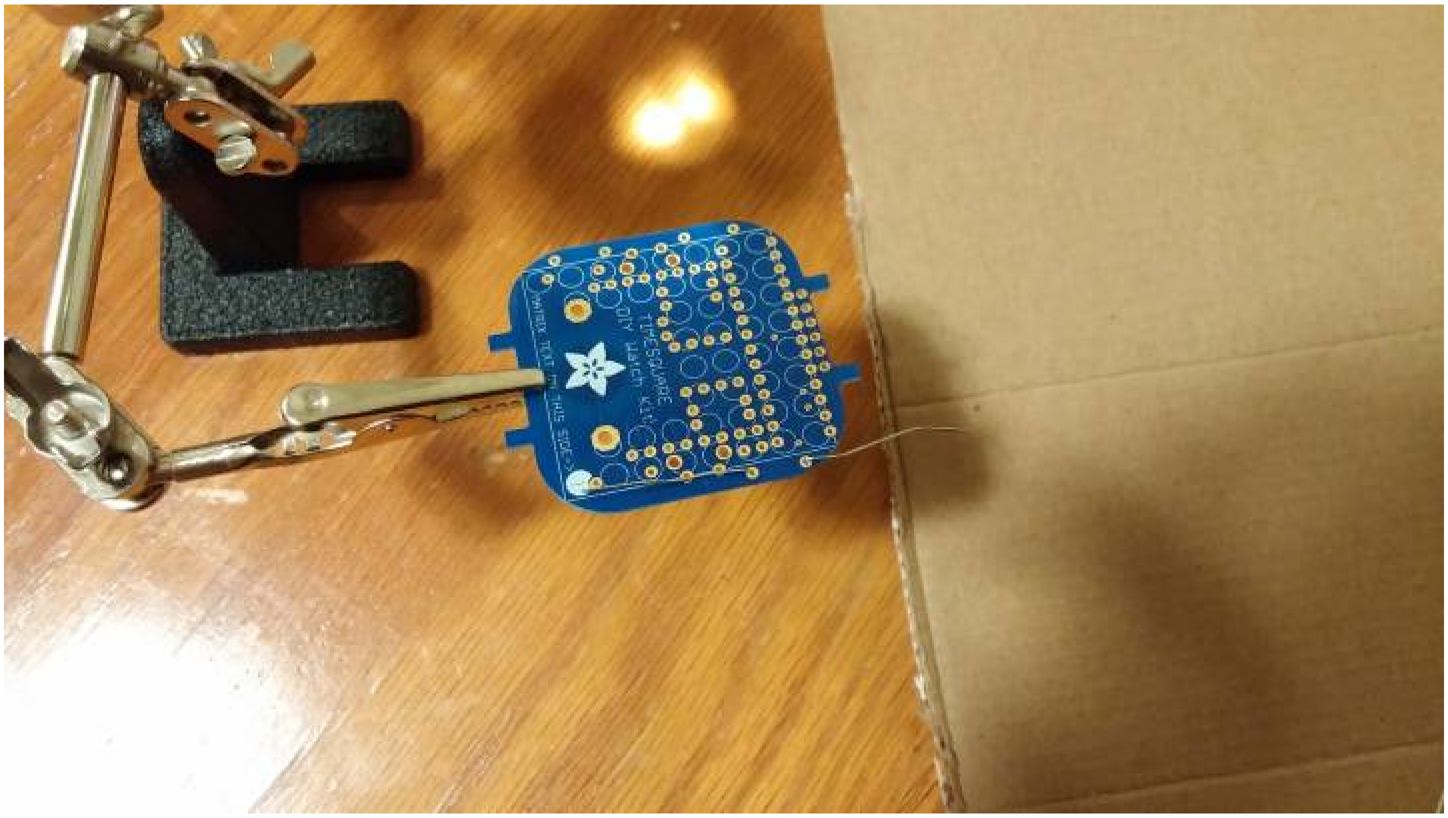
Watching Math Patterns

by Chris Yiu, Western University
a project by George Gadanidis, Western University
www.researchideas.ca

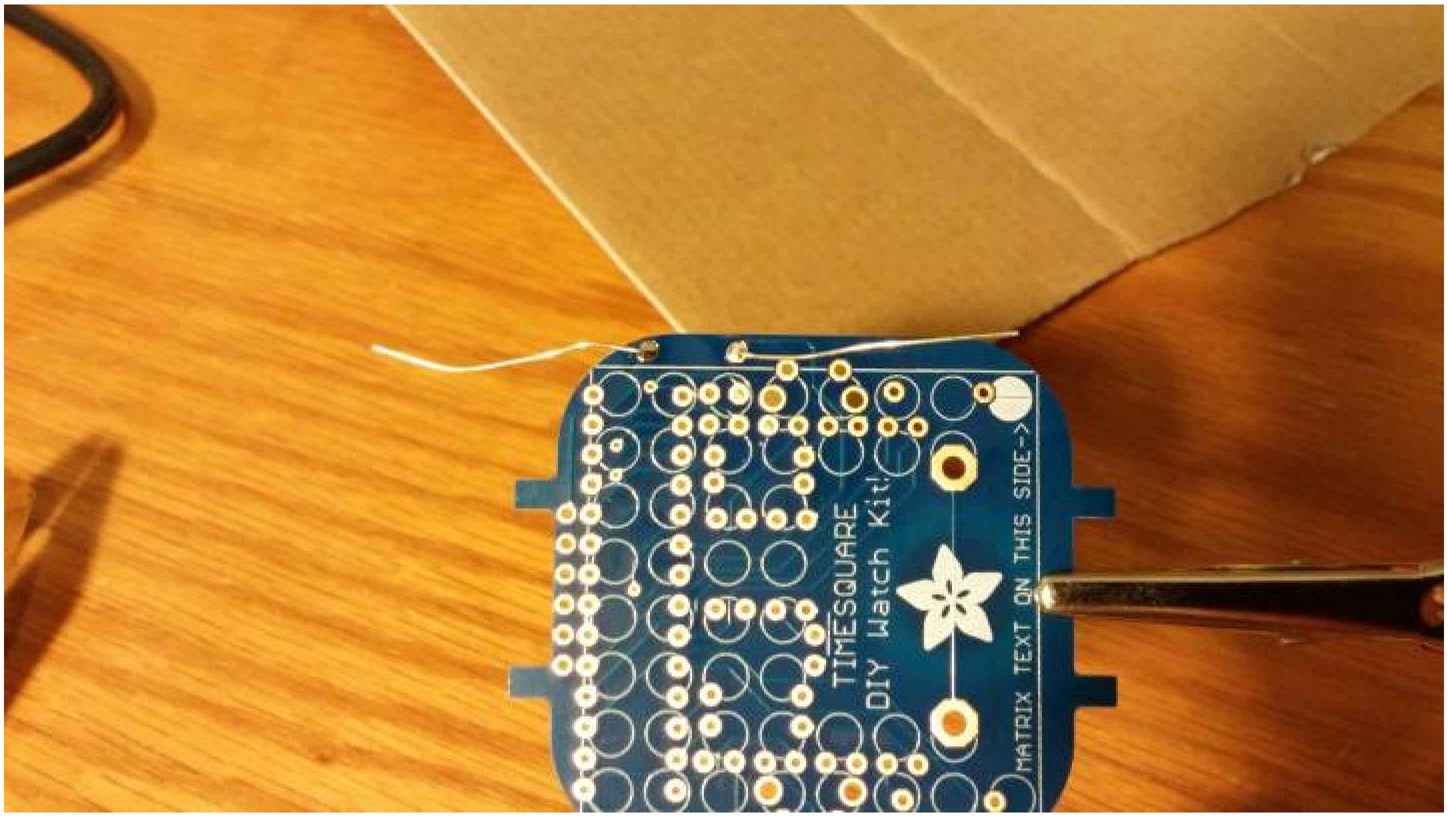
Getting everything together – laptop for the instructions, toolbox on the left, and parts on the right!

Pulling everything out of their boxes. Lots of stuff!

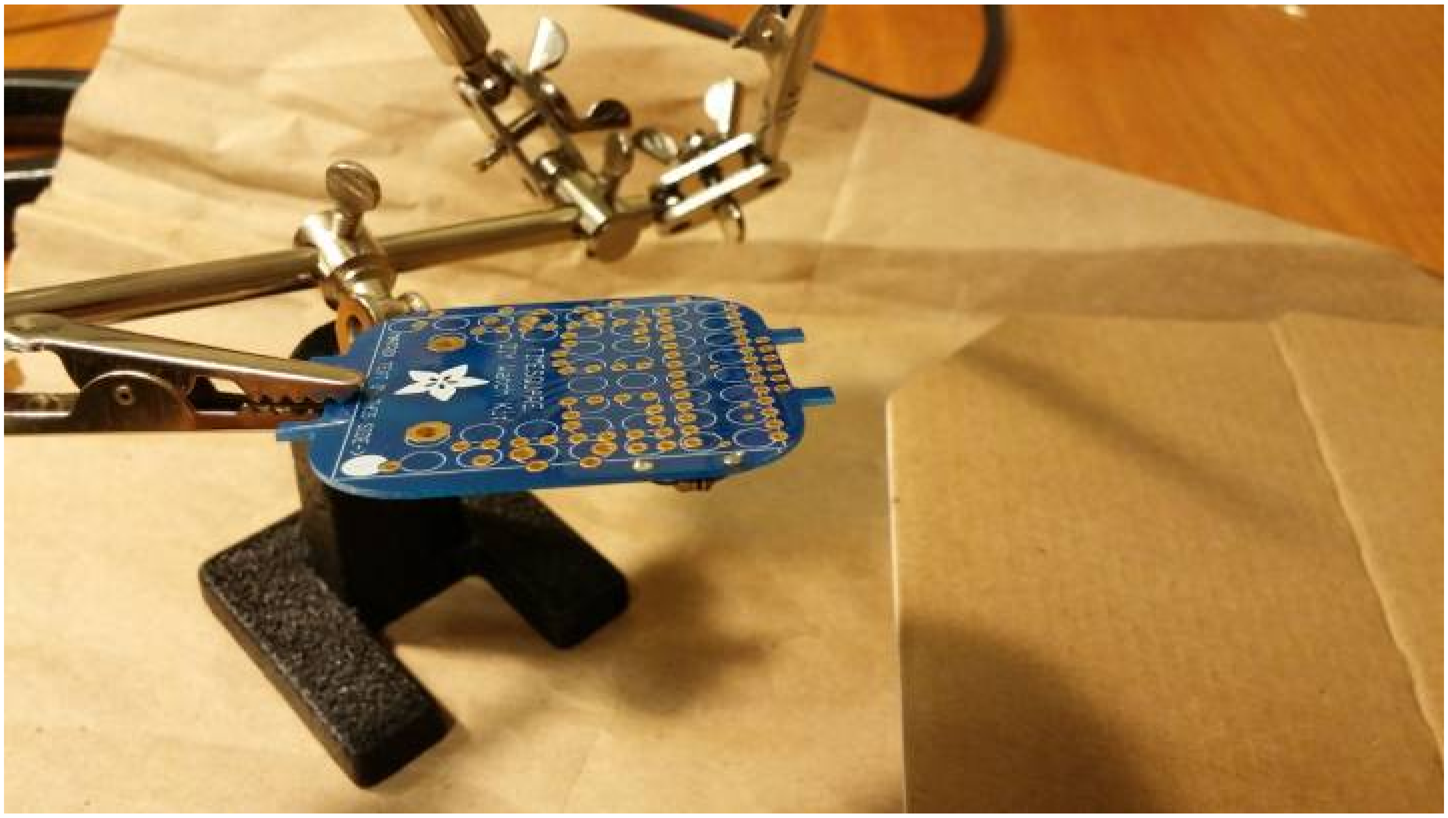

Here's the board and a resister. We'll eventually be soldering lots of stuff onto the board.



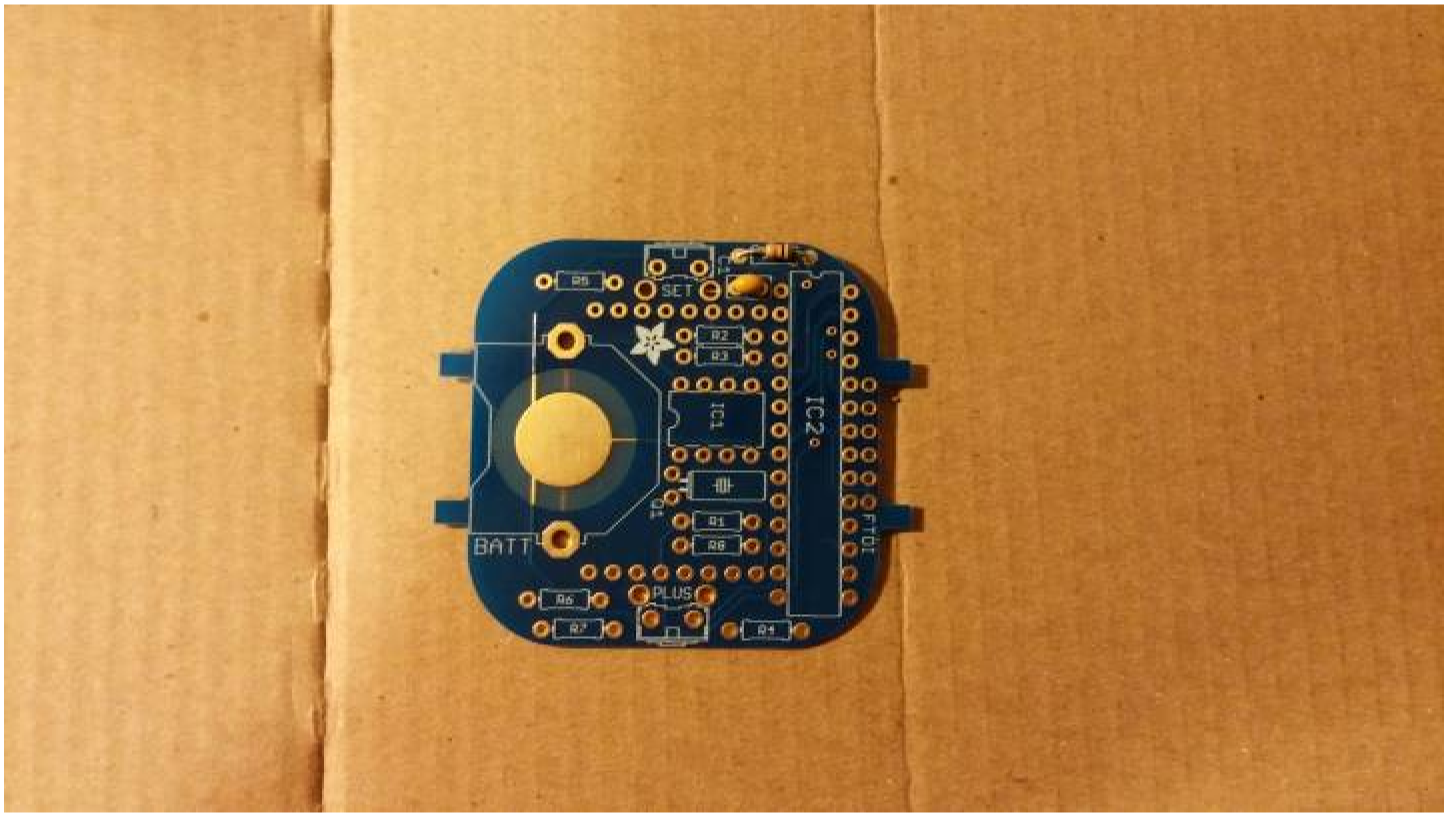
Threading the wires through and flipping it over – ready to solder!



