Tweet-a-watt

Created by Justin Cooper

https://learn.adafruit.com/tweet-a-watt

Last updated on 2023-08-29 02:16:25 PM EDT

Table of Contents

Overview 5

- Watch me make a Watt-watcher

Make it! 7

- Before you start...
- Make a tweet-a-watt

Prep 8

- Tutorials
- Tools

Receiver 12

- Overview
- What you'll need
- Solder the Adapter Together!
- Connect to the XBee

Configure 19

- Overview
- Upgrading the Firmware
- Rinse & Repeat
- Configure the Transmitter XBee
- Configure the receiver XBee
- Next!

Transmitter 27

- Before you start...
- Transmitter partslist
- Transmitter Schematic
- Step 3. Assemble and create the transmitter

Software 87

- Introduction
- Install python & friends
- Basic configure
- Test it out
- Calibrating
- Logging data
- Tweeting

Powermeter 96

- Tweet-a-Watt + Google Powermeter!
- Log in + Activate
- Python code
- Widgetty

Expand 101

- Improve on your system
- Add more outlets
- Graphing
- Remove the computer

• Get better range

Design 104

• Design overview

Listen 104

• Data listening & parsing
• Raw analog input
• Normalizing the data
• Basic data graphing
• Graphing wattage!
• Done!

Store 112

• You have 5 minutes!
• Multisensor!
• The App Engine
• Test!
• Getting the report out

Graph 122

• Making pretty pictures
• Configuring the sensor names
• Google Visualizer
• Data formatting
• Wrapping up the visualization
• Viz Viz Viz
• Timecodes!

Resources 130

• Other power monitoring projects!
• Power monitoring products
• Websites & Software

Download 131

• Software
• X-CTU profiles
• Design files

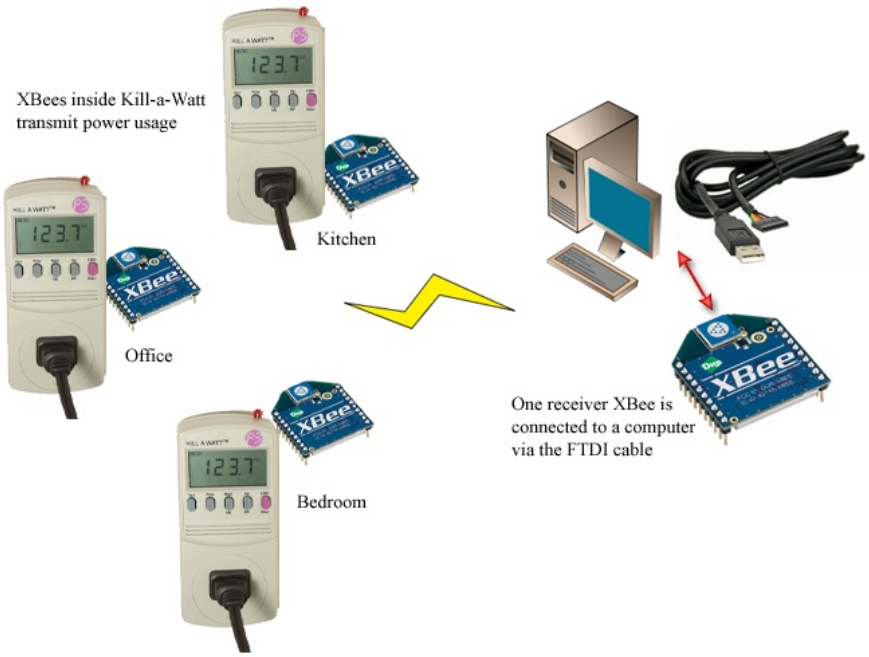
Overview

Watch me make a Watt-watcher

This project documents my adventures in learning how to wire up my home for wireless power monitoring. I live in a rented apartment so I don't have hacking-access to a meter or breaker panel. Since I'm still very interested in measuring my power usage on a long term basis, I built wireless outlet reporters. Building your own power monitor isn't too tough and can save money but I'm not a fan of sticking my fingers

into 120V power. Instead, I'll used the existing Kill-a-watt power monitor, which works great and is available at my local hardware store.

My plan is to have each room connected to a 6-outlet power strip which powers all the devices in that room (each kill-a-watt can measure up to 15A, or about 1800W, which is plenty!). That way I can track room-by-room usage, for example "kitchen", "bedroom", "workbench", and "office".



I spent about 10 minutes on this diagram .... can you tell?

Each wireless outlet/receiver can be built for ~$55 with a few easily-available electronic parts and light soldering, no microcontroller programming or high voltage engineering is necessary!

You can see my setup including graphs and reports at http://twitter.com/tweetawatt ()

If you'd like to build one for yourself

1. Buy a kit: () get all the parts you need, there's a starter kit at the adafruit webshop
2. Make (): turn each Kill-a-Watt into a wireless power level transmitter
3. Software: () Download & run it on your computer to get data and save it to a file and/or publish it

If you want to know how it was made, check out:

1. Listen (): write simple software for my computer (or Arduino, etc) to listen for signal and compute the current power usage
2. Store: () Create a database backend that will store the power usage for long-term analysis
3. View: () Graph and understand trends in power usage

Make it!

Before you start...

You should only attempt this project if you are comfortable and competant working with high voltage electricity, electronics and computers. Once the project is complete it is enclosed and there are no exposed high voltages. However, you must only work on the project when its not plugged in and never ever attempt to test, measure, open, or probe the circuitboards while they are attached to a wall socket. If something isnt working: stop, remove it from the wall power, then open it up and examine. Yes it takes a few more minutes but its a lot safer!

Your safety is your own responsibility, including proper use of equipment and safety gear, and determining whether you have adequate skill and experience. Power tools, electricity, and other resources used for this projects are dangerous, unless used properly and with adequate precautions, including safety gear. Some illustrative photos do not depict safety precautions or equipment, in order to show the project steps more clearly. This projects is not intended for use by children.

Use of the instructions and suggestions is at your own risk. Adafruit Industries LLC, disclaims all responsibility for any resulting damage, injury, or expense. It is your responsibility to make sure that your activities comply with applicable laws.

OK, if you agree we can move on!

Make a tweet-a-watt

To make the tweet-a-watt setup, we will have to go through a few steps

1. Prepare () by making sure we have everything we need and know the skills necessary to build the project
2. Build the receiver setup () by soldering up one of the adapter kits

3. Configure () the XBee wireless modems
4. Build the transmitter setup () by modifying a Kill-a-Watt to transmit via the XBee
5. Run the software (), which will retreive data and save it to a file, upload it to a database and/or twitter
6. Expand and extend () your setup

Prep

Tutorials

Learn how to solder with tons of tutorials! ()

Don't forget to learn how to use your multimeter too! ()

Tools

There are a few tools that are required for assembly. None of these tools are included. If you don't have them, now would be a good time to borrow or purchase them. They are very very handy whenever assembling/fixing/modifying electronic devices! I provide links to buy them, but of course, you should get them whereever is most convenient/inexpensive. Many of these parts are available in a place like Radio Shack or other (higher quality) DIY electronics stores.

There are great tools for all your kit making in the Adafruit Shop! ()

If you're just starting, and need a full set of tools, we suggest the "Ladyada's Toolkit" - for $100 it contains everything you need to get started (http://adafru.it/136)

But there's also lots of great individual tools available in the Adafruit store, so take a look around ()

Soldering iron

Any entry level 'all-in-one' soldering iron that you might find at your local hardware store should work. As with most things in life, you get what you pay for.

Upgrading to a higher end soldering iron setup, like the Hakko FX-888 that we stock in our store (http://adafru.it/180), will make soldering fun and easy.

Do not use a "ColdHeat" soldering iron! They are not suitable for delicate electronics work and can damage the kit (see here ()).

Click here to buy our entry level adjustable 30W 110V soldering iron (http://adafru.it/180).

Click here to upgrade to a Genuine Hakko FX-888 adjustable temperature soldering iron. (http://adafru.it/303)

Solder

You will want rosin core, 60/40 solder. Good solder is a good thing. Bad solder leads to bridging and cold solder joints which can be tough to find.

Click here to buy a spool of leaded solder (recommended for beginners) (http://adafru.it/145).

Click here to buy a spool of lead-free solder (http://adafru.it/734).

Multimeter

You will need a good quality basic multimeter that can measure voltage and continuity.

Click here to buy a basic multimeter. (http://adafru.it/71)

Click here to buy a top of the line multimeter. (http://adafru.it/308)

Click here to buy a pocket multimeter. (http://adafru.it/850)

Flush Diagonal Cutters

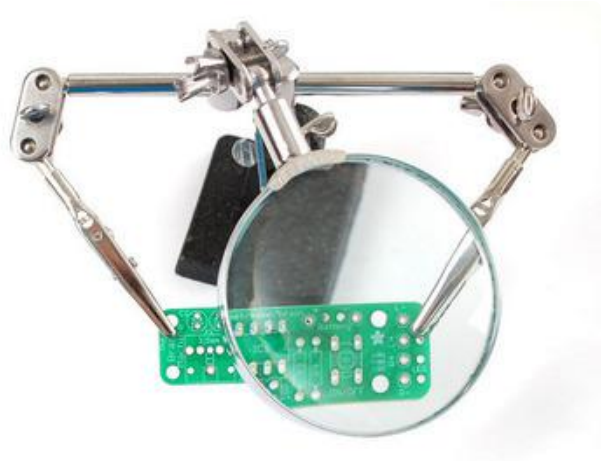
You will need flush diagonal cutters to trim the wires and leads off of components once you have soldered them in place.

Click here to buy our favorite cutters (http://adafru.it/152).

Solder Sucker

Strangely enough, that's the technical term for this desoldering vacuum tool. Useful in cleaning up mistakes, every electrical engineer has one of these on their desk.

Click here to buy a one (http://adafru.it/148).



Helping Third Hand With Magnifier

Not absolutely necessary but will make things go much much faster, and it will make soldering much easier.

Pick one up here (http://adafru.it/291).

Good light. More important than you think.

Receiver

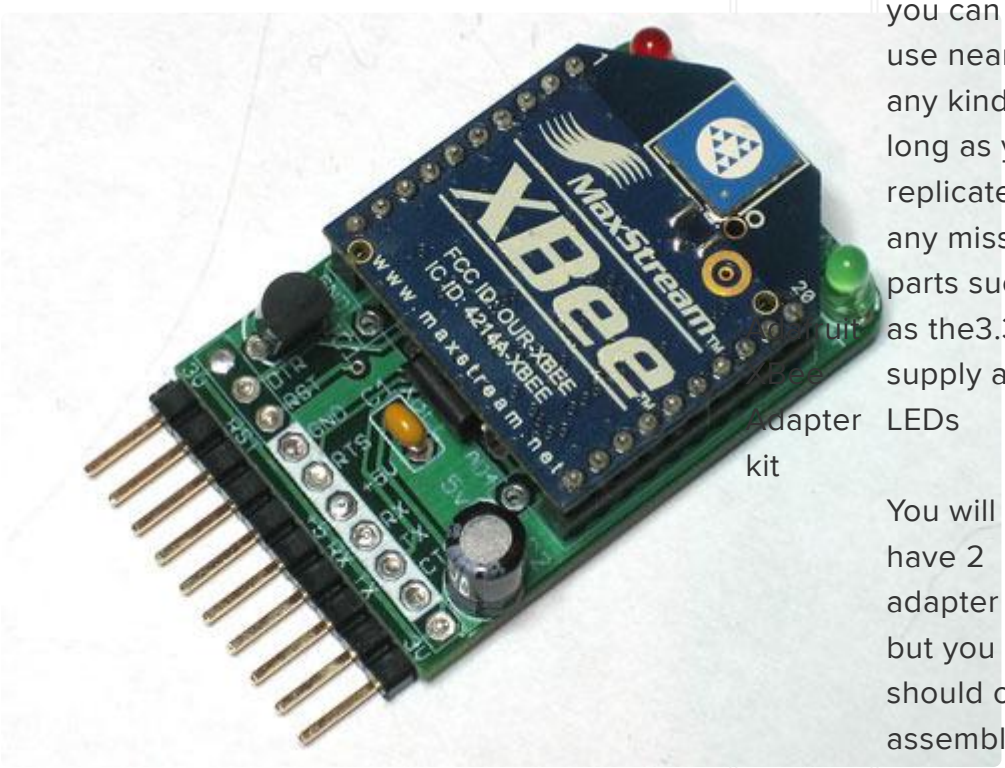
Overview

We'll start with the receiver hardware, thats the thing that plugs into the computer and receives data from the wireless power plug. The receiver hardware does 'double duty', it also is used to update the XBees modems' firmware (which, unfortunately, is necessary because they come from the factory with really old firmware) and configure the modems.