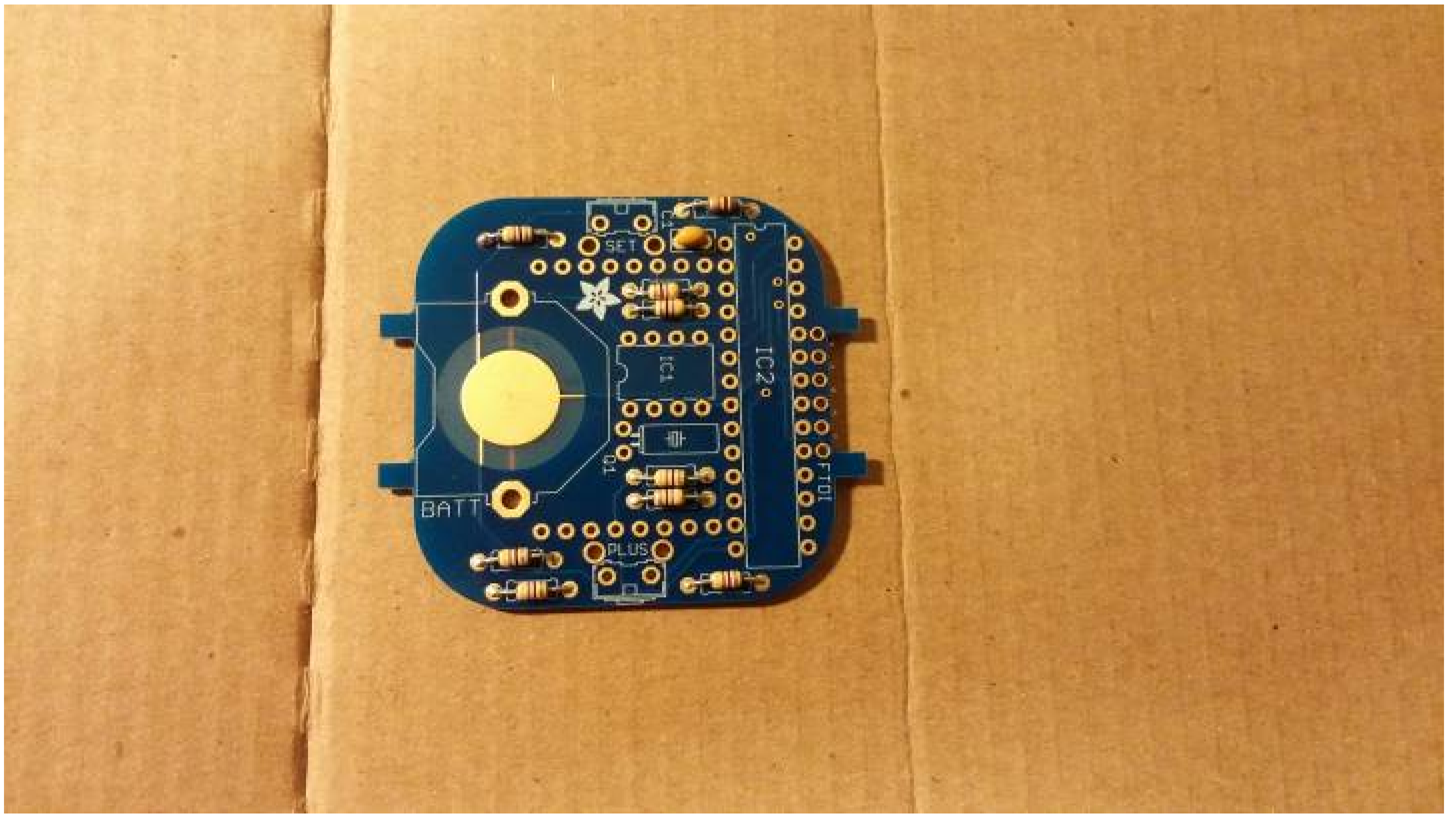
After soldering...I’m not the best at it, but it’ll do!



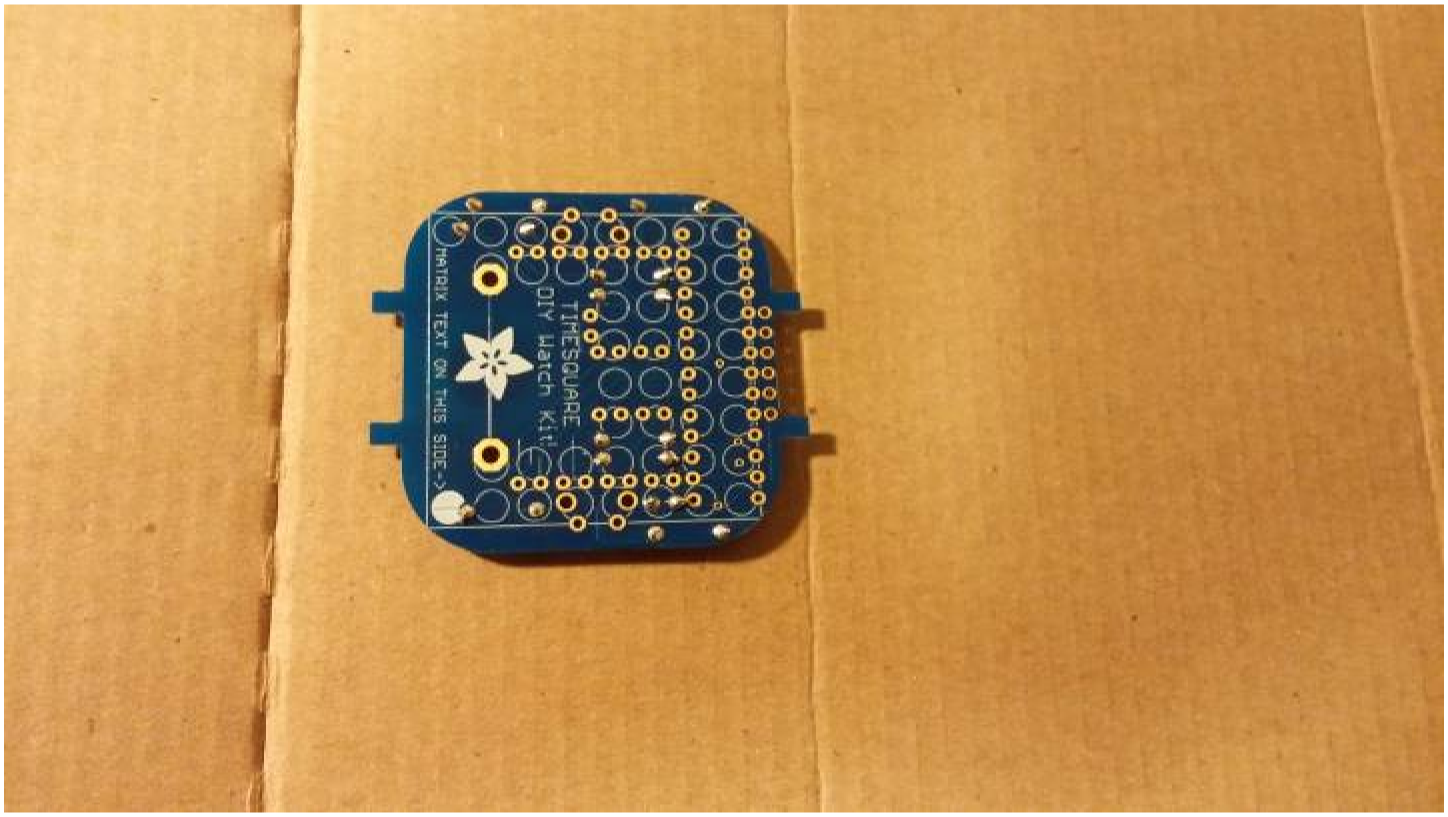
After snipping the wires. Looks good!



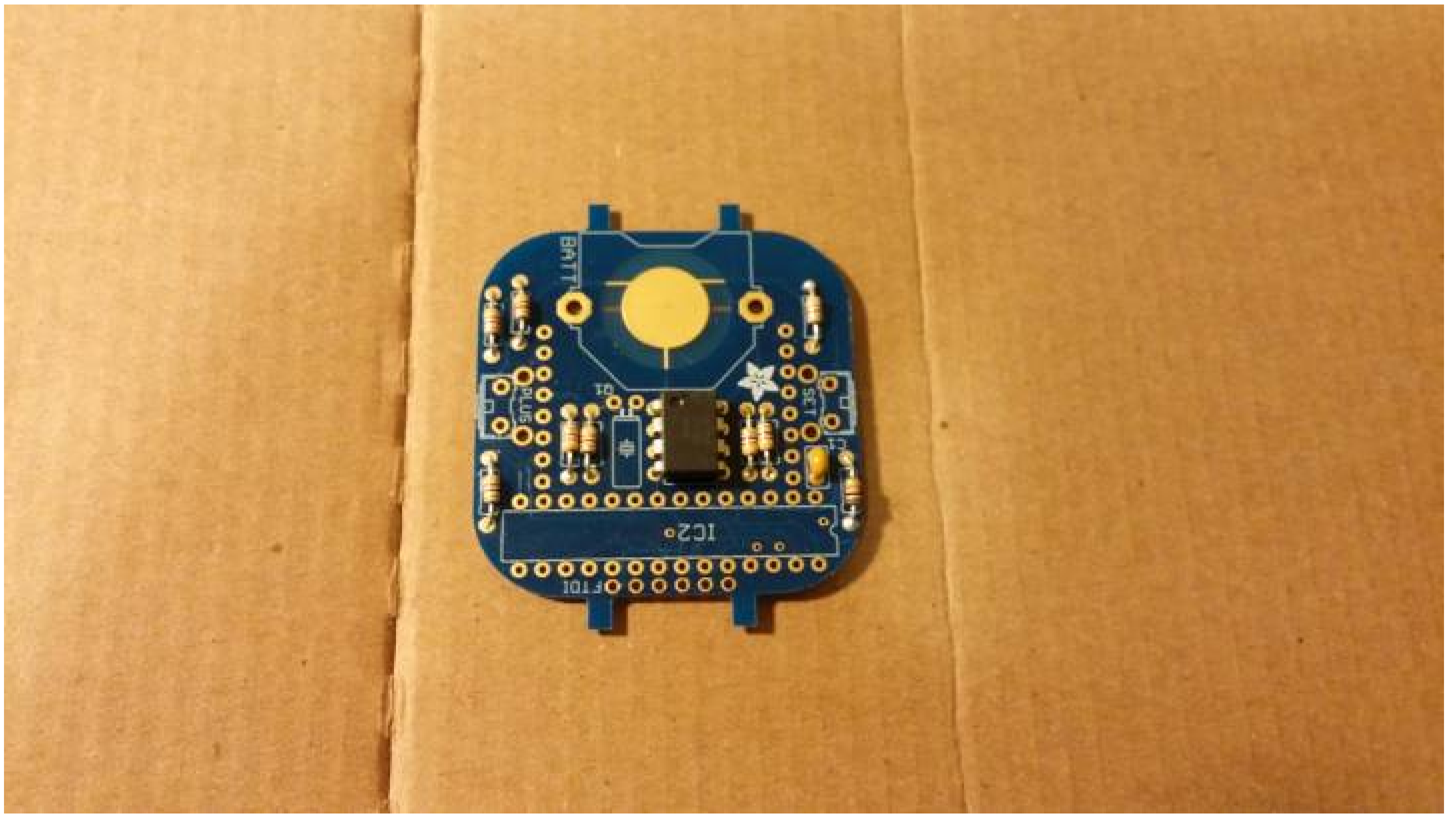
This one is a ceramic capacitor. Used to help reset the chip.



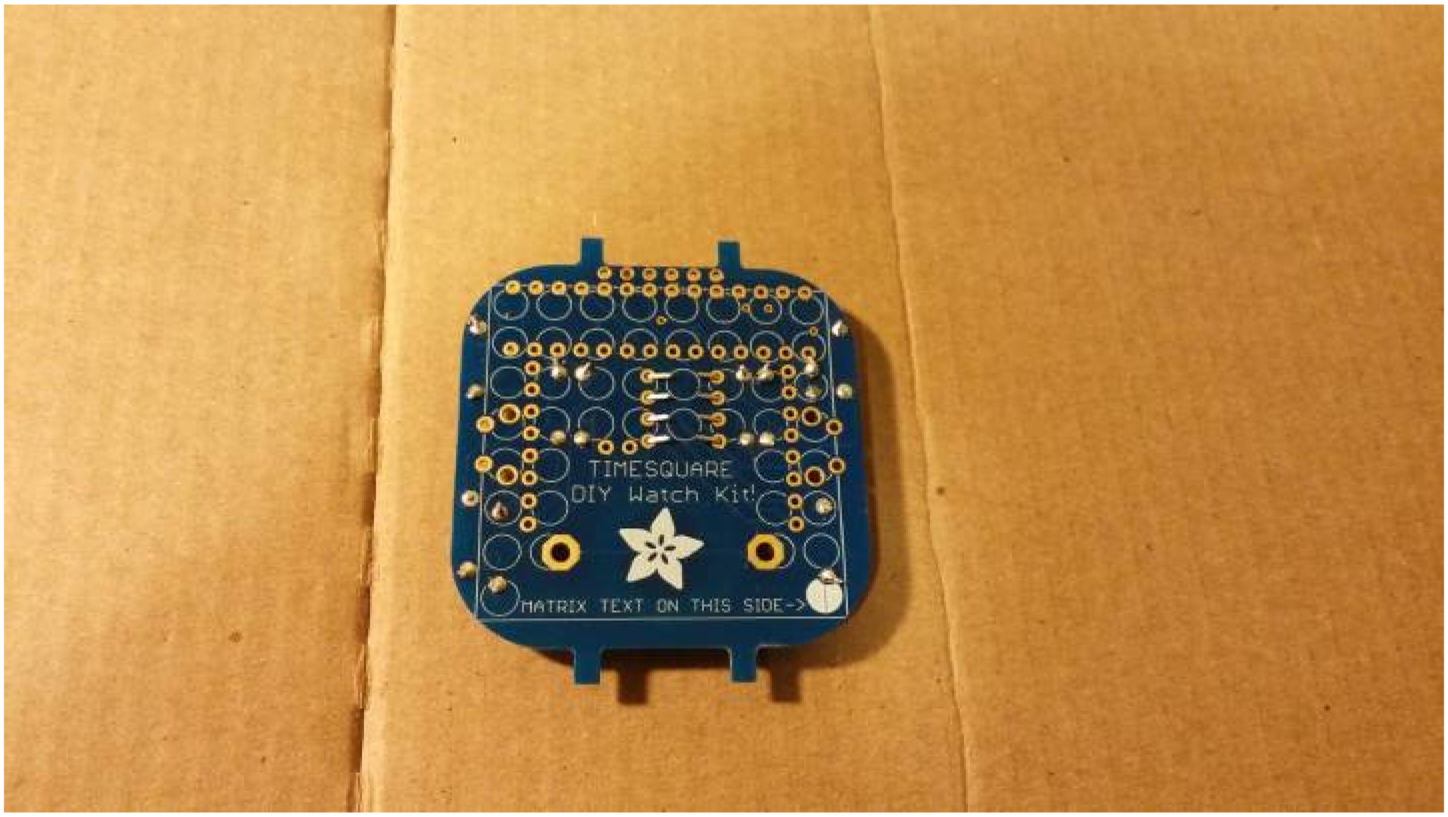
Lots of other resisters to solder in!