What you'll need

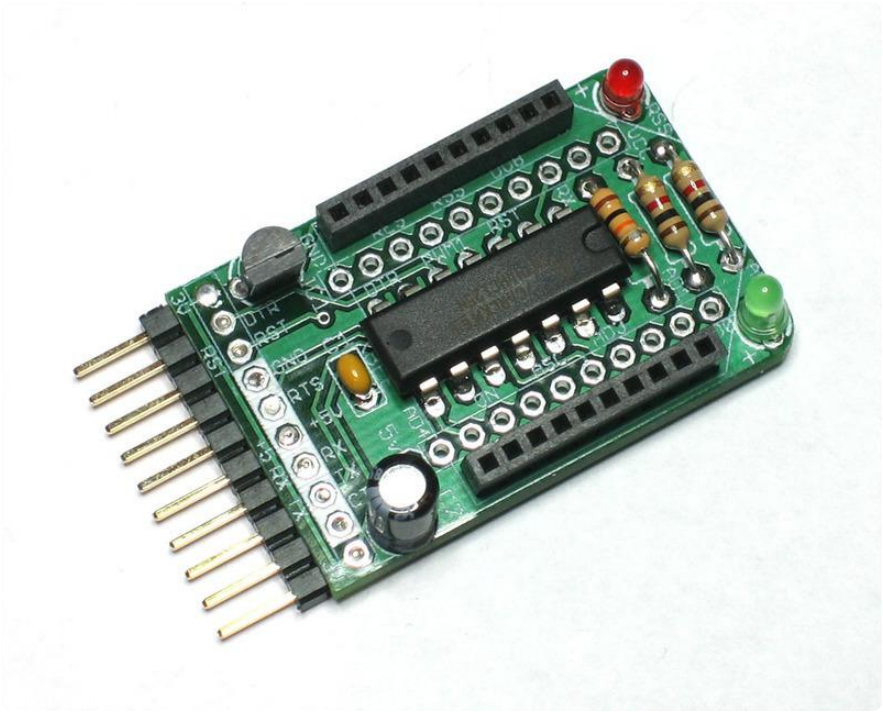
The receiver is essentially, an XBee, with a USB connection to allow a computer to talk to it the XBee.

Image	Name	Description	Datasheet	Distributor
TTL-232R_LRG.jpg	FTDI cable	A USB-to-serial converter. Plugs in neatly into the Adafruit XBee adapter to allow a computer to talk to the XBee.	TTL-232R 3.3V or 5.0V	Adafruit

	Adafruit XBee Adapter kit	I'll be using my own design for the XBee breakout/ carrier board but you can use nearly any kind as long as you replicate any missing parts such as the3.3V supply and LEDs You will have 2 adapter kits but you should only assemble one for this part! The other one needs different instructions so just hold off!	Webpage	Adafruit

	XBee module	We'll be using the XBee "series 1" point-to-multipoint 802.15.4 modules with a chip antenna part # XB24-ACI-001. They're inexpensive and work great. This project most likely won't work with any other version of the XBee, and certainly not any of the 'high power' Pro types!		Adafruit

Solder the Adapter Together!



This step is pretty easy, just go over to the XBee adapter webpage () and solder it together according to the instructions! ()

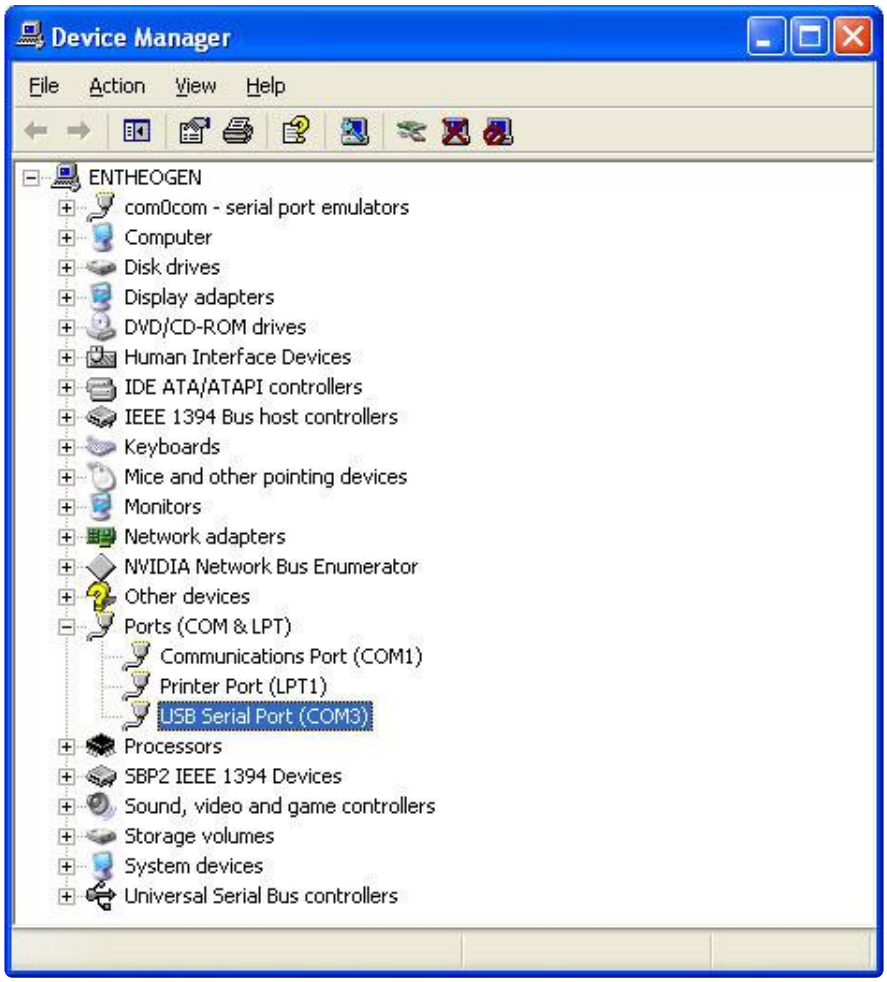
Remember: You will have 2 adapter kits but you should only solder one of them at this point! The other one needs different instructions so just hold off!

Connect to the XBee

Now its time to connect to the XBees.

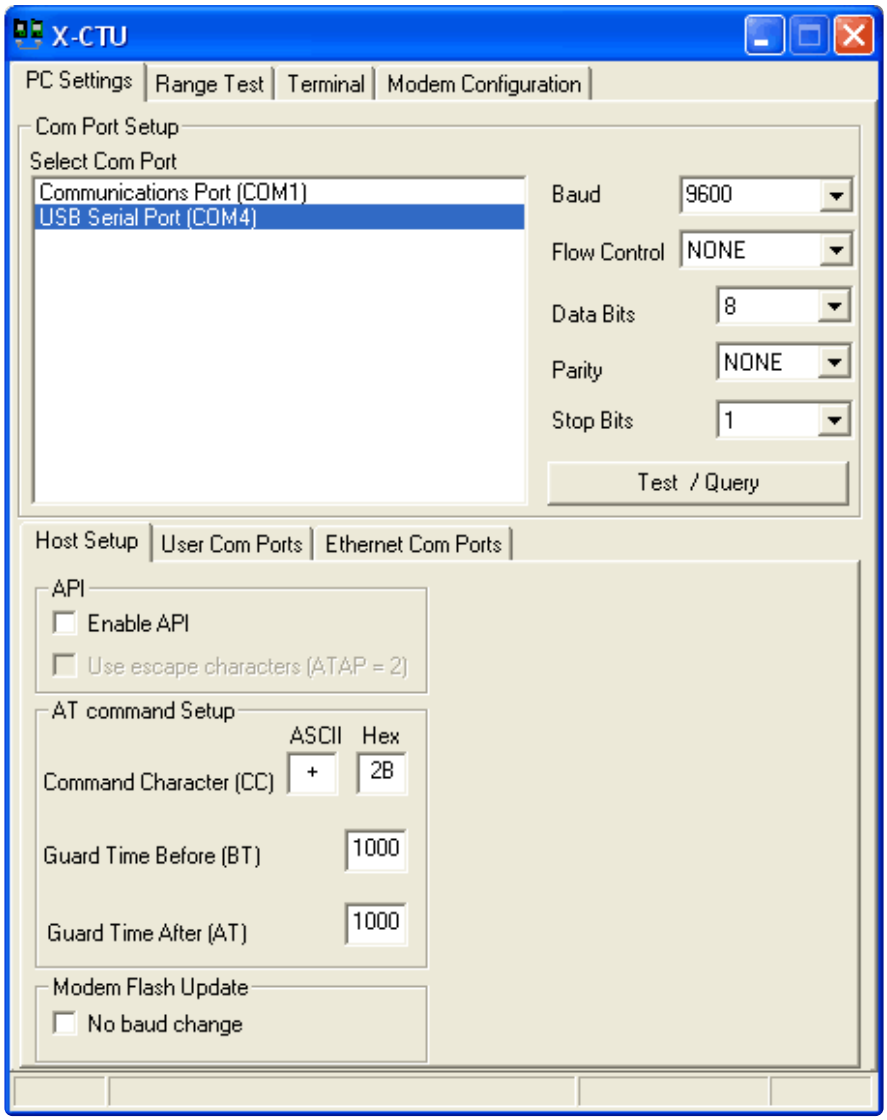
Find your FTDI cable ()- use either 3.3V or 5V. These cables have a USB to serial converter chip molded into them and are supported by every OS. Thus configuring or upgrading or connecting is really trivial. Simply plug the cable into the end of the module so that the black wire lines up with GND. There is a white outline showing where the cable connects.

You'll need to figure out which serial port (COM) you are using. Plug in the FTDI cable, USB adapter, Arduino, etc. Under Windows, check the device manager, look for "USB Serial Port."

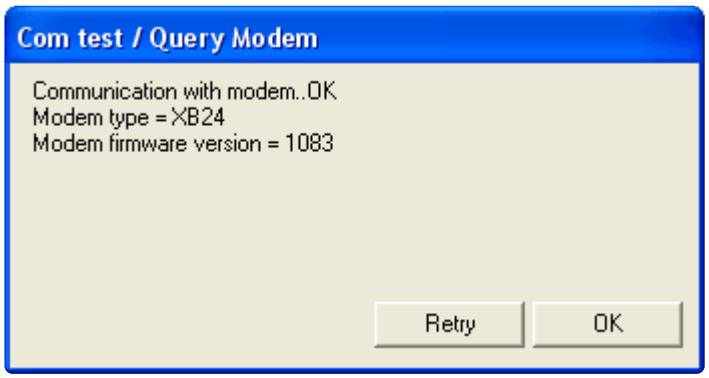


Digi/Maxstream wrote a little program to help configure XBees, its also the only way I know of to upgrade them to the latest firmware. Unfortunately it only runs on Windows. Download X-CTU from Digi and install it on your computer ().

After installing and starting the program, select the COM port (COM4 here) and baud rate (9600 is default). No flow control, 8N1. Make sure the connection box looks just like this (other than the com port which may be different).



To verify, click Test / Query



Hopefully the test will succeed. If you are having problems: check that the XBee is powered, the green LED on the adapter board should be blinking, the right COM port & baud rate is selected, etc.

Now unplug the adapter from the FTDI cable, carefully replace the first XBee with the other one and make sure that one is talking fine too. Once you know both XBees are working with the adapter, its time to upgrade and configure them, the next step!

If you are having problems connecting and you have a used xbee or you have one set to sleep mode by accident, it may not respond because its asleep. See this forum topic for how to reset it ().

Configure

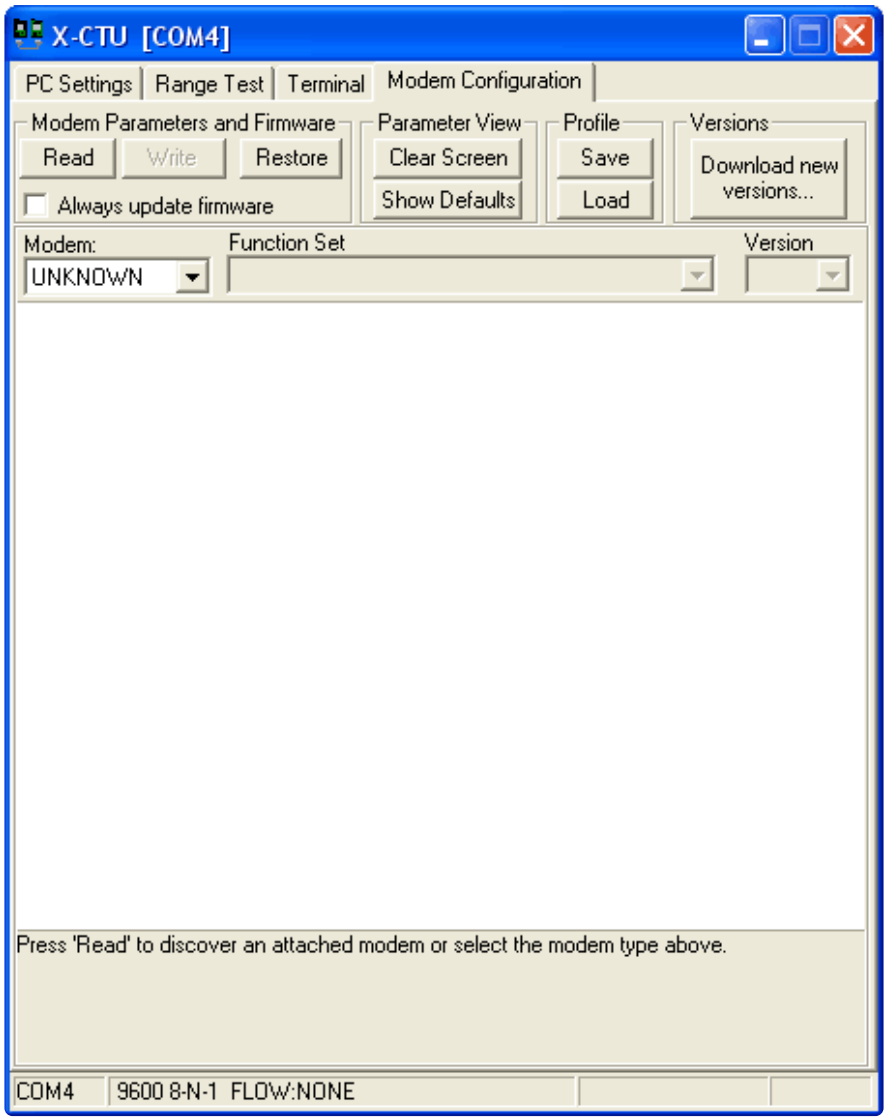
Overview

OK so far you have assembled one of the XBee adapter boards and connected it to your computer using the FTDI cable. (The other adapter is for later so don't do anything with it yet!) The XBees respond to the X-CTU software and are blinking just fine. Next we will update the firmware.

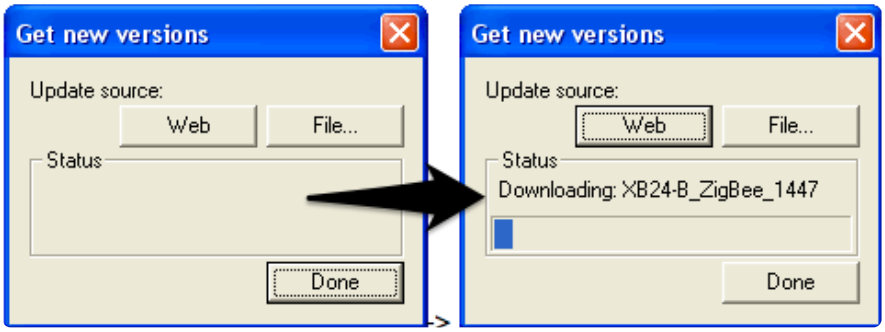
Upgrading the Firmware

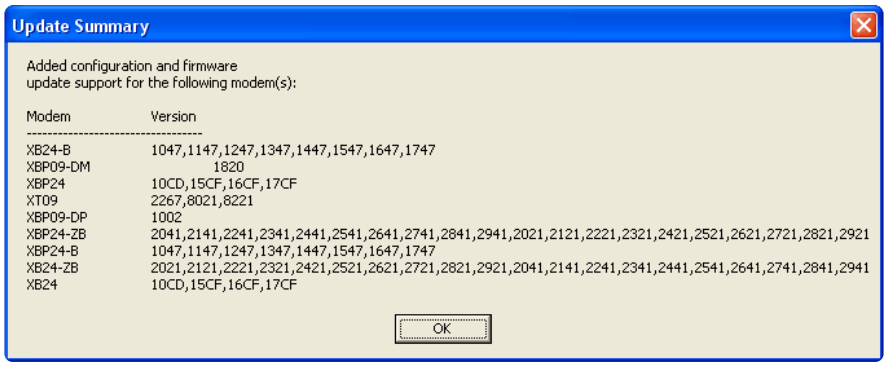
There's a good chance your XBees are not running the latest firmware & there's a lot of features added, some of which we need to get this project running. So next up is upgrading!

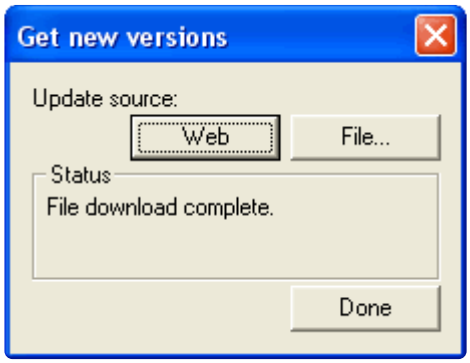
Go to the Modem Configuration tab. This is where the modem is configured and updated.



Click Download new versions... and select to download the latest firmwares from the Web.

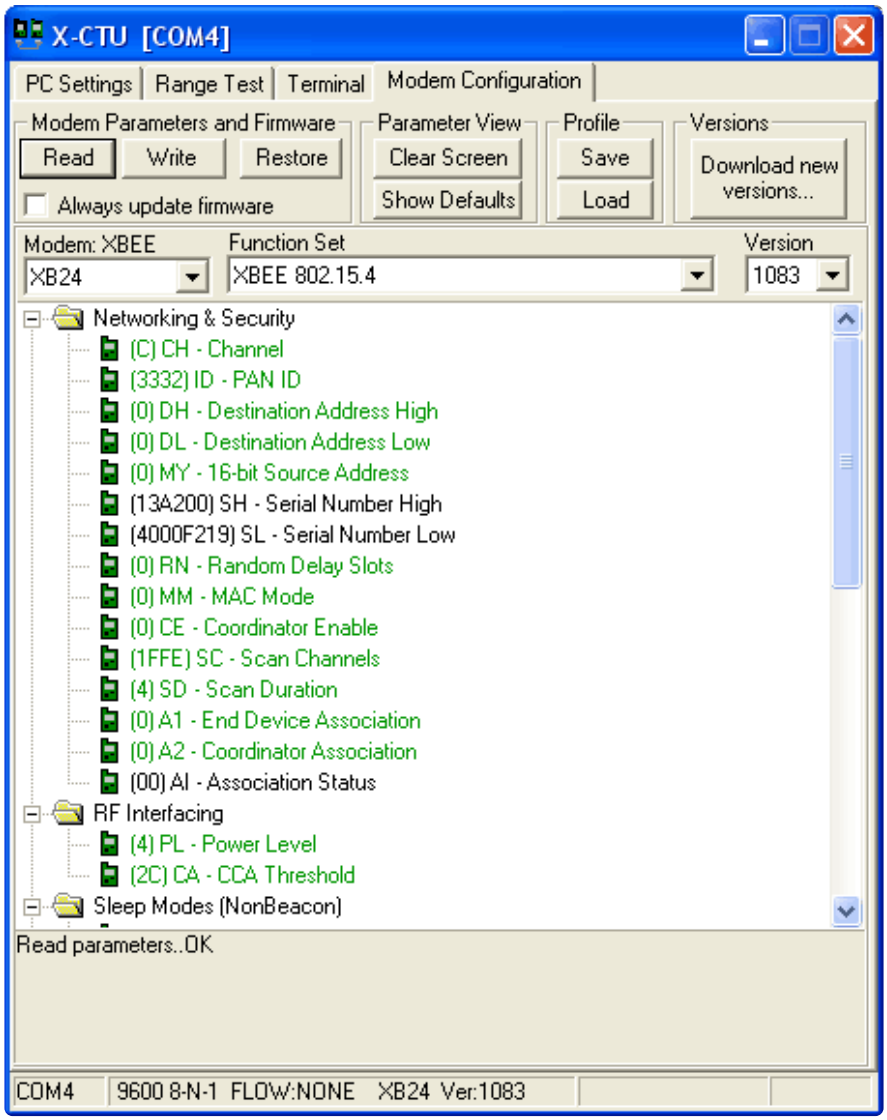






Once you have downloaded the newest firmware, its time to upgrade!