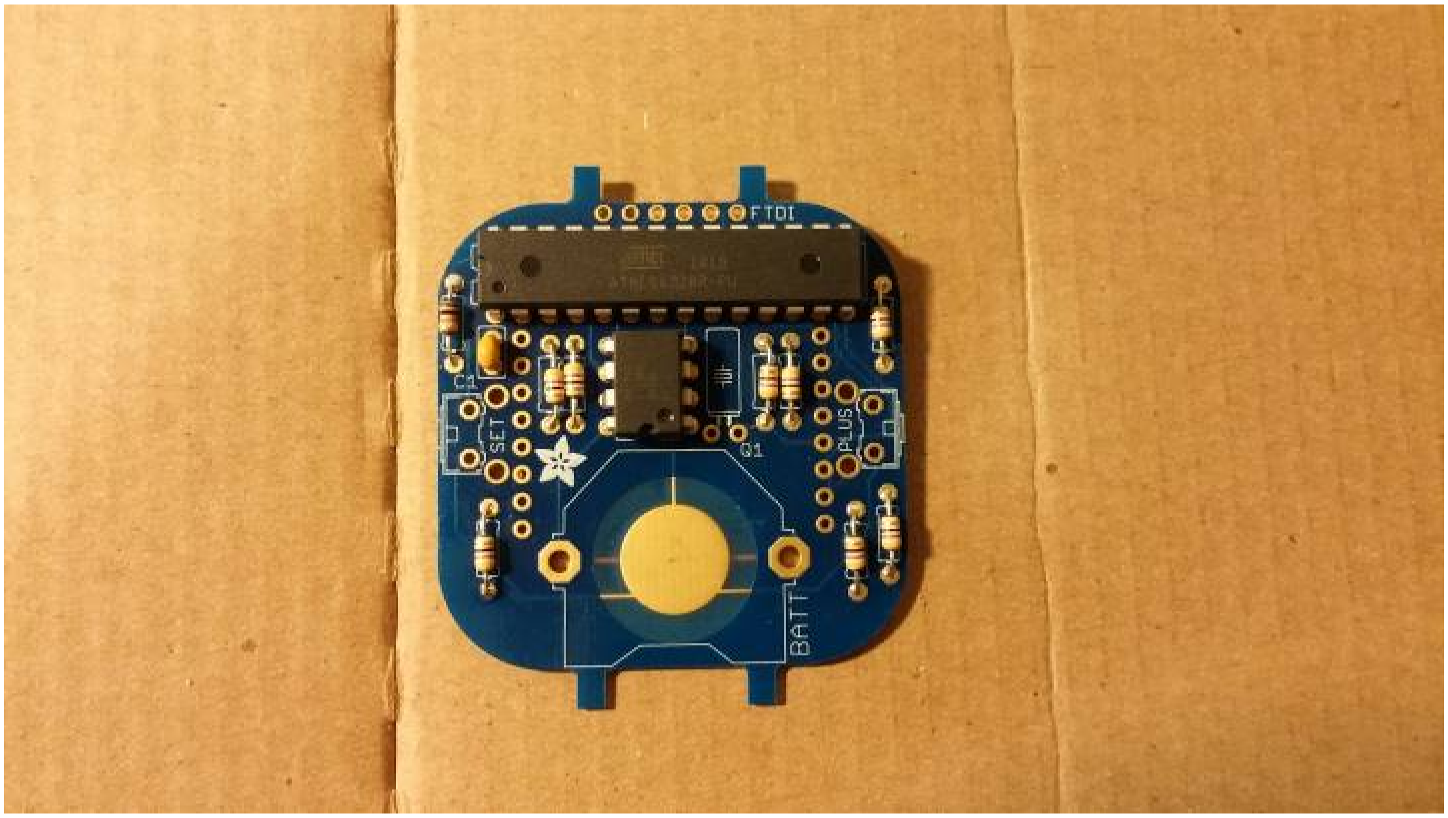
After soldering and snipping.



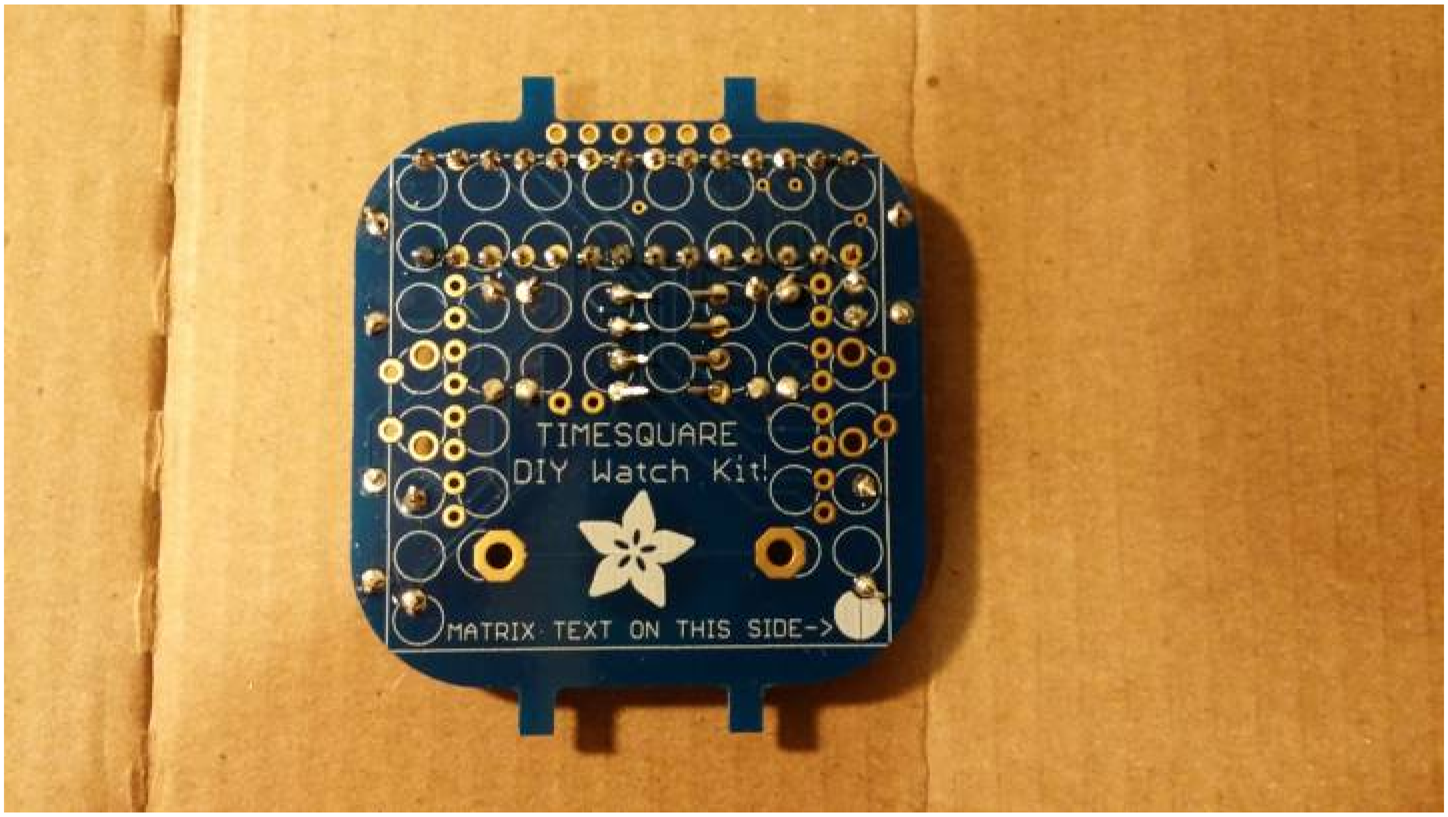
Our first chip – the DS1337. This one keeps track of the time for us.



Bendy pins to solder down!

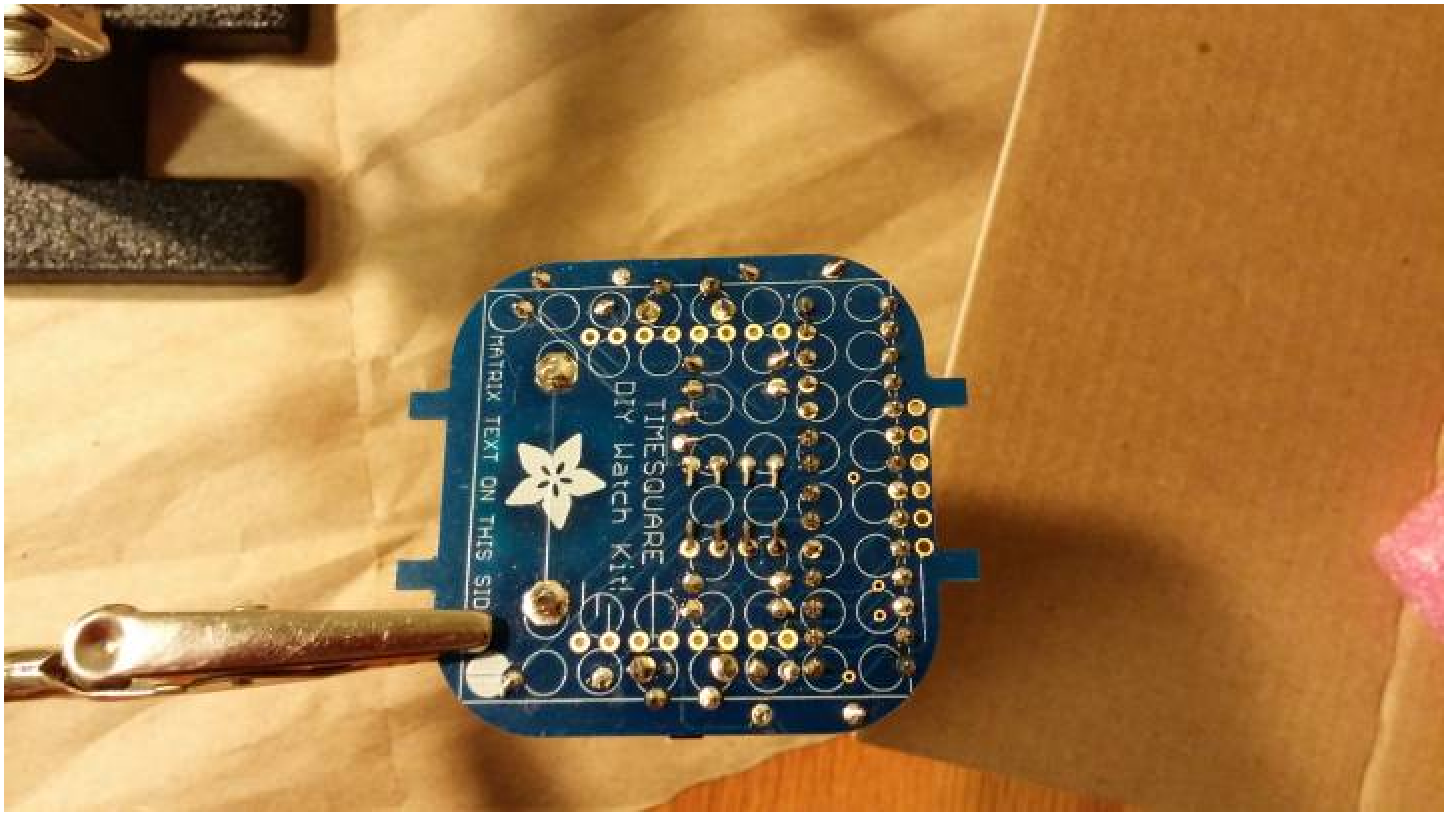


Our second chip – this is the ATMEGA328P. It's basically the CPU, or the brains of our board.