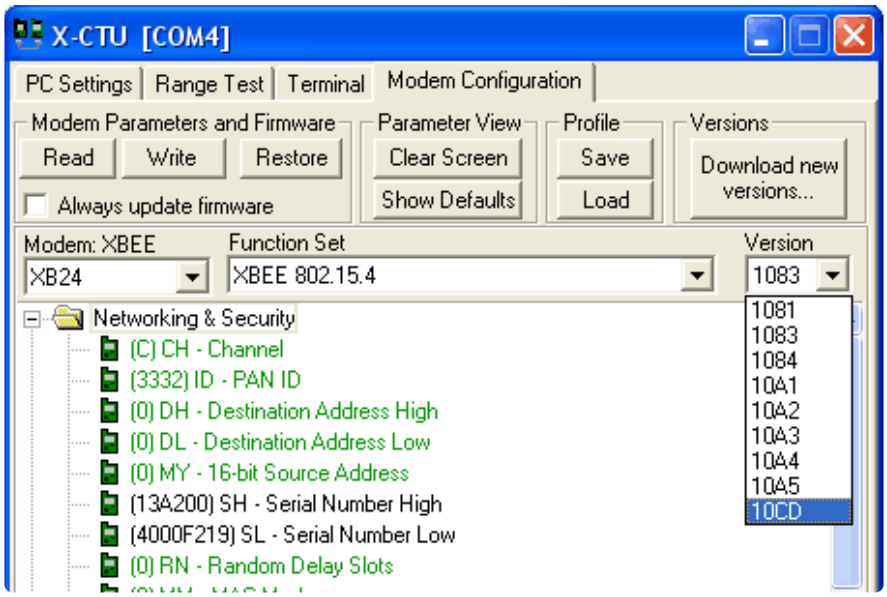
Click on Modem Parameters -> "Read" to read in the current version and settings



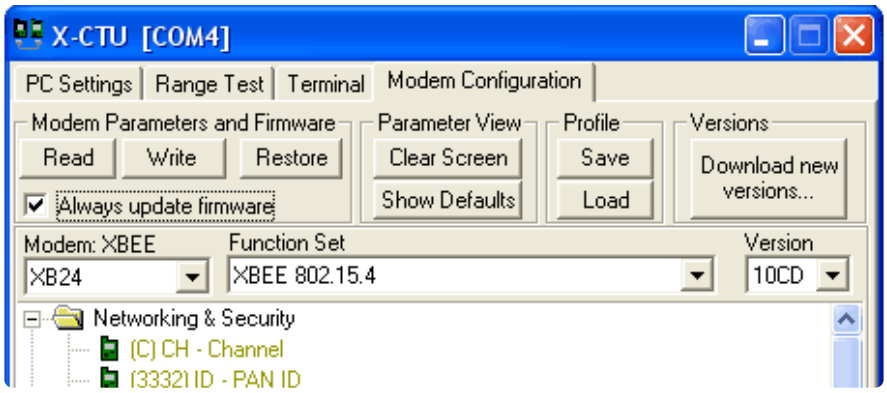
Now you will know for sure what function set, version and settings are stored in the modem.

If you are having problems connecting and you have a used xbee or you have one set to sleep mode by accident, it may not respond because its asleep. See this forum topic for how to reset it ().

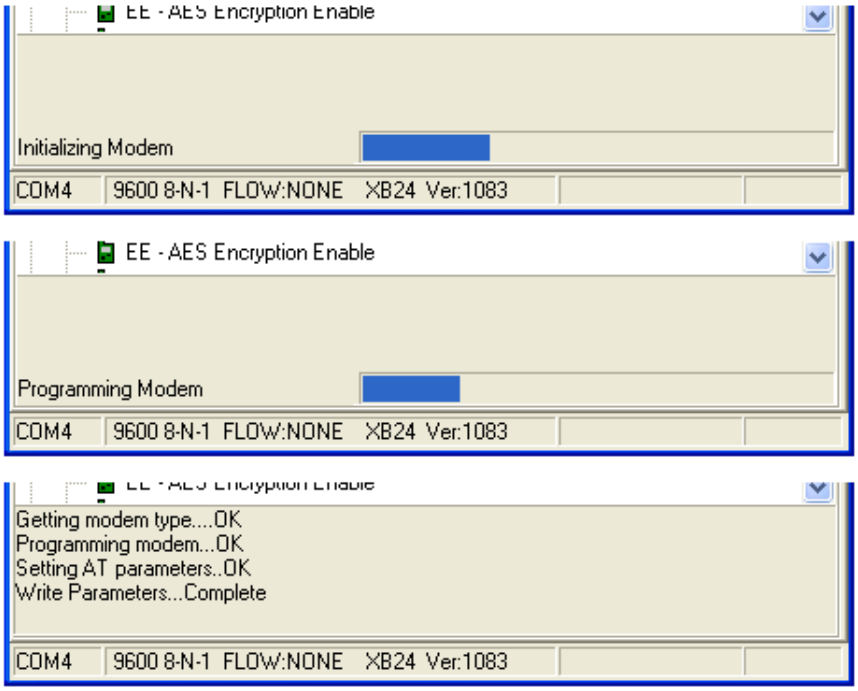
Select from the Version dropdown the latest version available.



Check the Always update firmware checkbox.



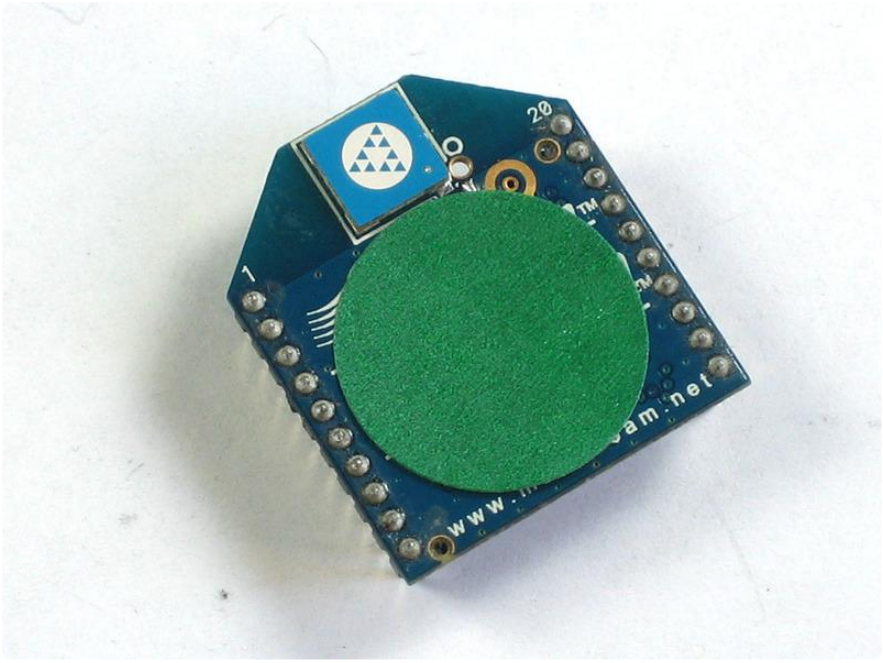
And click Write to initialize and program the new firmware in!



That's it, now you have the most recent firmware for your modem. You should now uncheck the Always update firmware checkbox. If you have problems, like for example timing out or not being able to communicate, make sure the RTS pin is wired up correctly as this pin is necessary for upgrading. FTDI cables are already set up for this so you shouldn't have a problem.

Rinse & Repeat

Upgrade the firmware on both of the XBees so they are both up to date.



At this point it might be wise to label the two XBees in a way that lets you tell them apart. You can use a sharpie, a sticker or similar to indicate which one is the receiver and which is the transmitter.

Configure the Transmitter XBee

Both XBee's need to be upgraded with the latest firmware but only the transmitter (which is going to be put inside a Kill-a-Watt) needs to be configured. The configure

process tells the XBee what pins we want to read the sensor data off of. It also tells the XBee how often to send us data, and how much.

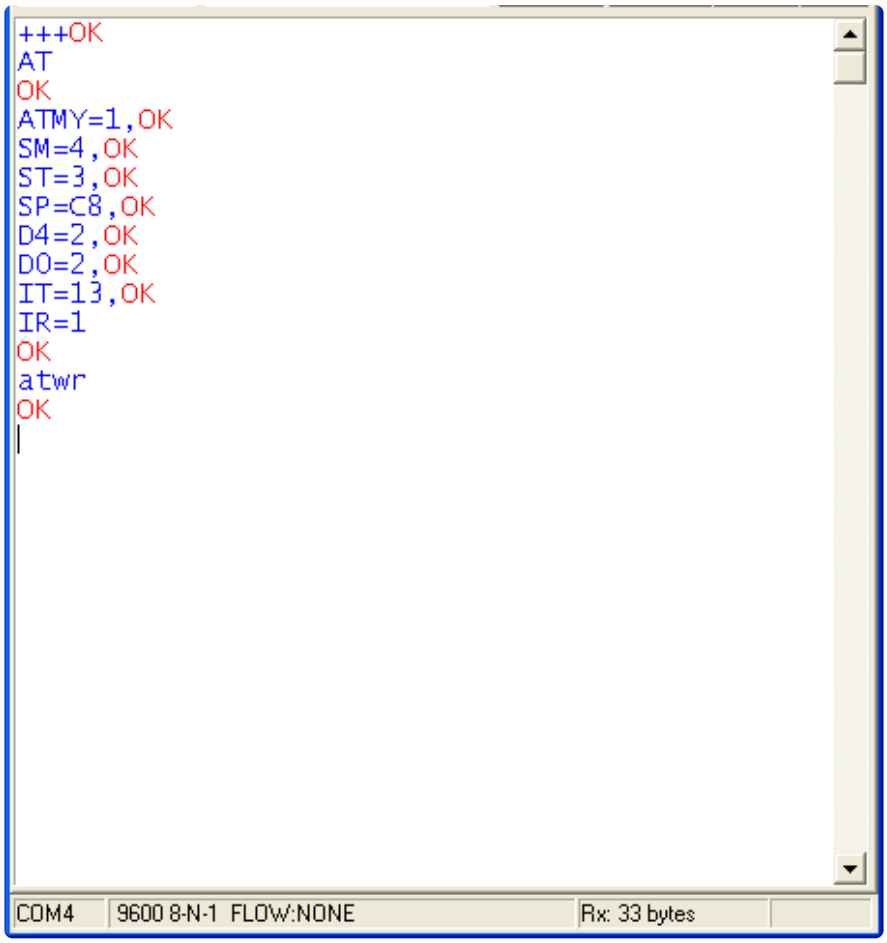
Plug the transmitter XBee into the USB connection (put the receiver XBee away) and start up X-CTU () or a Terminal program. Connect at 9600 baud, 8N1 parity.Then configure each one as follows:

1. Set the MY address (the identifier for the XBee) to 1 (increment this for each transmitter so you can tell them apart, we'll assume you only have one for now)
2. Set the Sleep Mode SM to 4 (Cyclic sleep)
3. Set the Sleep Time ST to 3 (3 milliseconds after wakeup to go back to sleep)
4. Set the Sleep Period SP to C8 (0xC8 hexadecimal = 200 x 10 milliseconds = 2 seconds between transmits)
5. Set ADC 4 D4 to 2 (analog/digital sensor enable pin AD4)
6. Set ADC 0 D0 to 2 (analog/digital sensor enable pin AD0)
7. Set Samples to TX IT to 13 (0x13 = 19 A/D samples per packet)
8. Set Sample Rate IR to 1 (1 ms between A/D samples)

if you think there will be more XBee's in the area that could conflict with your setup you may also want to

1. Set the PAN ID to a 4-digit hex number (its 3332 by default)

You can do this with X-CTU or with a terminal program such as hyperterm, minicom, zterm, etc. with the command string
ATMY=1,SM=4,ST=3,SP=C8,D4=2,D0=2,IT=13,IR=1 <return>
You'll need to start by getting the modem's attention by waiting 10 seconds, then typing in +++ quickly, then pausing for another 5 seconds. Then use AT <return> to make sure its paying ATtention to your commands.



Basically what this means is that we'll have all the XBees on a single PAN network, each XBee will have a unique identifier, they'll stay in sleep mode most of the time, then wake up every 2 seconds to take 19 samples from ADC 0 and 4, 1ms apart. If you're having difficulty, make sure you upgraded the firmware!

Make sure to WRITE the configuration to the XBee's permanent storage once you've done it. If you're using X-CTU click the "Write" button in the top left. If you're using a terminal, use the command ATWR !

Note that once the XBee is told to go into sleep mode, you'll have to reset it to talk to it because otherwise it will not respond and X-CTU will complain. You can simply unplug the adapter from the FTDI cable to reset or touch a wire between the RST and GND pins on the bottom edge of the adapter.

Now that the transmitters are all setup with unique MY number ID's, make sure that while they are powered from USB the green LED blinks once every 2 seconds (indicating wakeup and data transmit).

If you are having problems connecting and you have a used xbee or you have one set to sleep mode by accident, it may not respond because its asleep. See this forum topic for how to reset it ().

Configure the receiver XBee

Plug the receiver XBee into the USB connection (put the receiver XBee away) and start up X-CTU (). If you set the PAN ID in the previous step, you will have to do the same here

- Set the PAN ID to the same hex number as above

If you didn't change the PAN above, then there's nothing for you to do here, just skip this step

Next!

Now that the XBees are configured and ready, its time to go to the next step where we make the Kill-a-Watt hardware ()

Transmitter

Before you start...

You should only attempt this project if you are comfortable and competant working with high voltage electricity, electronics and computers. Once the project is complete it is enclosed and there are no exposed high voltages. However, you must only work on the project when it's not plugged in and never ever attempt to test, measure, open, or probe the circuitboards while they are attached to a wall socket. If something isn't working: stop, remove it from the wall power, then open it up and examine. Yes it takes a few more minutes but its a lot safer!

Your safety is your own responsibility, including proper use of equipment and safety gear, and determining whether you have adequate skill and experience. Power tools, electricity, and other resources used for this project are dangerous, unless used properly and with adequate precautions, including safety gear. Some illustrative photos do not depict safety precautions or equipment, in order to show the project steps more clearly. This projects is not intended for use by children.

Use of the instructions and suggestions is at your own risk. Adafruit Industries LLC, disclaims all responsibility for any resulting damage, injury, or expense. It is your responsibility to make sure that your activities comply with applicable laws.

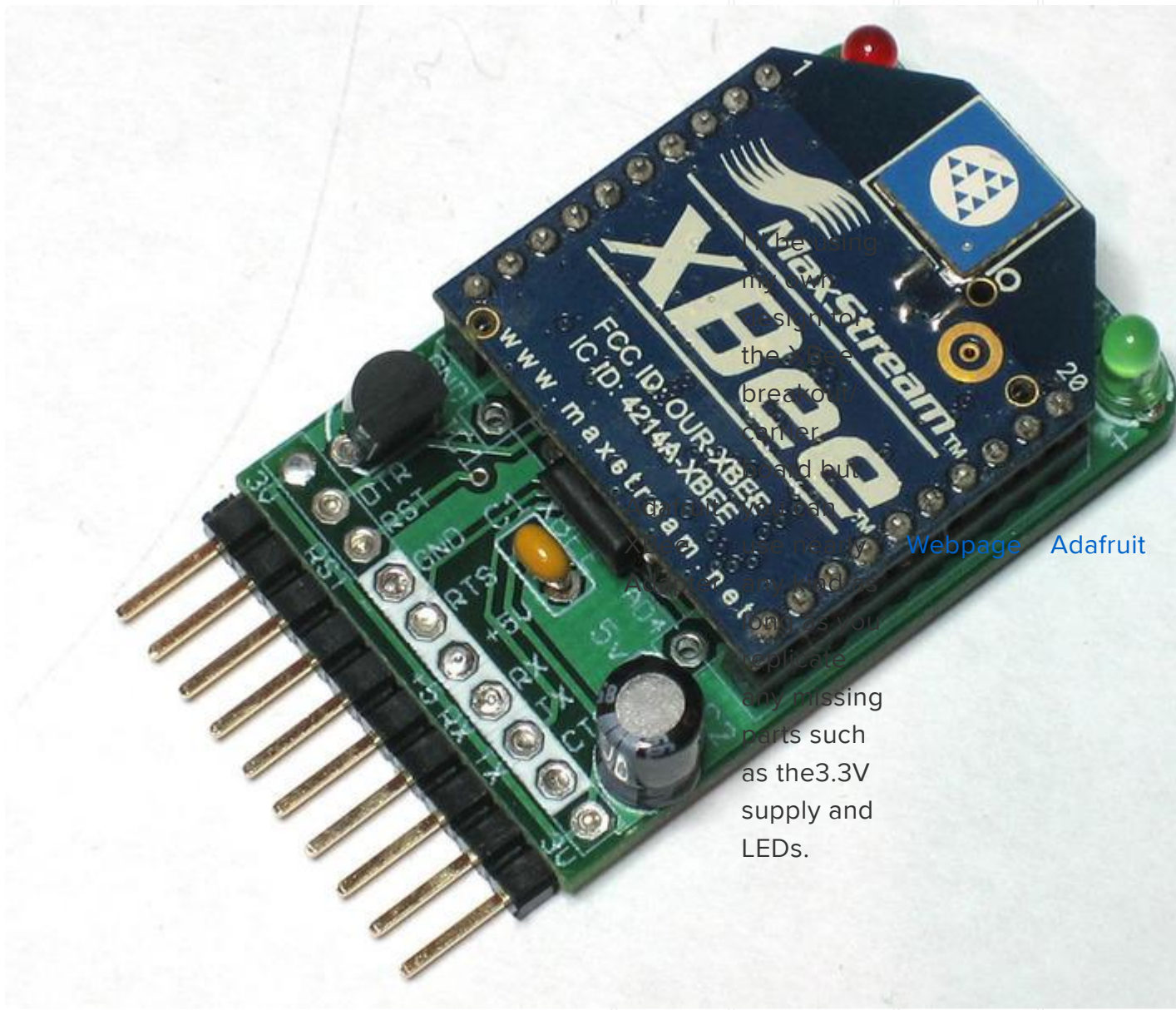
OK, if you agree we can move on!

Transmitter partslist

For each outlet you want to monitor, you'll need:

Image	Name	Description	Datasheet	Distributor

Kill-a-Watt	"Off the shelf" model P4400 power monitor Some people in the forum have gotten this to work with an EZ type but we don't suggest it. Try to go with a 'classic' Kill-a-Watt.	P3 Kill-a-watt	Lots! Also check hardware/ electronics stores.



done.jpg

I'll be using my own design for the XBee breakout carrier board but you can use nearly any breakout board you find online. Webpage Adafruit

Adafruit

Just duplicate any missing parts such as the3.3V supply and LEDs.

	XBee module	We'll be using the XBee "series 1" point-to-multipoint 802.15.4 modules with a chip antenna part # XB24-ACI-001. They're inexpensive and work great. This project most likely won't work with any other version of the XBee, and certainly not any of the 'high power' Pro types!		Adafruit

	D3	1N4001 diode. Any power diode should work fine. Heck, even a 1n4148 or 1n914 should be OK. But 1N4001 is suggested and is in the kit.	Generic 1N4001	Digikey Mouser
	D2	Large diffused LED, for easy viewing. The kit comes with green.		
	C2	220uF, 4V or higher (photo shows 100uF)	Generic	Digikey Mouser

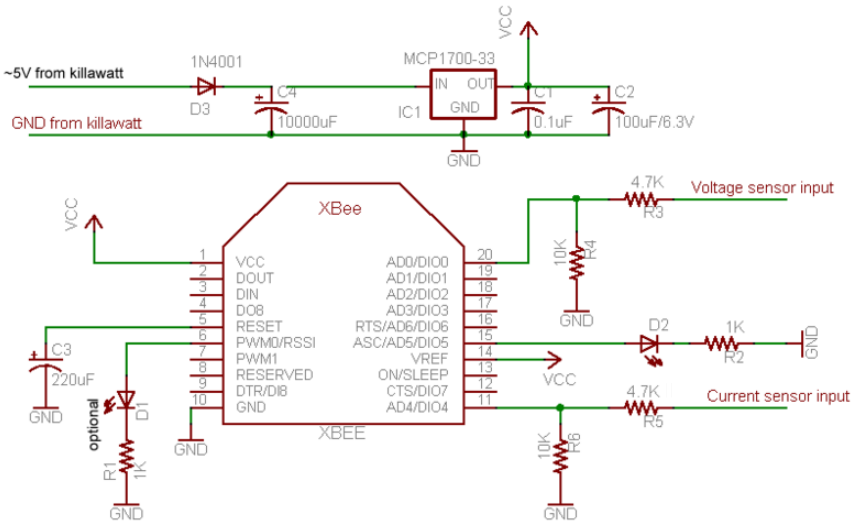
2200uf_t.jpg	C4	10,000uF capacitor (wow!) / 6.3V (photo shows a mere 2200uF) Try to get 16mm diameter, 25mm long.	Generic	Digikey Mouser
	R4 R6	10K 1/4W 1% resistor (brown black black red gold) or10K 1/4W 5% resistor (brown black orange gold) 1% is preferred but 5% is OK	Generic	Mouser Digikey

	R3 R5	4.7K 1/4W 1% resistor (yellow violet black brown gold), or 4.7K 1/4W 5% resistor (yellow violet red gold) 1% is preferred but 5% is OK.	Generic	Mouser Digikey
 10condwire.jpg		Ribbon cable, or other flexible wire, at least 6 conductors, about 6" long.	Generic Ribbon	Digikey
 heatshrink_t.jpg		Heat shrink! A couple inches of 1/8" and 3/32" each.	Generic	

It will run you about $50-$60 for each outlet.

Transmitter Schematic

We'll be making this:

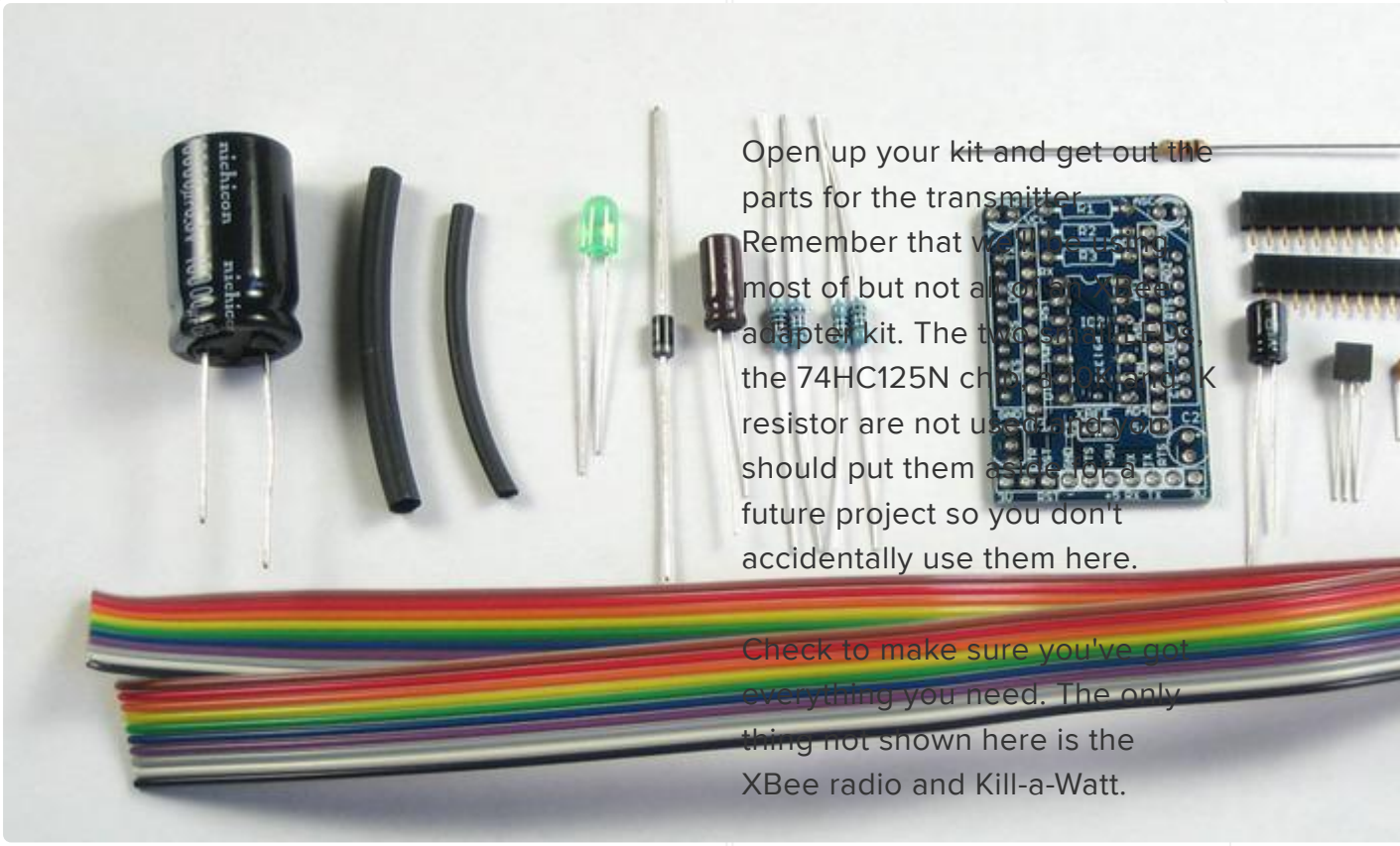


The XBee radio does all of the hard work, it listens on two analog input ports (AD0 and AD4) for voltage and current data. Then it transmits that information wirelessly to the host computer receiver XBee. There are a few we have to engineer around to make it Work:

1. We want to run the XBee off the Kill-a-Watt's internal power supply. However it's current limited and won't provide 50mA in a burst when the XBee transmits. We solve this by adding a simple 'rechargable battery' in the form of a really large capacitor C4.
2. The Kill-a-Watt runs at 5V but XBees can only run at 3.3V so we have a voltage regulator IC1 and two capacitors two stabilize the 3.3V supply, C1 and C2.
3. The XBee will transmit every few seconds, even while the capacitor is charging. This means that it will keep draining the capacitor, resetting, and trying again, basically freaking out while the power supply is still building. We prevent this by adding another fairly big capacitor C3 on the reset line. This slows down the XBee, delaying the startup by a few seconds & keeps the XBee from starting up till we have solid power.
4. The XBee analog sensors run at 3.3V but the Kill-a-Watt sensors run at 5V. We use simple voltage dividers R3/R4 and R5/R6 to reduce the analog signal down to a reasonable level.

Step 3. Assemble and create the transmitter

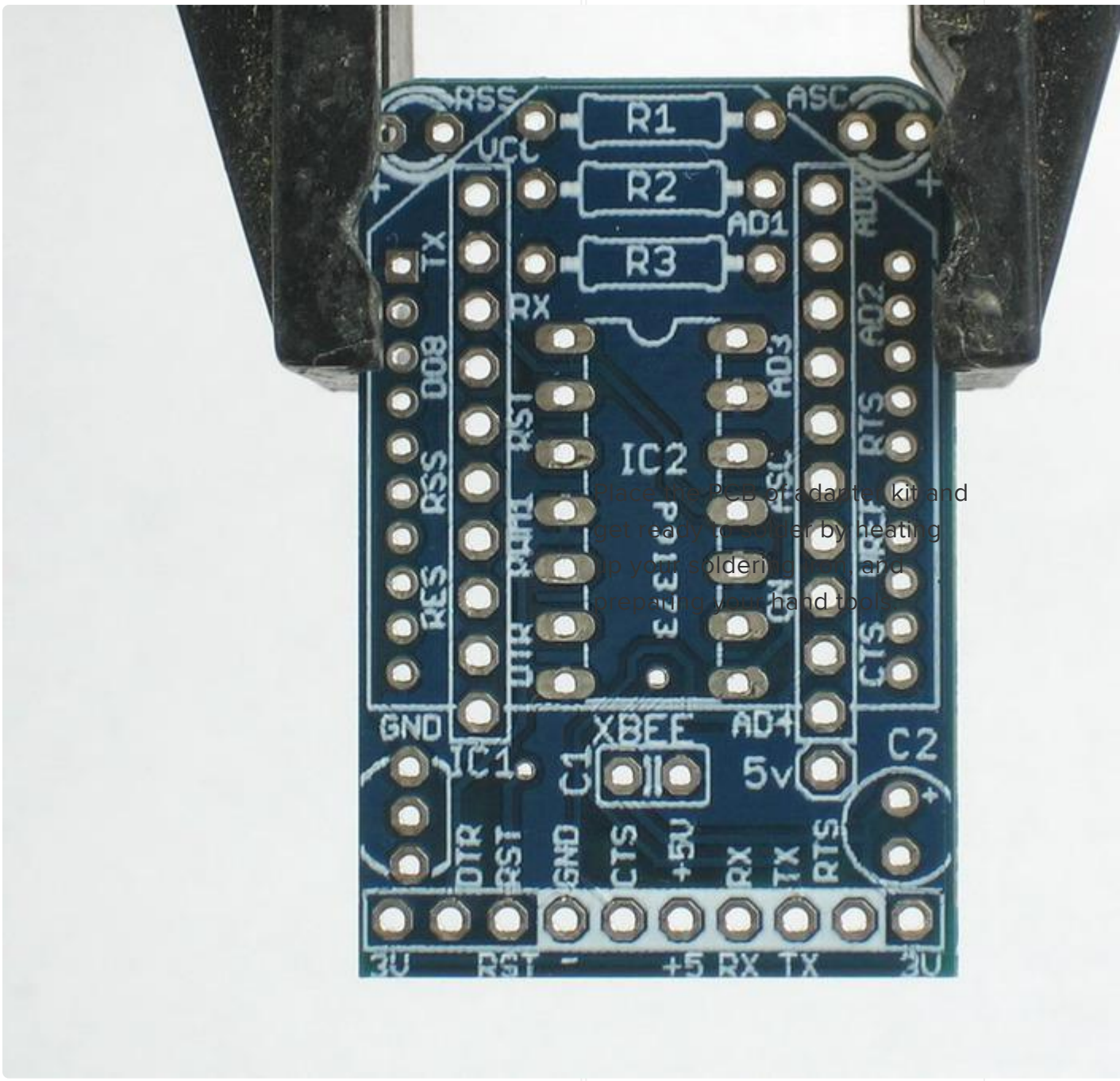
Now the fun part! We'll filet, stuff and reassemble the Kill-a-Watt with a radio inside!



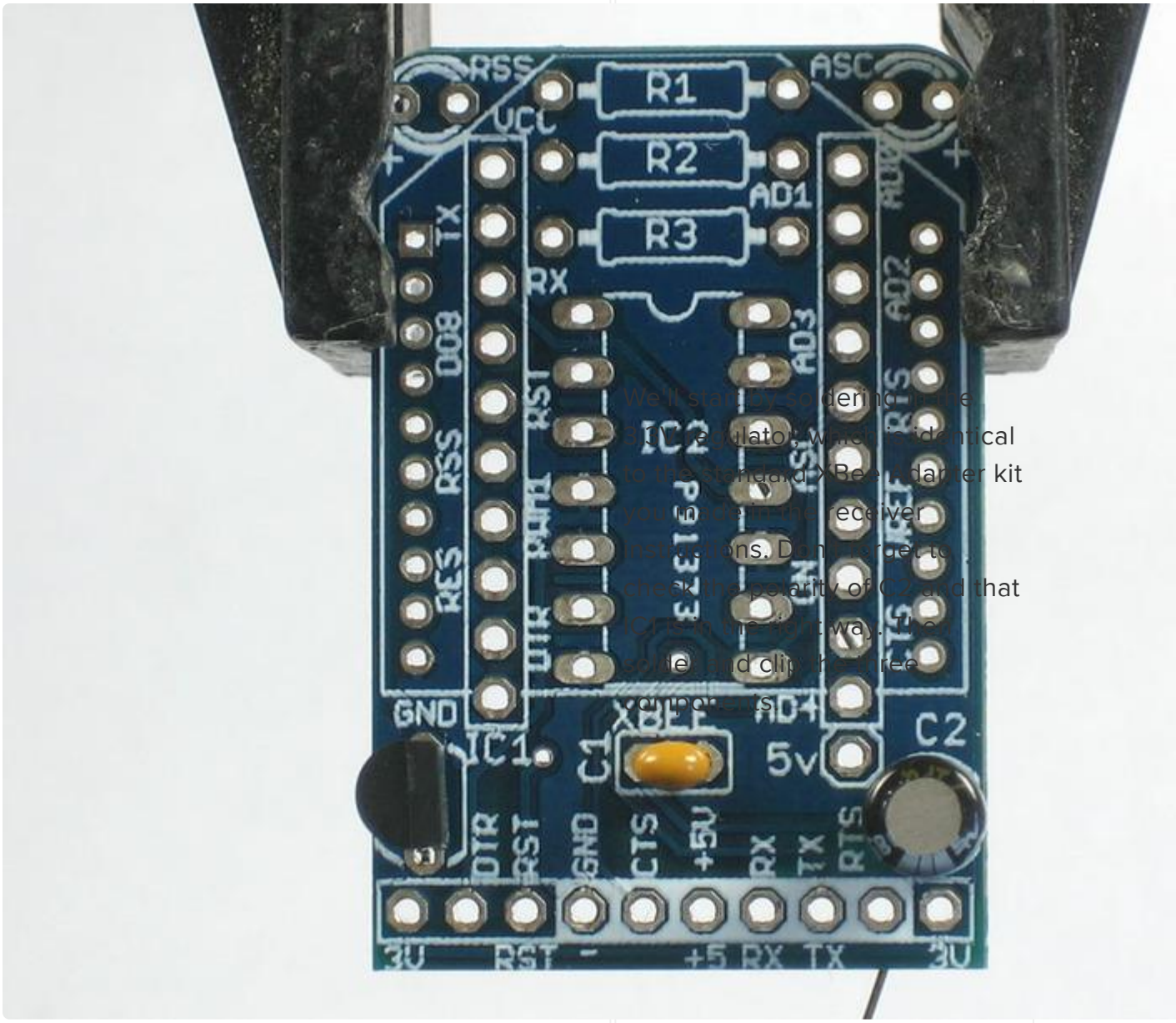
parts.jpg

Open up your kit and get out the parts for the transmitter. Remember that v[illegible] most of but not a[illegible] adapter kit. The t[illegible] the 74HC125N ch[illegible] resistor are not us[illegible] should put them a[illegible] future project so you don't accidentally use them here.

Check to make sure you've got everything you need. The only thing not shown here is the XBee radio and Kill-a-Watt.