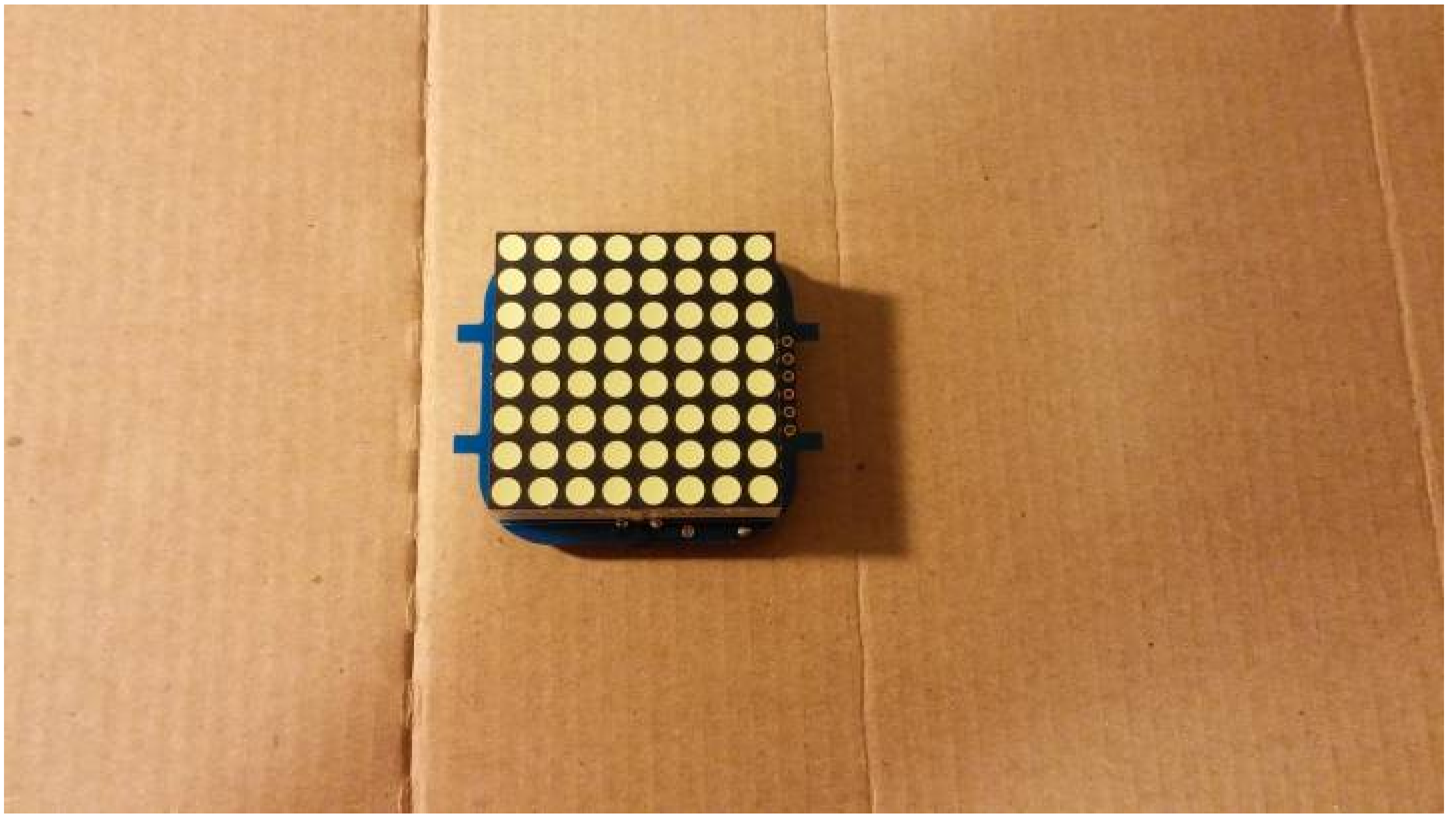


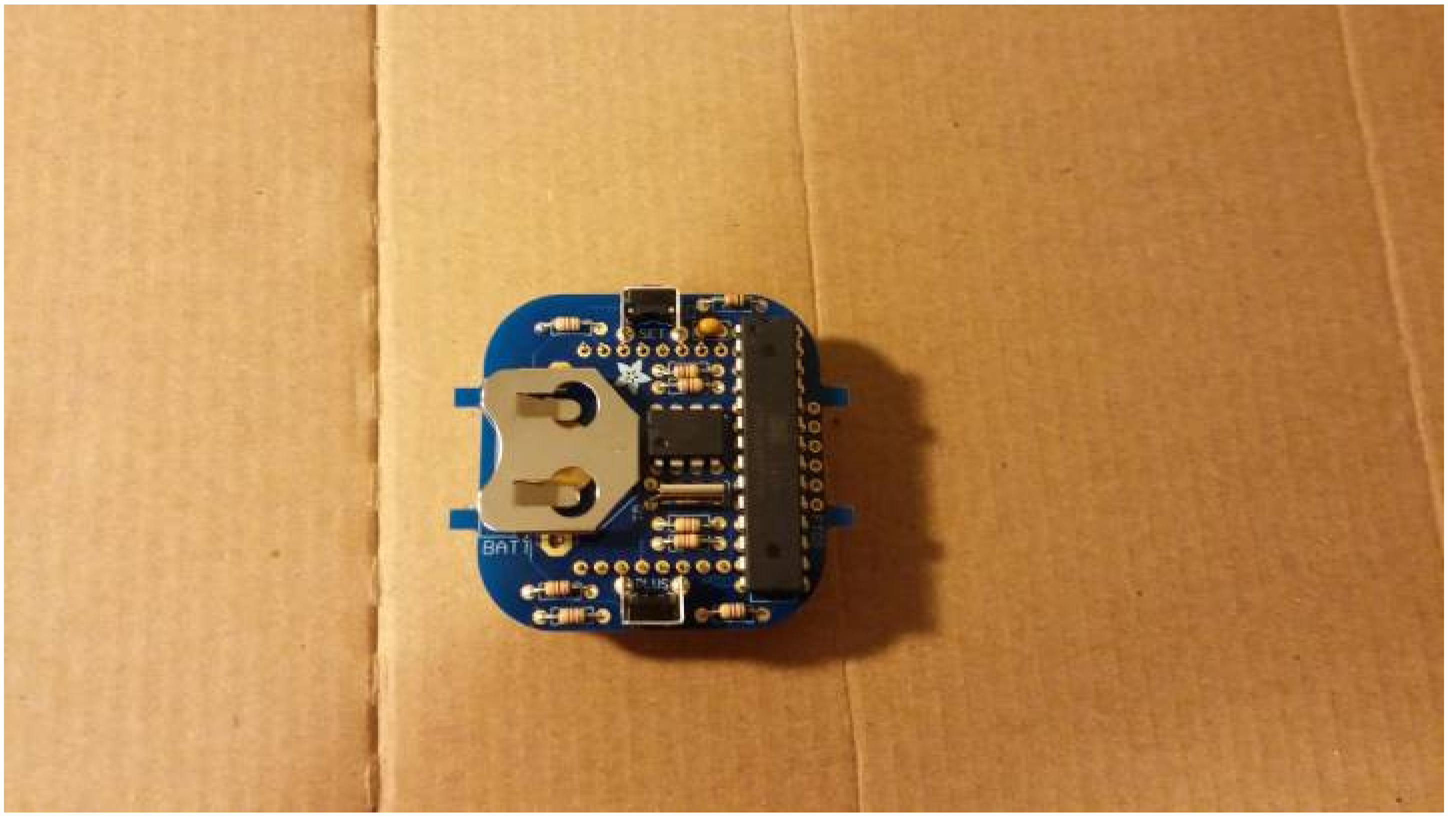
Lots of soldering to do!

The battery holder, timing crystal, and buttons. It’s getting pretty busy on the board!

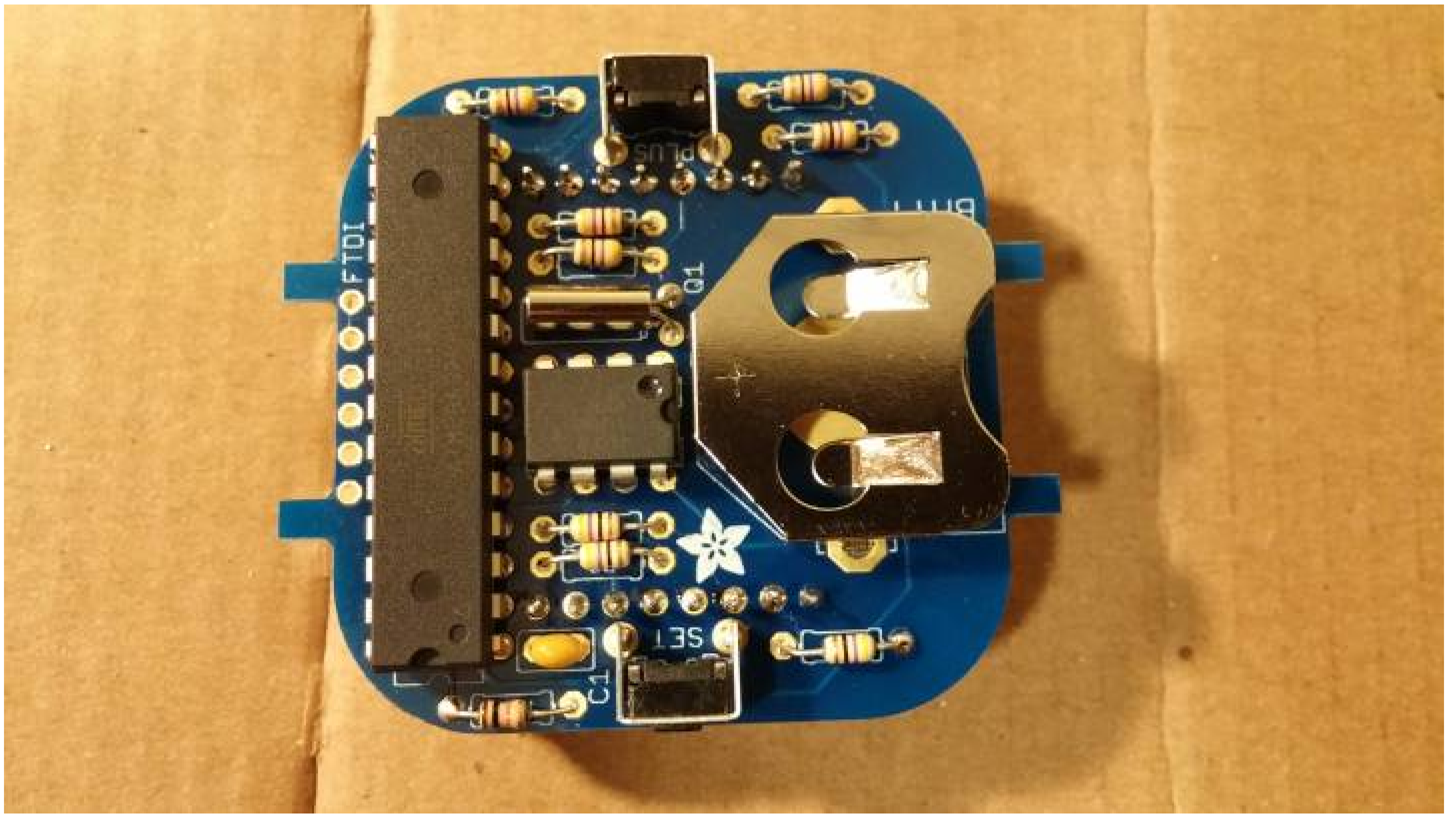


Big contacts for the battery holder, took a while to solder it.

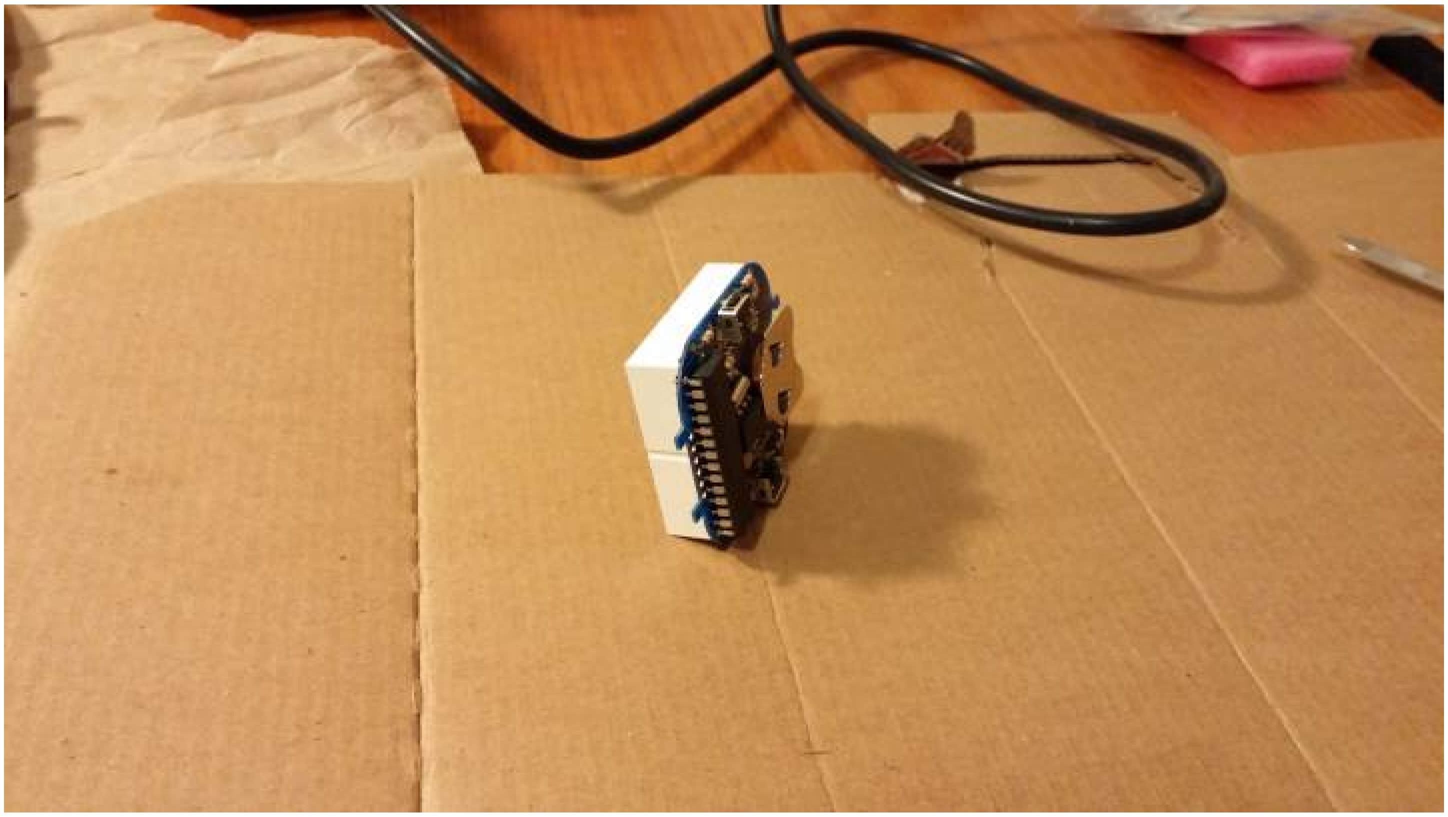

The LED matrix! This one actually goes onto the same side as all of the soldering did – it covers up my messy job!

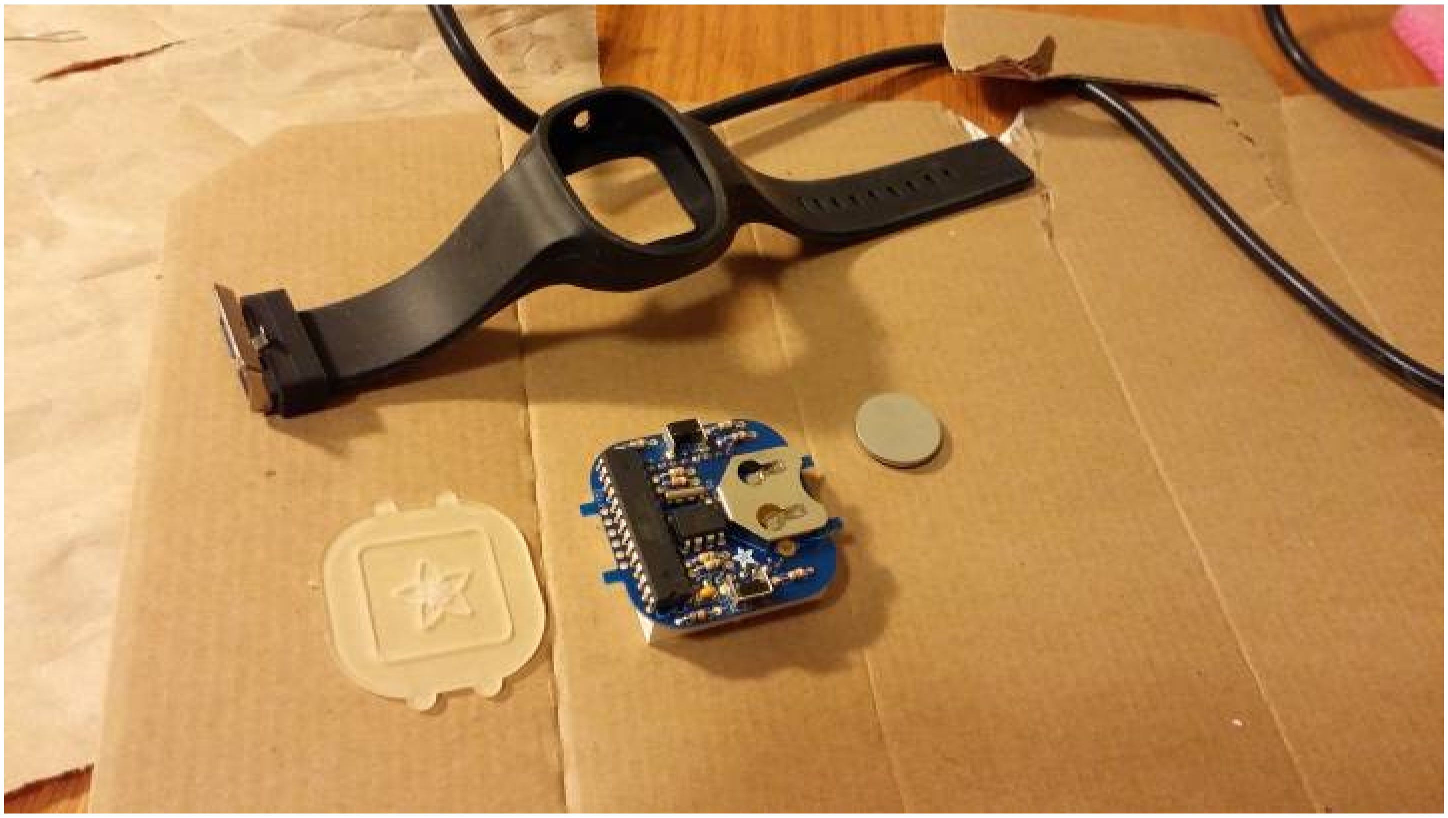


This step was tricky...see those two lines of pins at the top and bottom of the long chip? You have to solder them in without burning anything!



All done, with nothing burned!

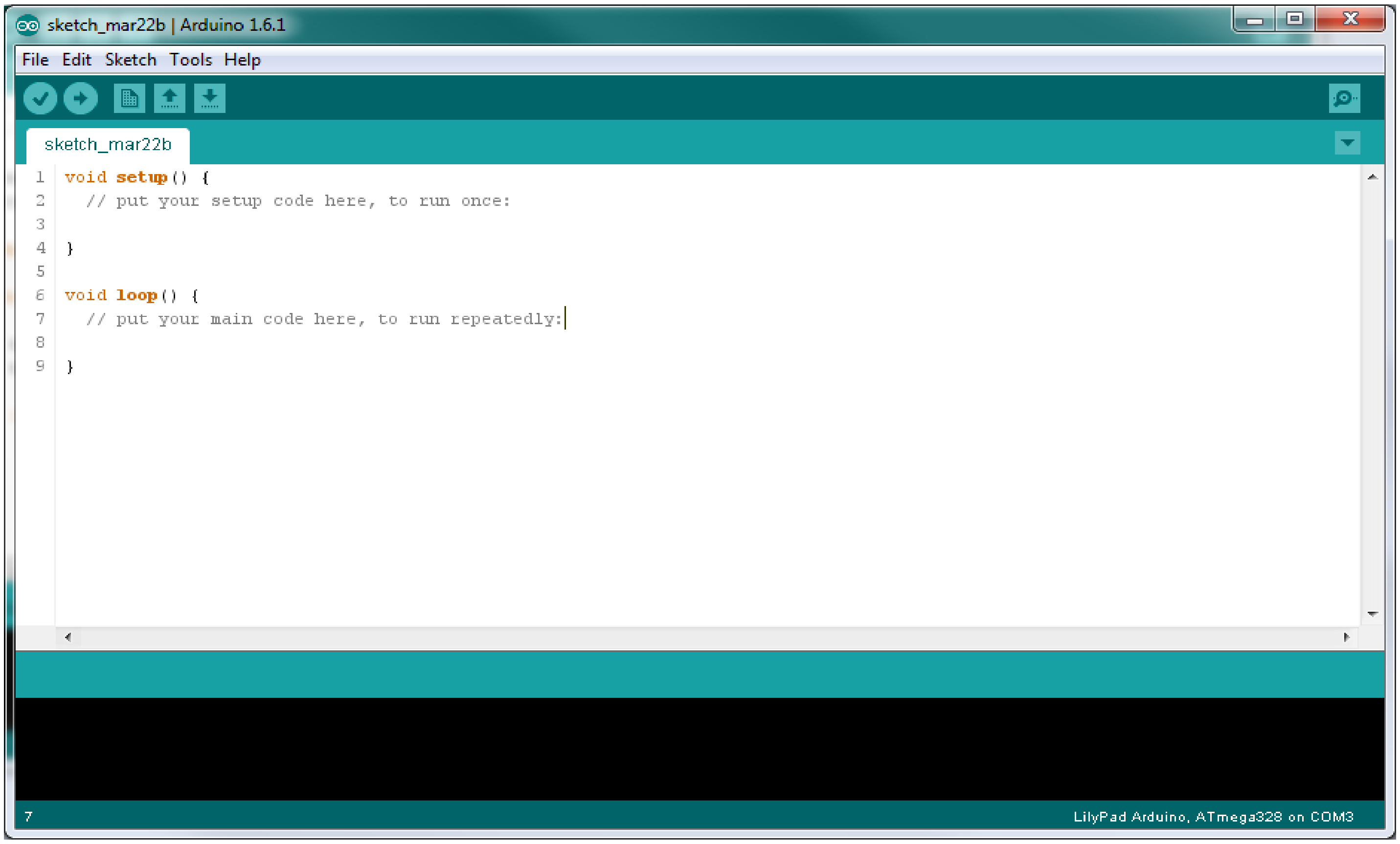

It looks like the chip is wearing a huge backpack...


Now we just need to put the battery in, and slip it into our watch band!

Hooray, it works! It’s asking me to set the date and time.

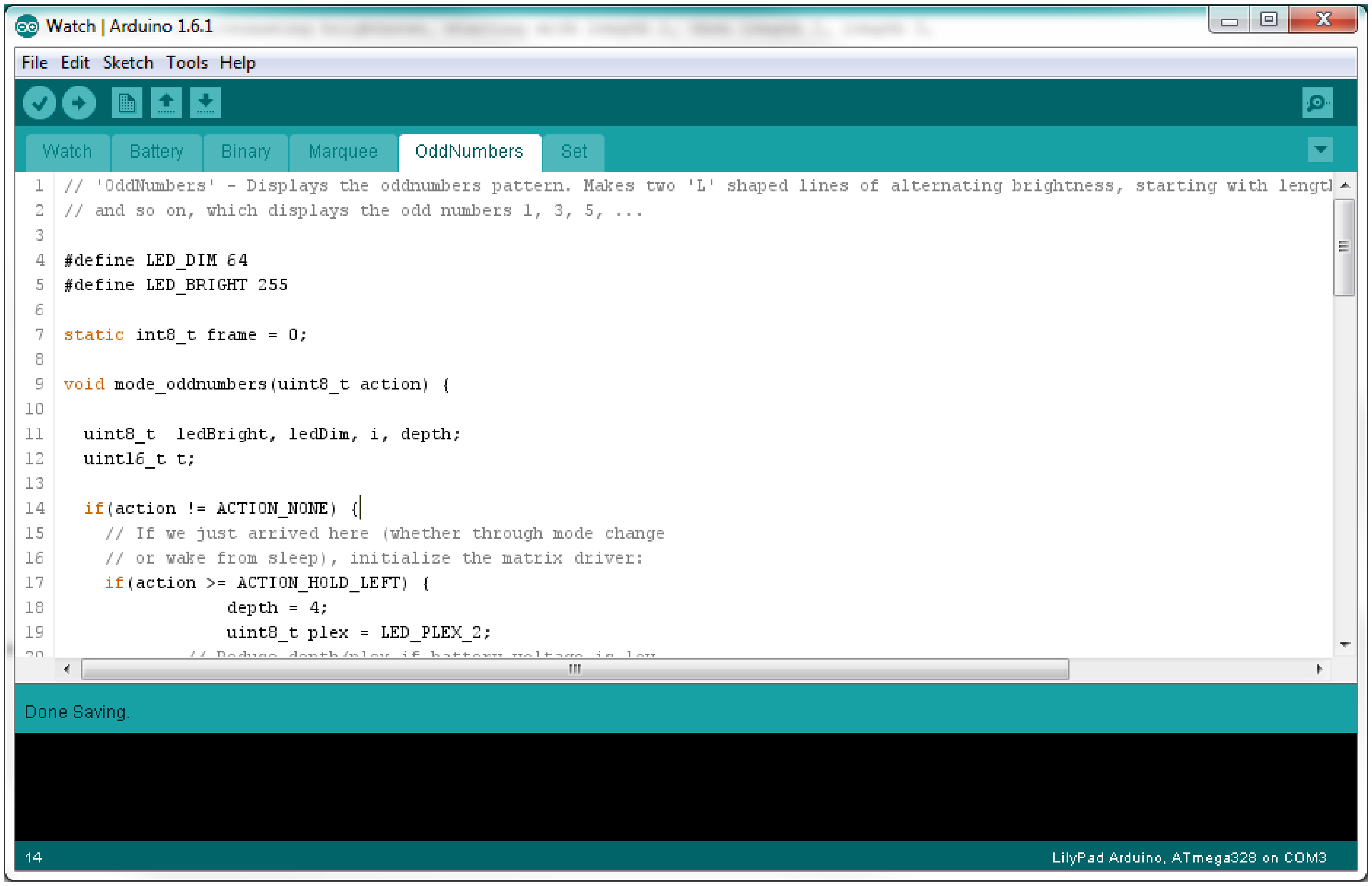
It’s quite bright for such a small thing.

Our FTDI friend – it'll connect our computer to the watch.


sketch_mar22b | Arduino 1.6.1
File Edit Sketch Tools Help
sketch_mar22b
7
LilyPad Arduino, ATmega328 on COM3

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Playing around with the Arduino IDE. This lets us put our own stuff into the watch.

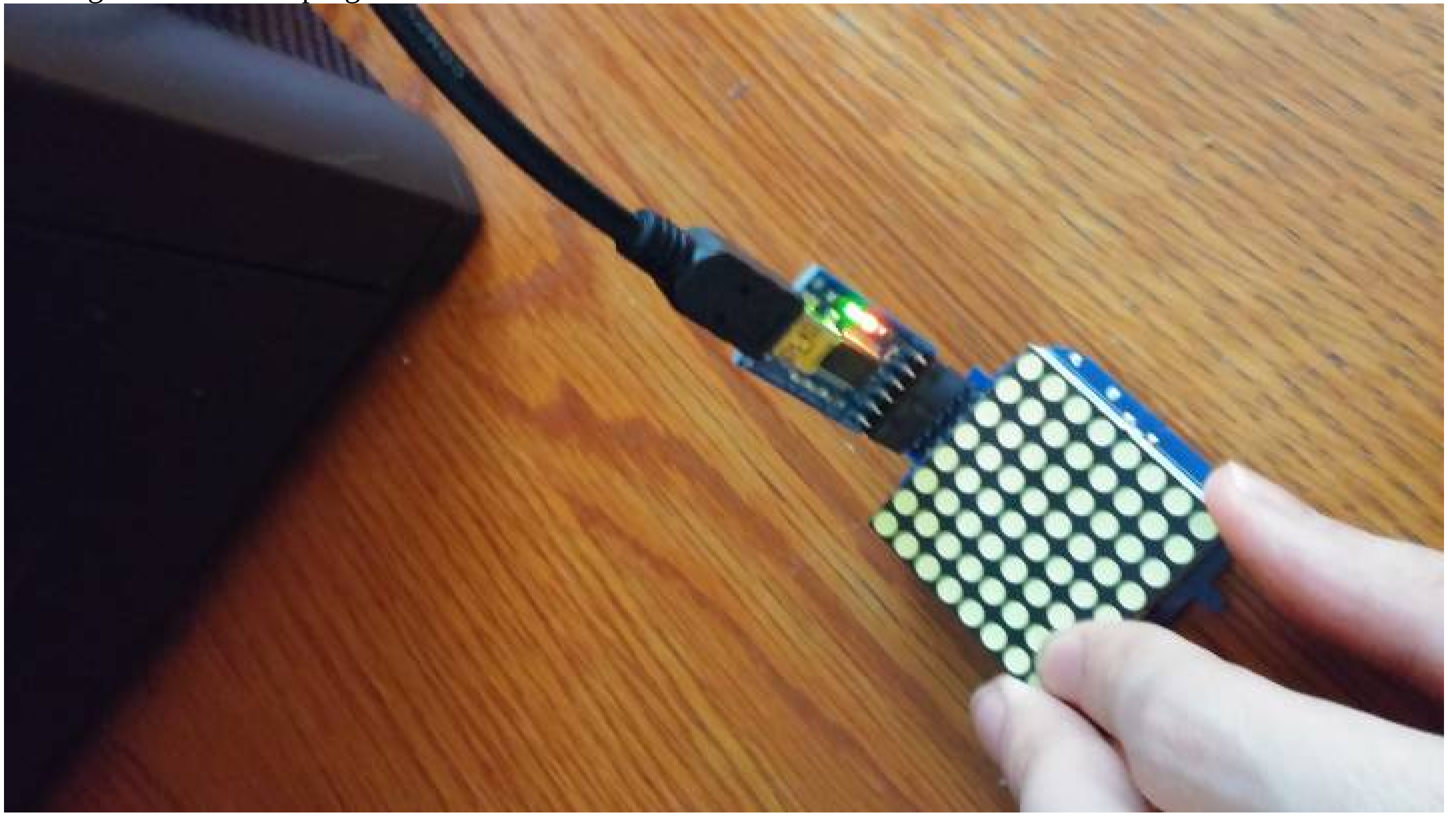


```
// 'OddNumbers' - Displays the oddnumbers pattern. Makes two 'L' shaped lines of alternating brightness, starting with lengt
// and so on, which displays the odd numbers 1, 3, 5, ...

#define LED_DIM 64
#define LED_BRIGHT 255

static int8_t frame = 0;

void mode_oddnumbers(uint8_t action) {

  uint8_t  ledBright, ledDim, i, depth;
  uint16_t t;

  if(action != ACTION_NONE) {
    // If we just arrived here (whether through mode change
    // or wake from sleep), initialize the matrix driver:
    if(action >= ACTION_HOLD_LEFT) {
            depth = 4;
            uint8_t plex = LED_PLEX_2;
```