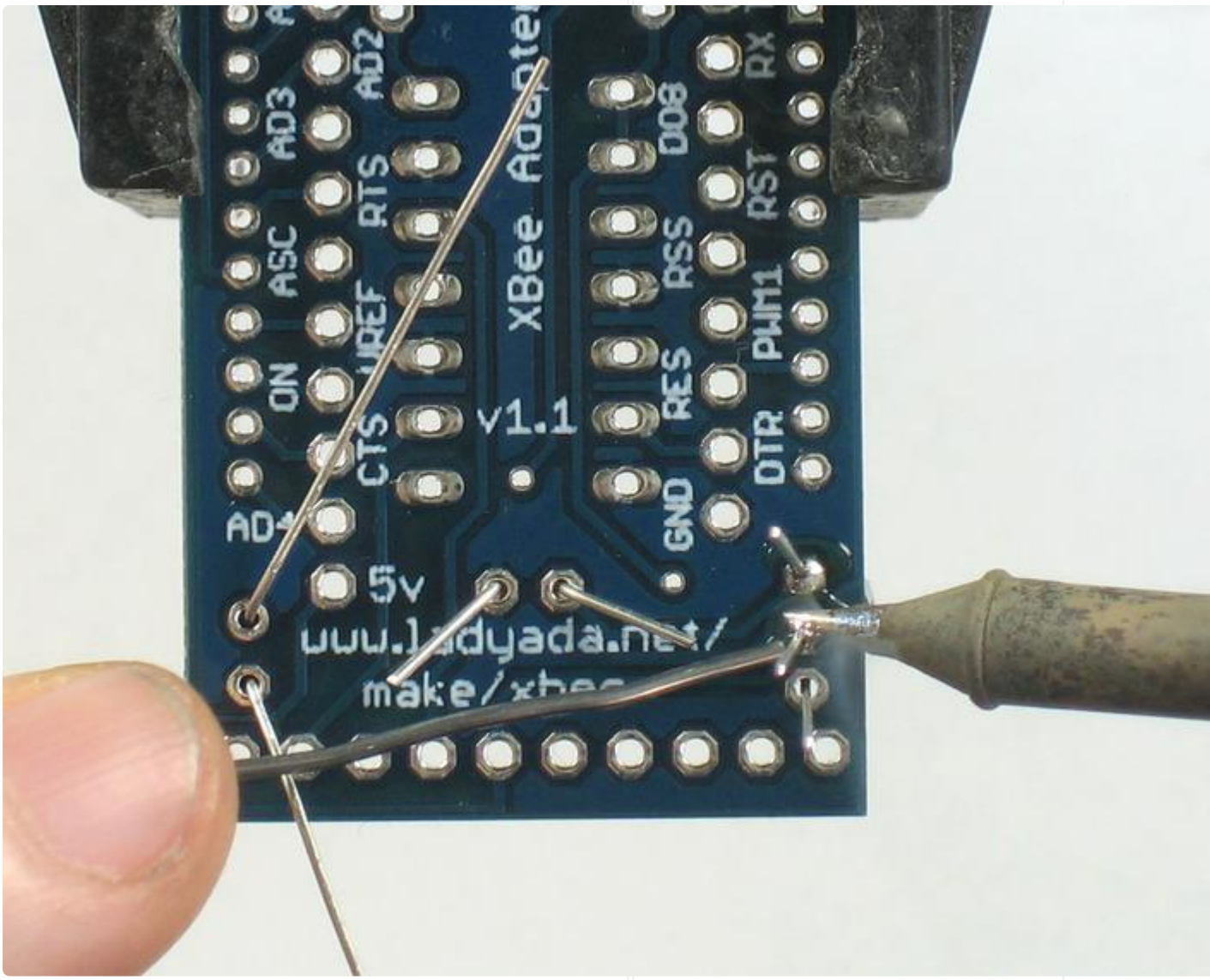


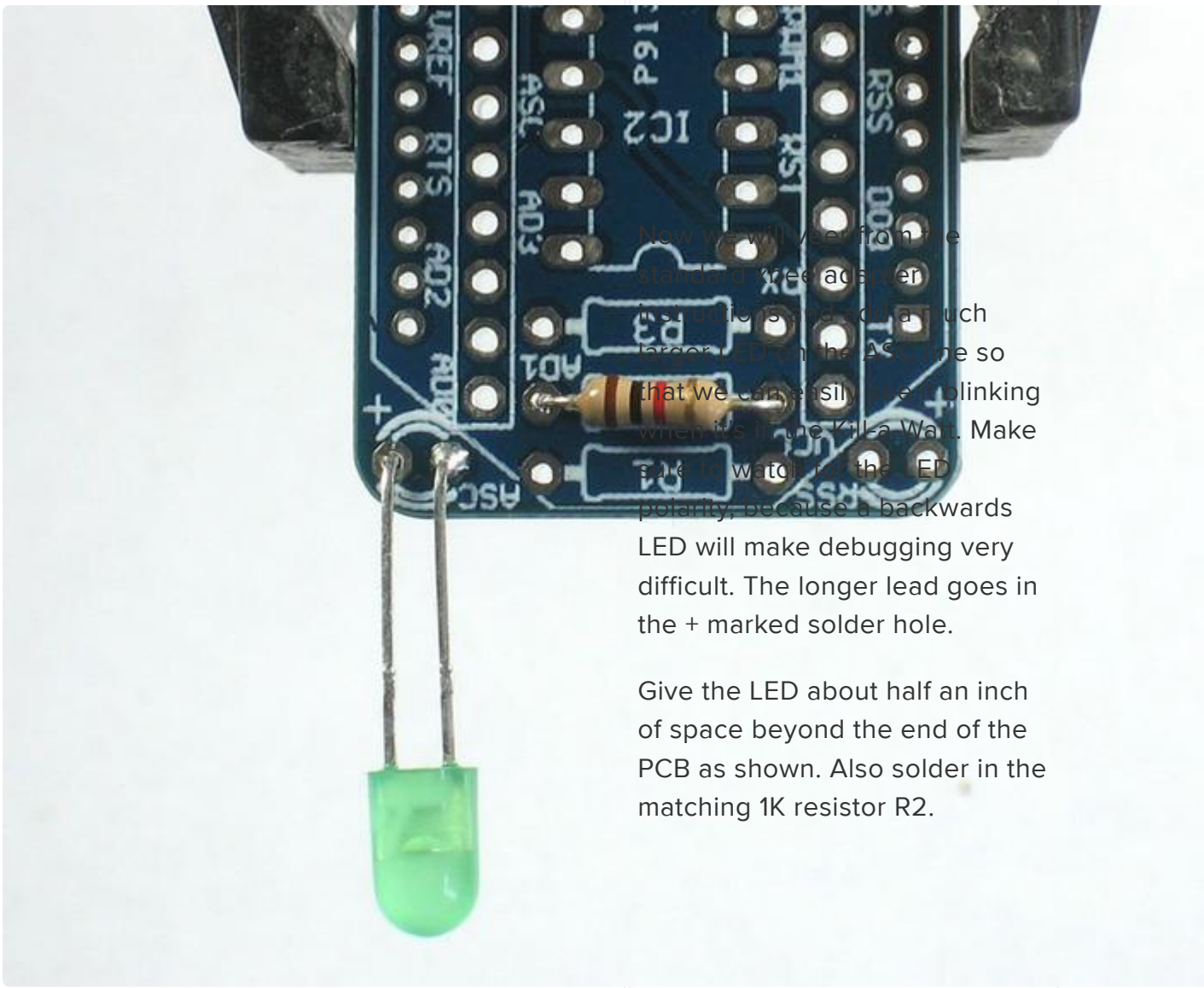
Place the PCB breakout kit and get ready to solder by heating up your soldering iron and preparing your hand tools.



psup.jpg

We'll start by soldering in the 3.3V regulator which is identical to the standard XBee adapter kit you made in the receiver Instructions. Don't forget to check the polarity of C2 and that IC1 is in the right way. Then solder and clip the three components.