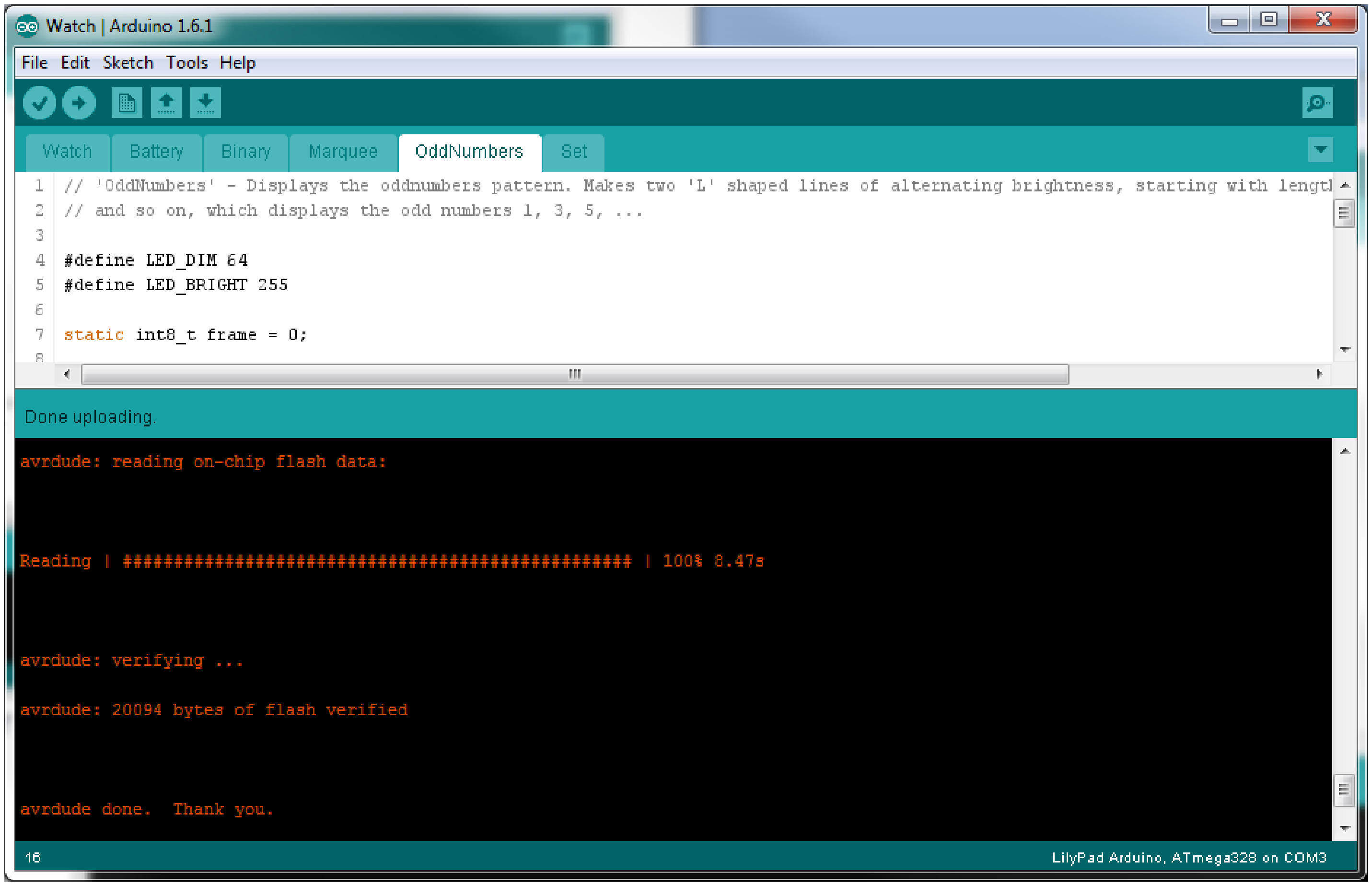
```
Done Saving.
```

```
14                                        LilyPad Arduino, ATmega328 on COM3
```

Making our first mode/program for the watch!



Ready, set, transfer!



```
// 'OddNumbers' - Displays the oddnumbers pattern. Makes two 'L' shaped lines of alternating brightness, starting with lengt
// and so on, which displays the odd numbers 1, 3, 5, ...

#define LED_DIM 64
#define LED_BRIGHT 255

static int8_t frame = 0;
```

Done uploading.

```
avrdude: reading on-chip flash data:


Reading | ################################################## | 100% 8.47s


avrdude: verifying ...

avrdude: 20094 bytes of flash verified


avrdude done.  Thank you.
```

All finished uploading!

Programming the watch

So now we've actually made the watch and we can see it in our hands...how can we hack it and make it do what we want it to? We're going to need just a few things before we can get started:

- FTDI cable OR FTDI Friend (uses a USB connection)
- Arduino Integrated Development Environment (IDE) – You can get it here (http://arduino.cc/en/main/software)

The FTDI cable is what will actually transfer the code that we create onto the watch. The Arduino IDE is where we will be writing our code. Once we've got those two things, we need one more thing before we can get coding – the watch libraries.

Installing the Required Libraries

The libraries are like the tools of a workbench. They provide us with things we need to make sure the watch works the way it's supposed to. In a more technical way, they provide us with the definitions of some procedures we're going to be using (such as drawing pixels, keeping track of the time, and so on). Here's how to install the libraries:

- Visit this website (https://github.com/adafruit/TIMESQUARE-Watch) and click on the 'Download zip' button
- Extract the zip wherever you like and rename the folder to 'Watch' (there should be a watch.cpp and watch.h file inside the renamed folder)
- Move the watch folder into the \Arduino\Libraries folder. This will depend on where you've installed Arduino, typically it will be in either C:\Program Files (x86)\Arduino or C:\Program Files\Arduino
- You're done! You can verify that you've installed the library correctly by opening up the Arduino IDE, and going to Sketch→Import Library and finding Watch in the list there. If it isn't there, then there was a problem installing the libraries – try again from the beginning step

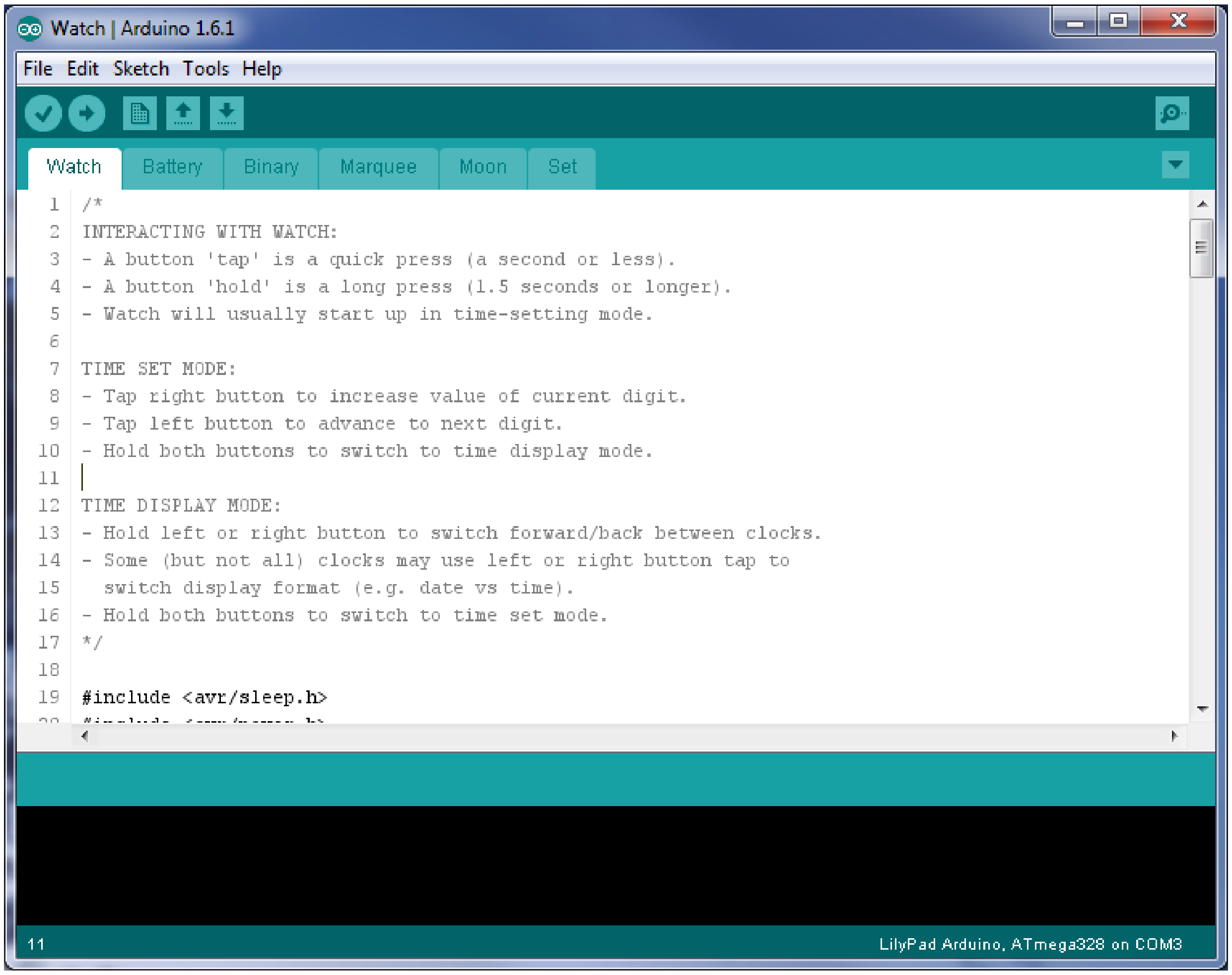
There are two other required libraries, both installed the same way as the previous (except you don't have to rename the folder to watch – just unzip it and move it into your \Arduino\Libraries folder)

- RTCLib (https://github.com/adafruit/RTClib) – this is a library for keeping time
- Adafruit_GFX (https://github.com/adafruit/Adafruit-GFX-Library) – this library is for illuminating the matrix

You can check that the libraries are installed the same way we did for the Watch library – look inside Sketch → Import Library and make sure Adafruit_GFX and RTCLib are listed there.

Getting Started

Now that we have all our tools ready, let's get into the programming! Launch the Arduino IDE and look for a file 'watch.ino' in the watch library (Arduino\Libraries\Watch\examples\Watch\watch.ino). Open it up and we should get another window with all of the watch things that looks like this:


Watch | Arduino 1.6.1
File Edit Sketch Tools Help
Watch
Battery
Binary
Marquee
Moon
Set
11
LilyPad Arduino, ATmega328 on COM3

```
/*
INTERACTING WITH WATCH:
- A button 'tap' is a quick press (a second or less).
- A button 'hold' is a long press (1.5 seconds or longer).
- Watch will usually start up in time-setting mode.

TIME SET MODE:
- Tap right button to increase value of current digit.
- Tap left button to advance to next digit.
- Hold both buttons to switch to time display mode.

TIME DISPLAY MODE:
- Hold left or right button to switch forward/back between clocks.
- Some (but not all) clocks may use left or right button tap to
  switch display format (e.g. date vs time).
- Hold both buttons to switch to time set mode.
*/

#include <avr/sleep.h>
```

The watch comes with all of this code already on the chip – what you're seeing is the stuff that makes the watch works out of the box. There are six different .ino files which all have a different role – the battery, binary, marquee, and moon files are all for the different 'modes' of the watch (when you cycle left or right through the watch). Each one of those files contains the logic for how that mode is supposed to work.

The set file contains the logic required to set the time of the watch (the set screen when you hold down both the left and the right buttons). We don't want to play with this file – it's pretty important for making the watch function!

Finally, the watch file is it a bit like the binder that keeps all of our code together and organized. It contains the logic for switching modes, starting the watch up the first time you power it on, and some other bookkeeping things. This file is where we are going to start.

Watch.ino

If we want to add our own mode, the first thing we need to learn is how to access it. Watch.pde takes care of switching modes, so we'll need to modify it slightly. Imagine the modes as being a circle of people. From a given mode, you can go to the left or to the right where you are, and you can keep going in one direction to circle all the way around. Let's see how the code does this:

```
void loop() {
uint8_t a = watch.action();
...
else if(a == ACTION_HOLD_RIGHT) {
    if(mode != MODE_SET) {
      // Switch to next display mode (w/wrap)
      if(++mode >= N_MODES) mode = 1;
    }
  }
...

}
```

This function stores any actions into a variable ‘a’, and then checks what the action was. In the case above, if the action was the user holding the right button, then as long as we aren’t in the set mode we would switch to the next mode.

```
if(++mode >= N_MODES) mode = 1;
```

This line does the real work for us. First it increases the current mode by 1 (++mode). Then it checks, is the number that I get after increasing the mode by 1 greater than the number of modes I have? If it is, then I’ve reached the end of my circle and need to loop around, so set the mode back to 1.

Now that we know how modes work, there are only two steps we need to do in order to add a new mode for the watch to cycle through. At the top of the watch.pde file, there is a list of definition for each mode:

```
#define MODE_SET     0
#define MODE_MARQUEE 1
#define MODE_BINARY  2
#define MODE_MOON    3
#define MODE_BATTERY 4
```

These give each mode a numerical value. Add a new one at the bottom and give it a number of 5:

#define MODE_OURMODE 5

We could of course give it whatever name we want, but it’s a good idea to keep the same naming convention as they have it. Just below these definitions there is an array of modes:

```
void (*modeFunc[])(uint8_t) = {
  mode_set,
  mode_marquee,
  mode_binary,
  mode_moon,
  mode_battery
};
```

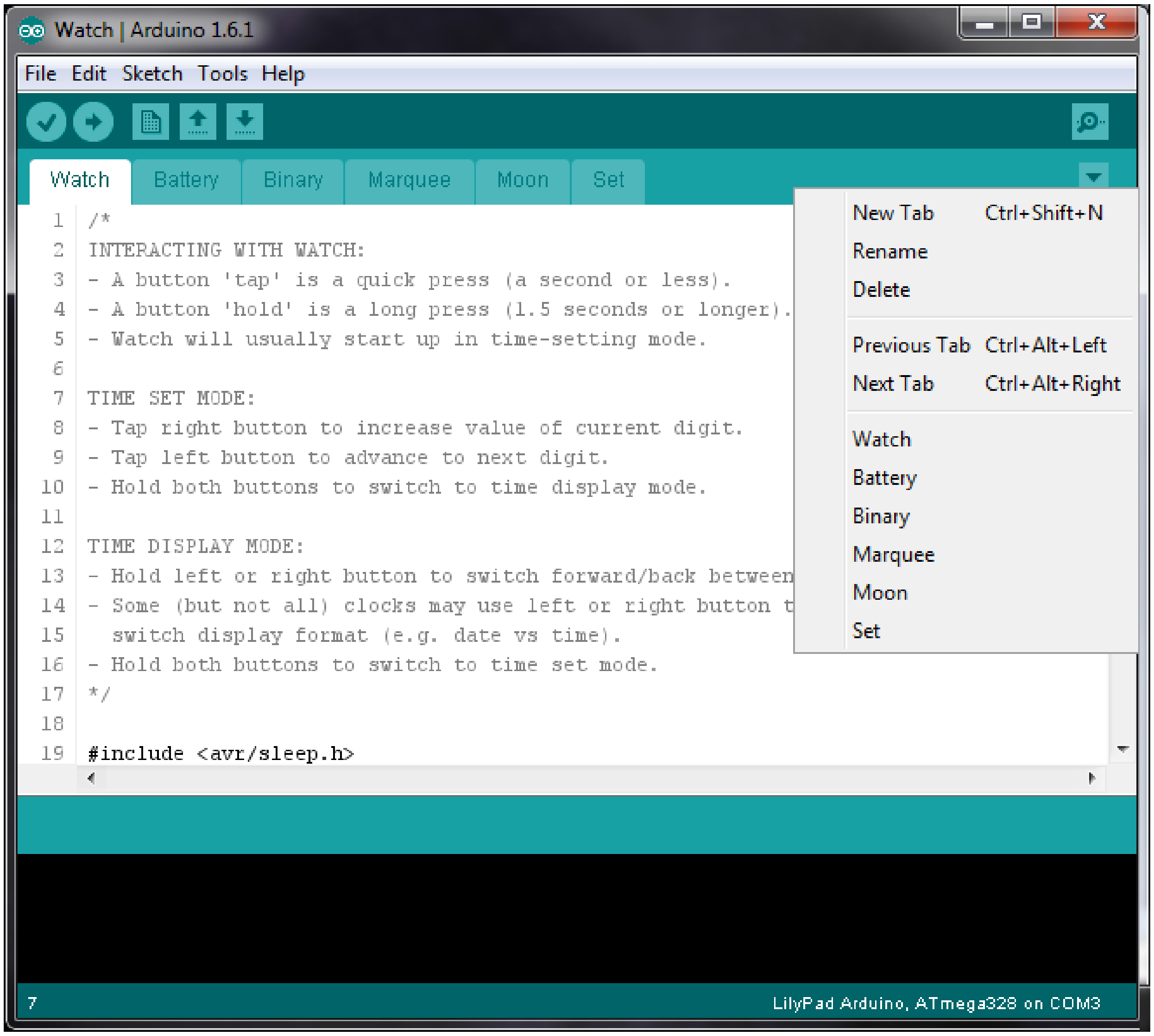
This array tells the watch how many modes there are, and what the order is. We want to add our mode to the end there, so put a comma after mode_battery, and add our new mode so that it looks like this:

```
void (*modeFunc[])(uint8_t) = {
  mode_set,
  mode_marquee,
  mode_binary,
  mode_moon,
  mode_battery,
  mode_ourmode
};
```

Now the watch knows that it has another mode, namely mode_ourmode, which it has to cycle through. At this point, the watch knows that there is another mode called mode_ourmode, but we haven't actually made it yet, so that's our next step!

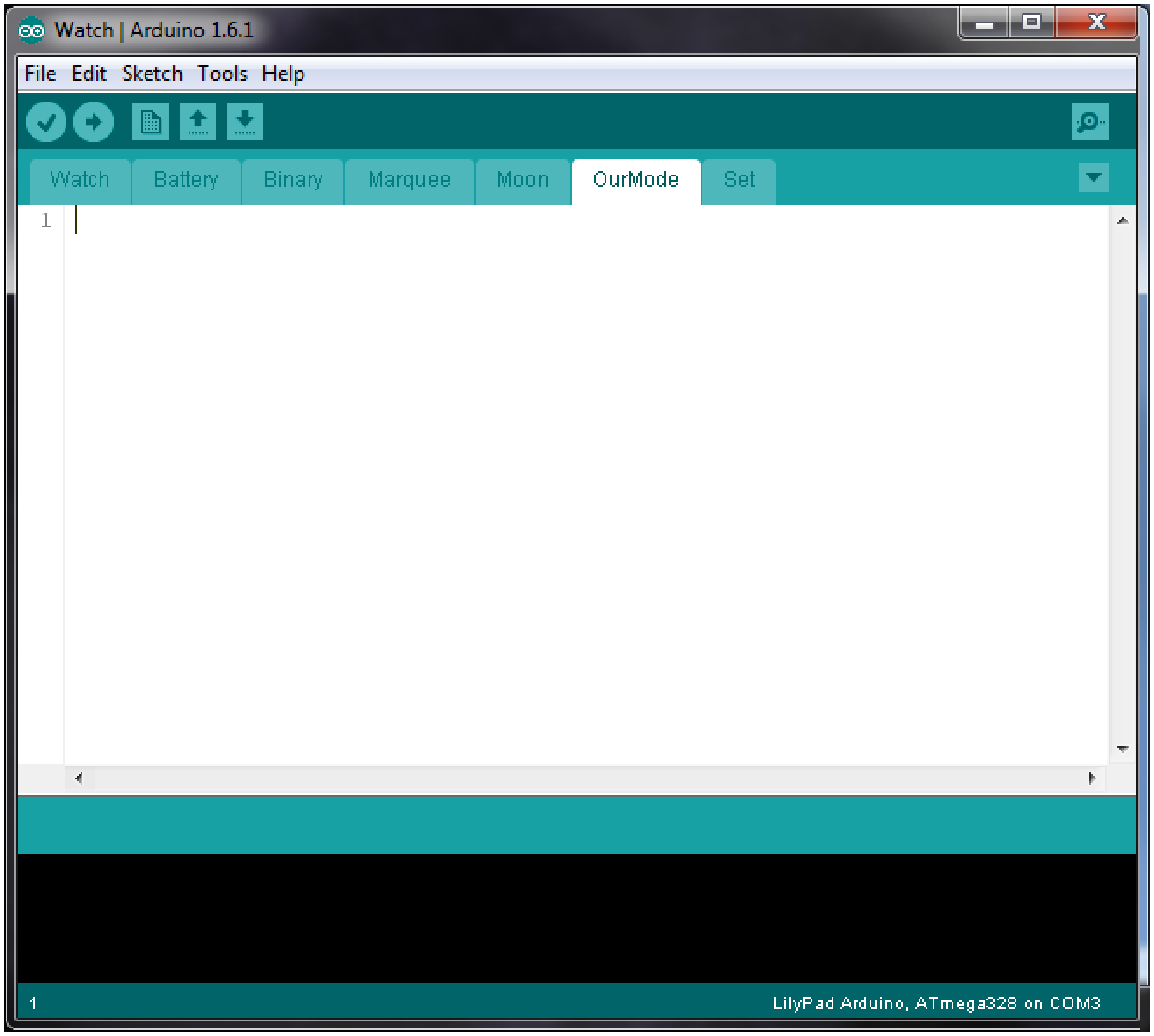
OurMode.ino

We've told the watch we have a new mode...now we actually need to write it! At the top right, there is a down arrow. Click on it and select 'New Tab'. It'll ask you to name the file at the bottom of the screen – let's call it OurMode.



```
/*
INTERACTING WITH WATCH:
- A button 'tap' is a quick press (a second or less).
- A button 'hold' is a long press (1.5 seconds or longer).
- Watch will usually start up in time-setting mode.

TIME SET MODE:
- Tap right button to increase value of current digit.
- Tap left button to advance to next digit.
- Hold both buttons to switch to time display mode.

TIME DISPLAY MODE:
- Hold left or right button to switch forward/back between
- Some (but not all) clocks may use left or right button t
  switch display format (e.g. date vs time).
- Hold both buttons to switch to time set mode.
*/

#include <avr/sleep.h>
```

You should now have a blank page open for the tab ‘OurMode’, and it should look something like this:


Watch | Arduino 1.6.1
File Edit Sketch Tools Help
Watch
Battery
Binary
Marquee
Moon
OurMode
Set
LilyPad Arduino, ATmega328 on COM3

This page is where all of our instructions for the watch are going to go. Whatever we want the watch to do for our newly created mode needs to go in here. Let's start off with some basics:

```
void mode_ourMode(uint8_t action) {

}
```

This is the function that holds all of the information for our mode. While the watch.ino file handles cycling through modes (i.e. holding down the left or right buttons), we can process other actions here, namely just pressing (instead of holding) the left and right buttons. That's what uint8_t action is – the unit8_t is a number that represents what action occurred. Of course, we need to do something with that action. Let's add in some more code:

```
void mode_ourMode(uint8_t action) {

// If we have some kind of action
if(action != ACTION_NONE) {
   // If we got here either through someone holding left/right or
   // the watch woke up here
   if(action >= ACTION_HOLD_LEFT)
```

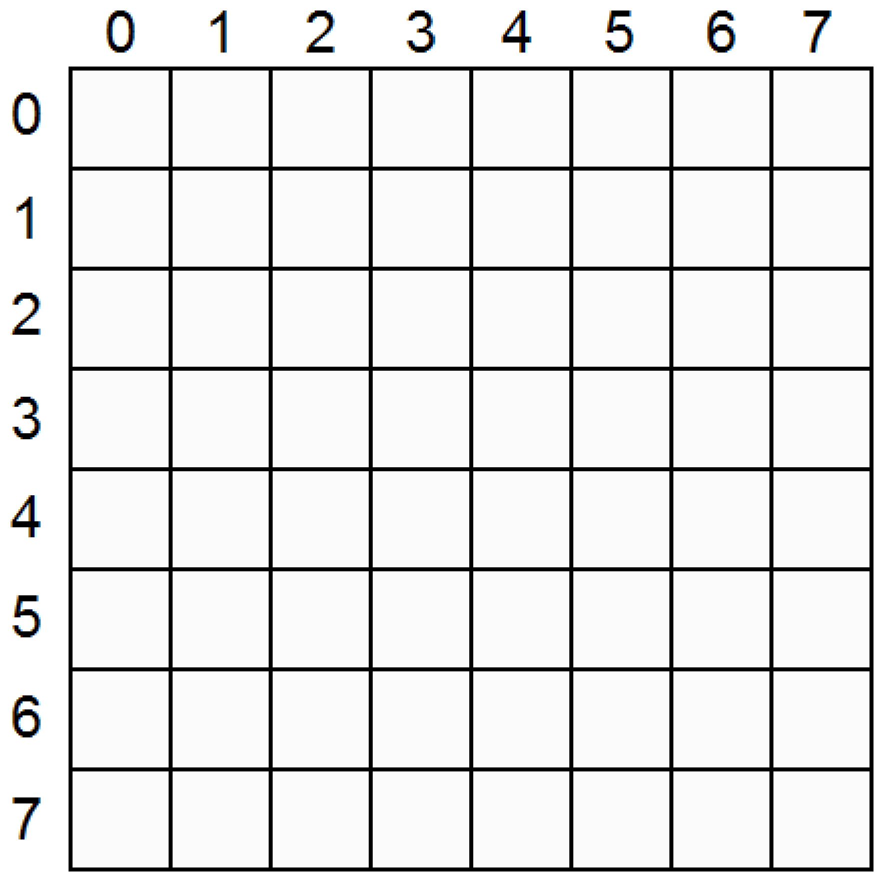
```
    {
      // Set the depth and plex
      uint8_t depth = 4, plex = LED_PLEX_2;
      // Reconfigure display if needed
      if((watch.getDepth() != depth) || (watch.getPlex() != plex))
        fps = watch.setDisplayMode(depth, plex, true);
    }
  }
}
```

Most of this stuff is just initialization required. When we get to the mode, we need to set the depth and plex, which is essentially how bright the LEDs are. Different modes will have different settings, so we need set the depth and plex to our setting if it isn't already.

Now that we have the initialization out of the way, let's make our mode actually do something. The simplest thing we can do is just to light up a pixel on our watch, so let's start with that. The function we are going to use to draw a single pixel is the following:

```
watch.drawPixel(x, y, colour);
```

Though the board only has one colour, the intensity can be controlled with the colour parameter. The position of the pixel is determined through (x,y) coordinates like a graph, with the top left square being (0,0).



So if we want to light up the corner pixels, we would just have to write in four lines of code:

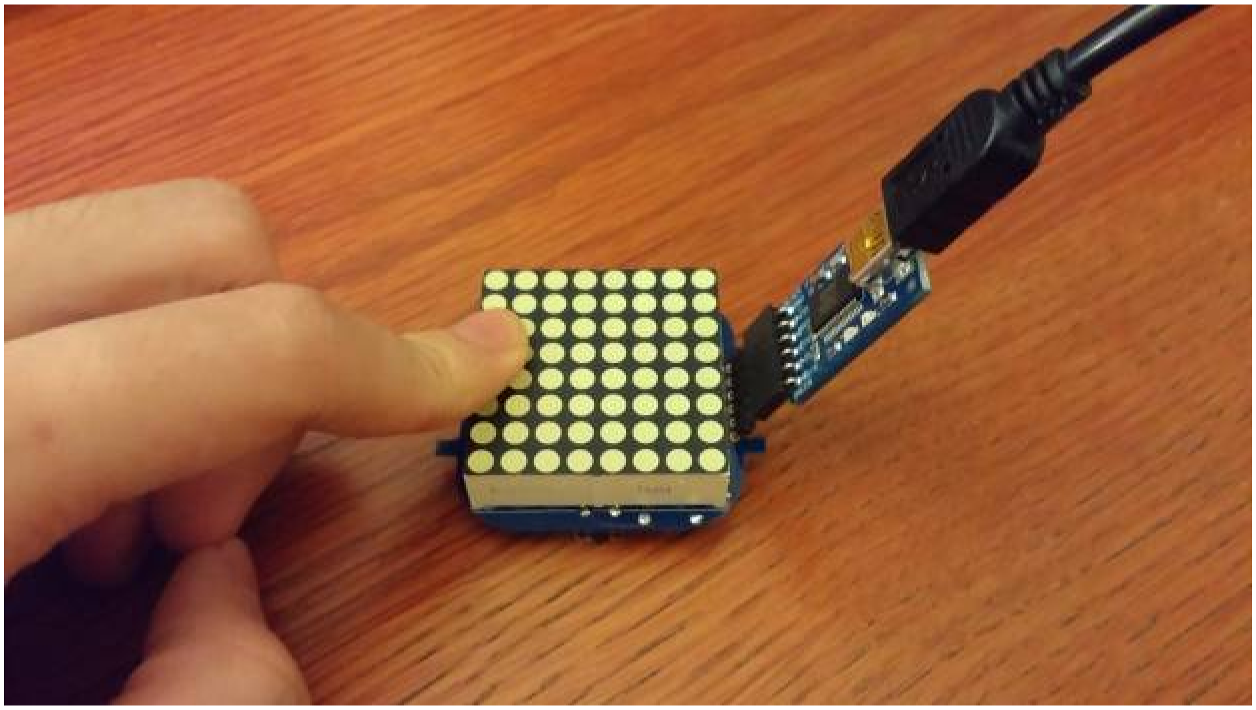
```
void mode_ourMode(uint8_t action) {
```

```
// If we have some kind of action
if(action != ACTION_NONE) {
    // If we got here either through someone holding left/right or
    // the watch woke up here
    if(action >= ACTION_HOLD_LEFT)
    {
      // Set the depth and plex
      uint8_t depth = 4, plex = LED_PLEX_2;
      // Reconfigure display if needed
      if((watch.getDepth() != depth) || (watch.getPlex() != plex))
        fps = watch.setDisplayMode(depth, plex, true);
    }
  }
watch.drawPixel(0, 0, 1);
watch.drawPixel(0, 7, 2);
watch.drawPixel(7, 0, 3);
watch.drawPixel(7, 7, 4);
}
```

For choosing a value for the colour (intensity), higher values will give you a brighter pixel (up to about 8 or so). In this case, the top left pixel will be the faintest while the bottom right will be the brightest.

Great, we have all the code we need. First let's compile it – hit the checkmark at the top left of the IDE to compile (or CTRL-R). This turns our code from English into something that the watch can understand. Once it says we are done compiling, we can upload it onto our watch.

Make sure your FTDI cable or FTDI Friend is plugged into your computer and ready to go. If you're having problems with the FTDI Friend, check out the 'Troubleshooting the FTDI Friend' section. Put the pins from your FTDI friend or cable into the respective holes on the watch – make sure it looks like the picture below! If you have the watch or pins in the wrong alignment, the upload won't work!



Once it's physically connected, click the right arrow next to the checkmark at the top left of the IDE to upload your code onto the watch (or CTRL-U). You'll see some status messages at the bottom, and the lights on the FTDI Friend should start blinking. Make sure you keep the connection together until the upload is finished!

Once finished, hold the left button until the mode changes – it should go straight to ours since it's the last mode. You should see the four lit pixels just as we programmed. Congratulations! We just hacked the watch and made our very own mode. Granted, it doesn't do a whole lot right now, but soon we'll get into the more interesting things we can do.

It would be tedious to have to turn on and off every pixel individually. Thankfully, the Adafruit_GFX library contains some functions that allow us to draw more than one pixel at a time such as a line, but also things like rectangles, triangles, even circles! If you're curious about all the things the Adafruit_GFX library can do, you can look it up in the Arduino\Libraries\Adafruit_GFX folder. The Adafruit_GFX.h file is the header file that contains all the function declarations (i.e. what the function is called, what parameters it takes in, and what it returns). The Adafruit_GFX.cpp contains the actual implementation, though it can be hard to decipher.

Here are some useful commands for drawing things:

watch.drawLine(x_0, y_0, x_1, y_1, colour)
//x_0 and y_0 are the start point of the line, and x_1 and y_1 are the end points

```
watch.drawRect(x, y, width, height, colour)
// x and y are the top left point in the rectangle, and it extends
// width-1 to the right and height-1 down

watch.fillRect(x, y, width, height, colour)
// similar to above, but fills in the rectangle too

watch.fillScreen(colour)
// an easy way to clear the matrix, just call watch.fillScreen(0);
```

Remember, we only have one colour for our matrix. It is an integer value from 0 up, which increases the intensity as the integer increases (to about 8). Zero means the LED is off.

Static Variables
Inside of our watch there is a little timer that runs whenever the watch is active (a few seconds after you press a button, until the screen goes dark). Every time this timer runs up, the watch checks what mode it's in, and executes all of the code inside that mode. This happens very quickly...about thirty times a second or so. This is sometimes called the refresh rate, usually given in Hertz (Hz). Our watch refreshes at about 30 times a second, so its refresh rate would be 30 Hz.

The code for our mode is relatively simple – all we have done so far is light up the four corners with varying intensities. Every time the watch executes our code again, nothing changes – the display stays the same from start to finish. Let's try an example where we can see a change for every refresh.

```
void mode_ourMode(uint8_t action) {

// If we have some kind of action
if(action != ACTION_NONE) {
   // If we got here either through someone holding left/right or
   // the watch woke up here
   if(action >= ACTION_HOLD_LEFT)
```

```
    {
      // Set the depth and plex
      uint8_t depth = 4, plex = LED_PLEX_2;
      // Reconfigure display if needed
      if((watch.getDepth() != depth) || (watch.getPlex() != plex))
        fps = watch.setDisplayMode(depth, plex, true);
    }
  }
  watch.fillScreen(0);
  watch.drawPixel(random(0,8), random(0,8), 7);

}
```

We added a watch.fillScreen(0), which just clears the display. This means that every time the watch refreshes the screen, it will turn off all of the LED lights. The next line is similar to our previous drawPixel lines, but we've added a 'random(0,8)' in place of the fixed numbers for each x and y coordinate. This function will select a random number from 0-7 (including zero and seven, but not eight). So for every frame that is rendered, every LED will be turned off, and then a random LED will be turned on. The next frame will turn that LED off, and pick a new random LED to turn on, and so on. Upload it again and see how it works!

You'll notice that is *extremely* fast. You can barely see the pixels as they light up then go away! If we want to slow it down then we shouldn't pick a new LED to light up every time it refreshes, but maybe every fourth or eighth time it refreshes...but how do we tell the watch to only pick a new LED every eighth refresh?

When the code inside the function is executed, all of the variables inside are reset, so if we want to keep track of something throughout all of our refreshes, we need to use what's called a **static** variable. A static variable is only ever instantiated once, and can 'live' outside of a function. Let's take a look at some examples.

```
void mode_ourMode(uint8_t action) {

// If we have some kind of action
if(action != ACTION_NONE) {
    // If we got here either through someone holding left/right or
    // the watch woke up here
    if(action >= ACTION_HOLD_LEFT)
    {
      // Set the depth and plex
      uint8_t depth = 4, plex = LED_PLEX_2;
      // Reconfigure display if needed
      if((watch.getDepth() != depth) || (watch.getPlex() != plex))
        fps = watch.setDisplayMode(depth, plex, true);
    }
  }

int x = 0;
for (int i = 0; i <= x; i++)
{
  watch.drawPixel(i, 0, 7);
}
x = x + 1;

}
```

In this example, we have a variable ‘x’ inside of our function. It starts at zero, so the first time the function is called, the for loop only goes up to zero (since x is zero), so we draw a pixel at (0,0). The line after the loop says that we should increase the value of x by 1...so the next time the watch refreshes and goes into our function again, we should draw the pixel at (1,0)...right?

If you try it and see, you’ll notice that the LED at (1,0) never lights up, only the top left LED does. Because our variable here isn’t static, every time the watch refreshes and goes into our function, it sets x back to zero so the drawPixel() will only ever draw (0,0). Let’s try this again, but this time we’ll use a static variable.

```
static int x = 0;
void mode_ourMode(uint8_t action) {

// If we have some kind of action
if(action != ACTION_NONE) {

    // Reset our static variable to zero
    x = 0;

    // If we got here either through someone holding left/right or
    // the watch woke up here
    if(action >= ACTION_HOLD_LEFT)
    {
      // Set the depth and plex
      uint8_t depth = 4, plex = LED_PLEX_2;
      // Reconfigure display if needed
      if((watch.getDepth() != depth) || (watch.getPlex() != plex))
        fps = watch.setDisplayMode(depth, plex, true);
    }
  }

for (int i = 0; i <= x; i++)
{
  watch.drawPixel(i, 0, 7);
}
x = x + 1;

}
```

Note this time that at the very start of our code, we have a static int at defined before our function. This variable will persist through function calls, so we can store a number here and it’ll keep it in memory for us. Also note that whenever some kind of action occurs (that either wakes the watch or switches to this mode), we need to reset the static variable to zero.

This time, the code does indeed draw the entire line, although it does it quite quickly. Let’s see how we can slow things down using static variables.

```
static int x = 0;
static int frame = 0;
void mode_ourMode(uint8_t action) {

  // If we have some kind of action
  if(action != ACTION_NONE) {
    // Reset our static variable to zero
    x = 0;
    frame = 0;
    // If we got here either through someone holding left/right or
    // the watch woke up here
    if(action >= ACTION_HOLD_LEFT)
    {
      // Set the depth and plex
      uint8_t depth = 4, plex = LED_PLEX_2;
      // Reconfigure display if needed
      if((watch.getDepth() != depth) || (watch.getPlex() != plex))
        fps = watch.setDisplayMode(depth, plex, true);
    }
   }

  // Clear the screen before each draw
  watch.fillScreen(0);

  // We draw every frame, up to whatever x is at the moment
  for (int i = 0; i <= x; i++)
  {
    watch.drawPixel(i, 0, 7);
  }

  // We only update the value of x every eighth frame
  if (frame >= 8)
  {
    x = x + 1;
    // Since we updated this frame, reset the frame counter to zero
    frame = 0;
  }
  // If we didn't update, then increase the frame counter by one
  else
  {
    frame++;
  }

}
```
We’ve added a new static variable called *frame*. Instead of telling the watch to draw the next pixel every time we go through the function, we use the variable frame to count the number of frames that have passed since last time we drew. Every eighth frame, we draw the appropriate pixels and then increase the x variable by 1 and reset the frame counter to zero. If we didn’t draw this frame, we increase the frame counter by 1. The frame counter will keep increasing until it gets to 8, at which point we will draw again and then reset.

We’ve also added a new line just before we do any drawing: watch.fillScreen(0).This function simply clears the screen so that if there were pixels that were lit up from previously, they will be shut off. In this case, the pixels don’t need to disappear so it won’t matter too much here, but we’ll see in the next example that this line is important.

‘Moving’ Pixels

With all of this in mind, how can we create a ‘moving’ pixel or image? The pixels obviously can’t move, but we can create the illusion of movement by creating two different images in quick succession, much like on a television or monitor. Let’s take a look:

```
static int x = 0;
static int frame = 0;
void mode_ourMode(uint8_t action) {

  // If we have some kind of action
  if(action != ACTION_NONE) {
      // Reset our static variable to zero
      x = 0;
      frame = 0;
      watch.fillScreen(0);
      // If we got here either through someone holding left/right or
      // the watch woke up here
      if(action >= ACTION_HOLD_LEFT)
      {
        // Set the depth and plex
        uint8_t depth = 4, plex = LED_PLEX_2;
        // Reconfigure display if needed
        if((watch.getDepth() != depth) || (watch.getPlex() != plex))
          fps = watch.setDisplayMode(depth, plex, true);
      }

    // Reset sleep timeout on ANY button action
    watch.setTimeout(fps * 5);
    }

  // Clear the screen before each draw
  watch.fillScreen(0);

  // We draw every frame, but only the value of x (not up to it)
  watch.drawPixel(x, 0, 7);

  // We only update the value of x every eighth frame
  if (frame >= 8)
  {

    x = x + 1;
    // Since we updated this frame, reset the frame counter to zero
    frame = 0;
  }
  // If we didn't update, then increase the frame counter by one
  else
  {
    frame++;
  }

}
```

This time instead of drawing all the pixels in the line, we only draw one pixel at any time. When we update the value of x, we move the pixel over. This gives ‘movement’ to the pixel – it looks like it’s moving right along the row.

We also added a new section of code: watch.setTimeout(fps * 5). This sets the time before the watch goes back to sleep. If you find that your code needs a little more time to show off, you can set the timeout timer to something higher (e.g., fps * 10). Just be careful not to set too high – we don’t want to drain the battery more than necessary!

Don’t want to draw pixels individually? You can draw lines of pixels as well! The function watch.drawLine(x_1, y_1, x_2, y_2, colour) will draw a line start from position (x_1, y_1) to (x_2, y_2)

```
static int x = 0;
static int frame = 0;
void mode_ourMode(uint8_t action) {

  // If we have some kind of action
  if(action != ACTION_NONE) {
     // Reset our static variable to zero
     x = 0;
     frame = 0;
     watch.fillScreen(0);
     // If we got here either through someone holding left/right or
     // the watch woke up here
     if(action >= ACTION_HOLD_LEFT)
     {
       // Set the depth and plex
       uint8_t depth = 4, plex = LED_PLEX_2;
       // Reconfigure display if needed
       if((watch.getDepth() != depth) || (watch.getPlex() != plex))
         fps = watch.setDisplayMode(depth, plex, true);
     }

    // Reset sleep timeout on ANY button action
    watch.setTimeout(fps * 5);
    }

  // Clear the screen before each draw
  watch.fillScreen(0);

  // We draw every frame, but only the value of x (not up to it)
  watch.drawLine(x, 0, x, 7, 7);

  // We only update the value of x every eighth frame
  if (frame >= 8)
  {

    x = x + 1;
    // Since we updated this frame, reset the frame counter to zero
    frame = 0;
  }
  // If we didn't update, then increase the frame counter by one
  else
  {
    frame++;
  }

}
```

Troubleshooting the FTDI Friend

If you can't seem to upload to your watch using the FTDI friend, we might need to install some drivers to get it to work. Head over here (http://www.ftdichip.com/Drivers/VCP.htm) and grab the correct driver for your system (probably x86 or x64 for Windows, depending on which version you have). Install the driver and then restart your computer. If this doesn't work, the Adafruit tutorial goes more in depth on getting the FTDI friend working for you: https://learn.adafruit.com/ftdi-friend/installing-ftdi-drivers

Perimeter Sample

```
static int x = 0;
static int y = 0;
static int frame = 0;
void mode_ourMode(uint8_t action) {

  // If we have some kind of action
  if(action != ACTION_NONE) {
      // Reset our static variable to zero
      x = 0;
      y = 0;
      frame = 0;
      watch.fillScreen(0);
      // If we got here either through someone holding left/right or
      // the watch woke up here
      if(action >= ACTION_HOLD_LEFT)
      {
        // Set the depth and plex
        uint8_t depth = 4, plex = LED_PLEX_2;
        // Reconfigure display if needed
        if((watch.getDepth() != depth) || (watch.getPlex() != plex))
          fps = watch.setDisplayMode(depth, plex, true);
      }
    // Reset sleep timeout on ANY button action
    watch.setTimeout(fps * 5);
    }
  watch.fillScreen(0);
  // We draw every frame, but only the value of x (not up to it)
  watch.drawPixel(x, y, 7);

  // We only update the value of x every eighth frame
  if (frame >= 2)
  {

   // We're on the top row
```

```
    if (y == 0)
    {
      // We're not all the way to the right yet, move to the right
      if (x < 7)
        x++;
      // We're at the edge, so move the pixel down
      else
        y++;
    }
    // We're on the very right column
    else if (x == 7)
    {
      // We're not all the way down yet, so move down
      if (y < 7)
        y++;
      // We're at the bottom, so move left
      else
        x--;
    }
    // We're on the bottom row
    else if (y == 7)
    {
      // We're not all the way left yet, so move to the left
      if (x > 0)
       x--;
      // We're at the very left, so move up
      else
       y--;
    }
    // We're on the very left column
    else if (x == 0)
    {
      // Not all the way to the top yet, move up
      if (y > 0)
        y--;
      // We're at the top, so move right again
      else
        x++;
    }


   // Since we updated this frame, reset the frame counter to zero
     frame = 0;
   }
   // If we didn't update, then increase the frame counter by one
   else
   {
     frame++;
   }

}
```