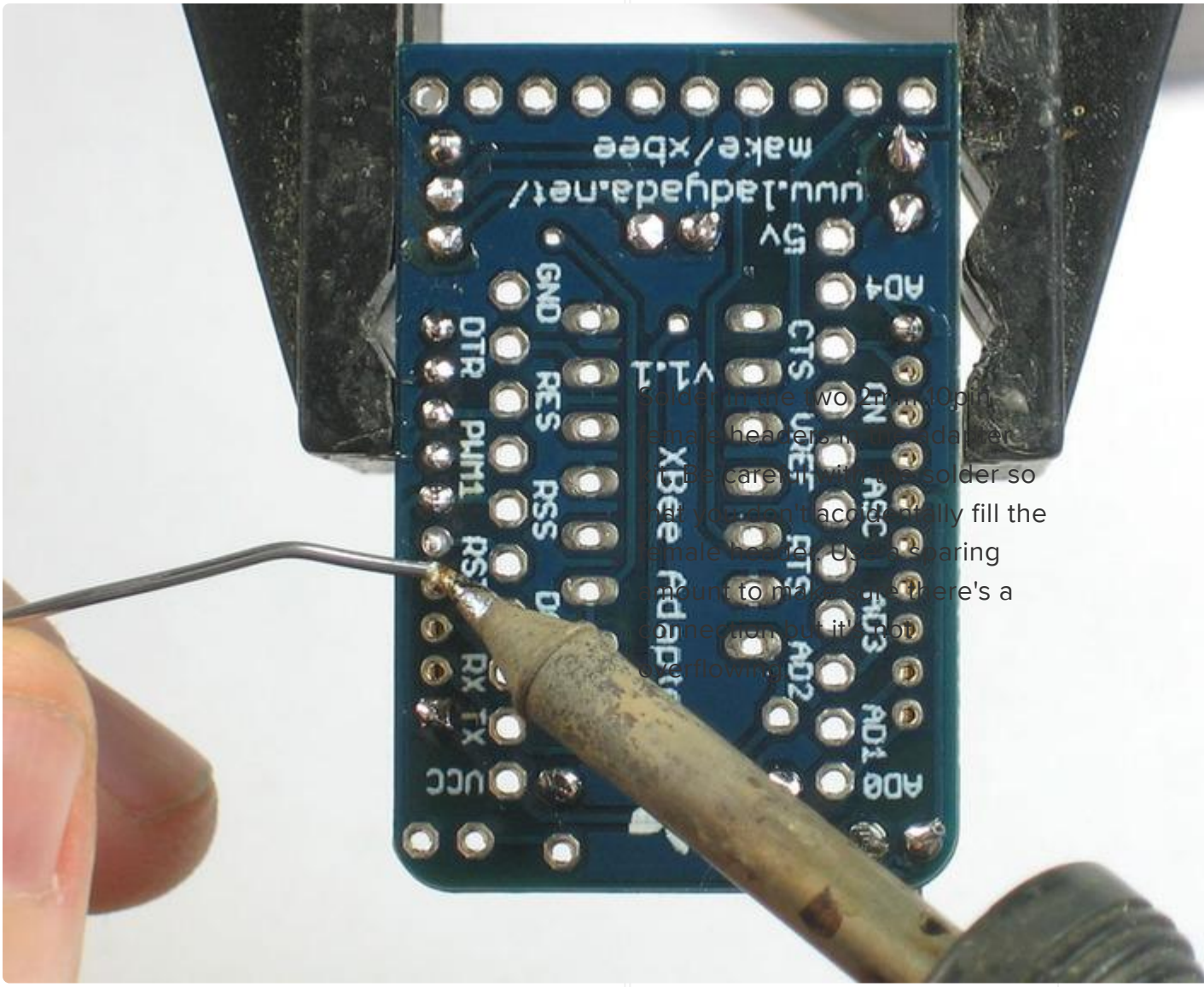

AD3
AD2
RTS
ASC
ON
CTS
AD4
5v
XBee Adapter
v1.1
DO8
RSS
RES
GND
RST
PWM1
DTR
RX



bigled.jpg

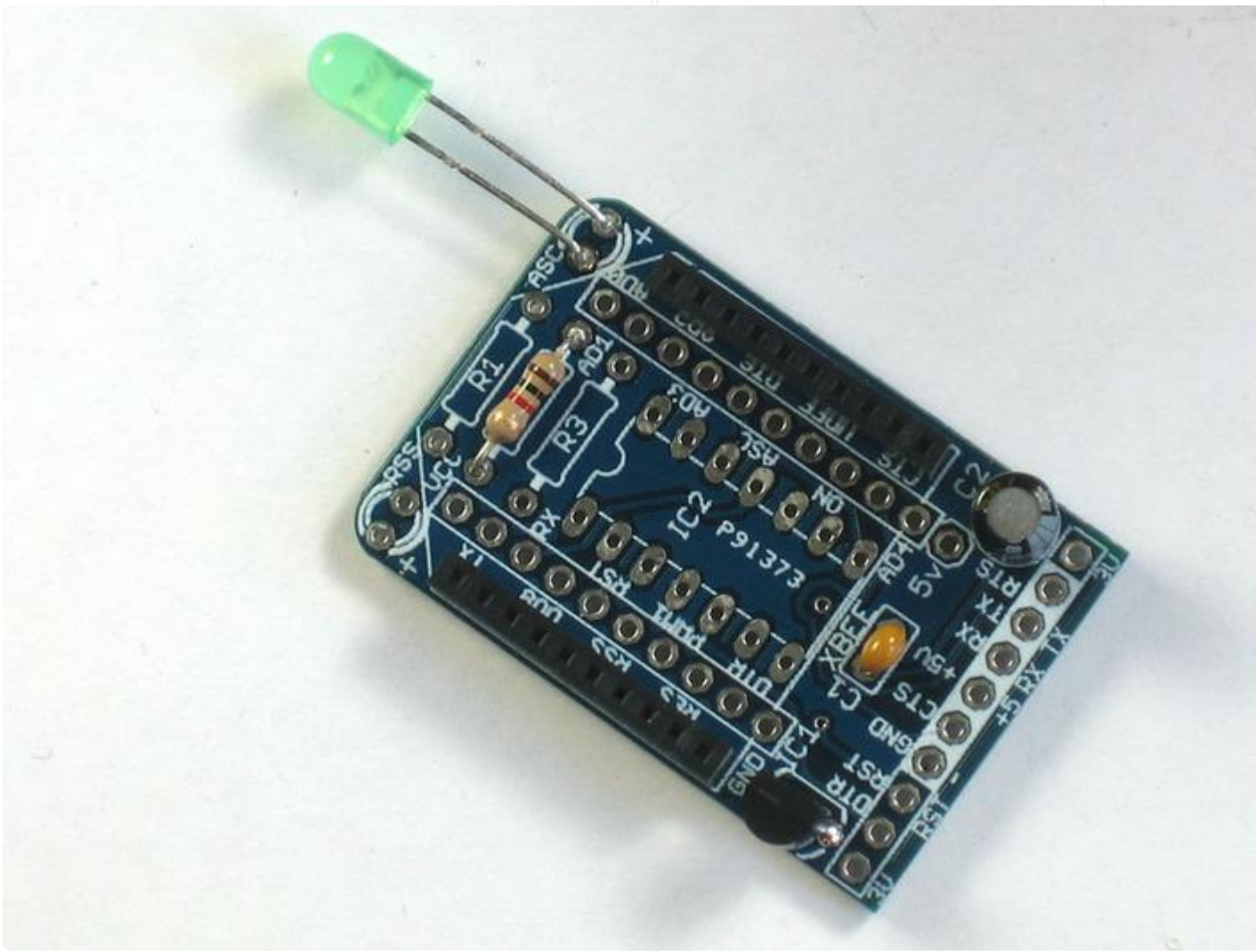
Now we will veer from the standard Kill-a-Watt adapter instructions and add a much larger LED on the ASL line so that we can easily see it blinking when it's in the Kill-a-Watt. Make sure to watch for the LED polarity, because a backwards LED will make debugging very difficult. The longer lead goes in the + marked solder hole.

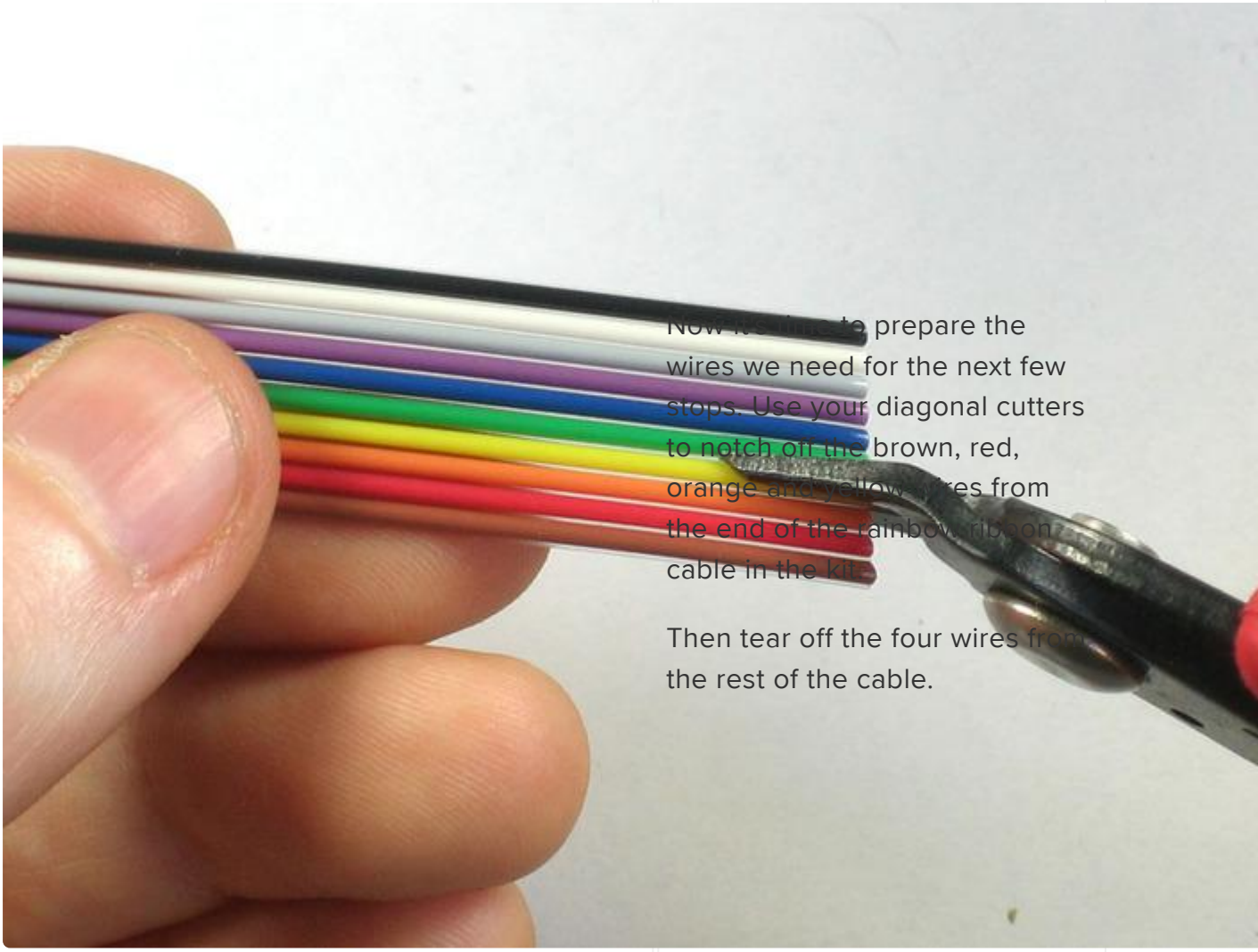
Give the LED about half an inch of space beyond the end of the PCB as shown. Also solder in the matching 1K resistor R2.



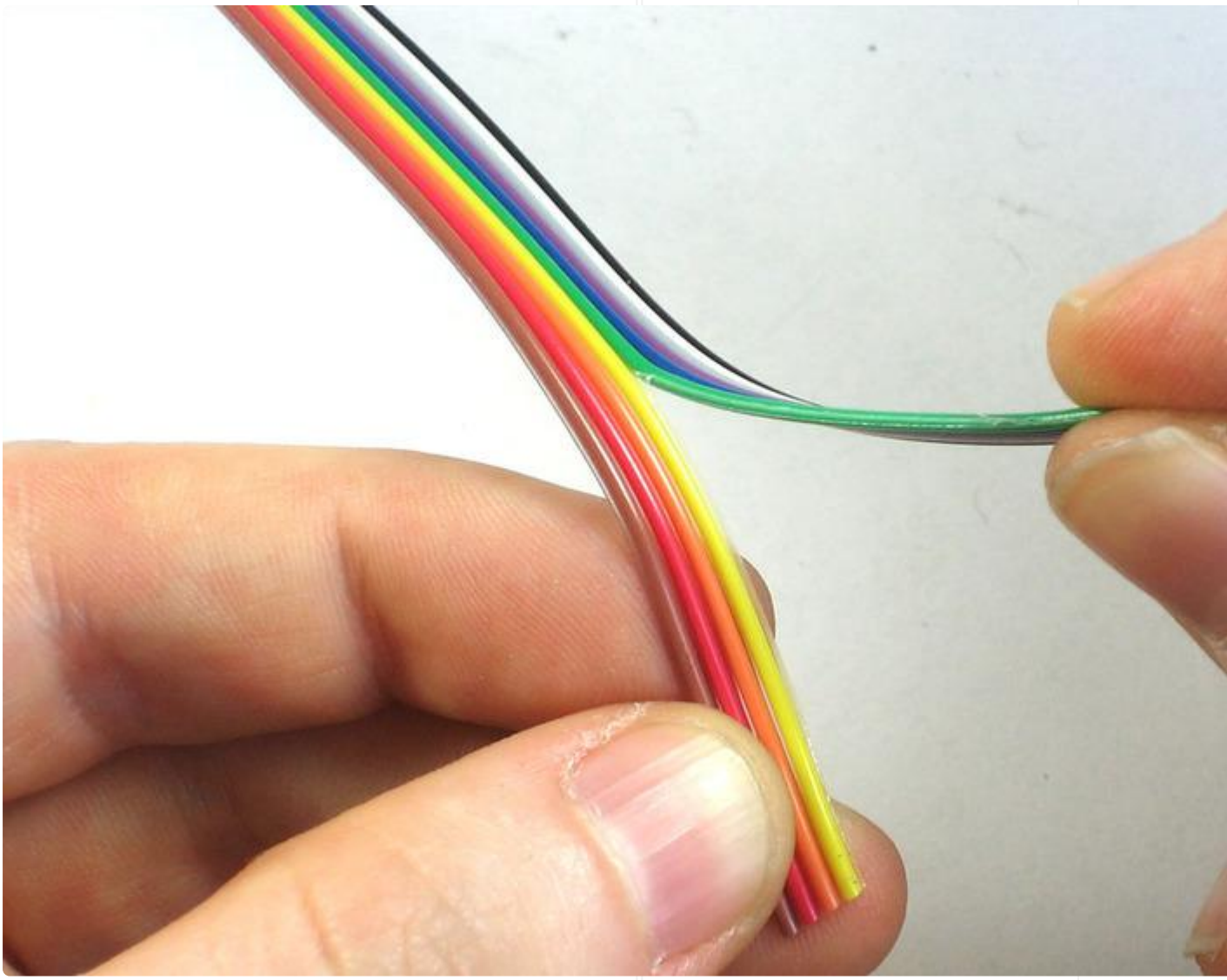
header.jpg

Solder in one 2mm 10pin female header in your adapter. Be careful with the solder so that you don't accidentally fill the female header. Use a sparing amount to make sure there's a connection but it's not overflowing.

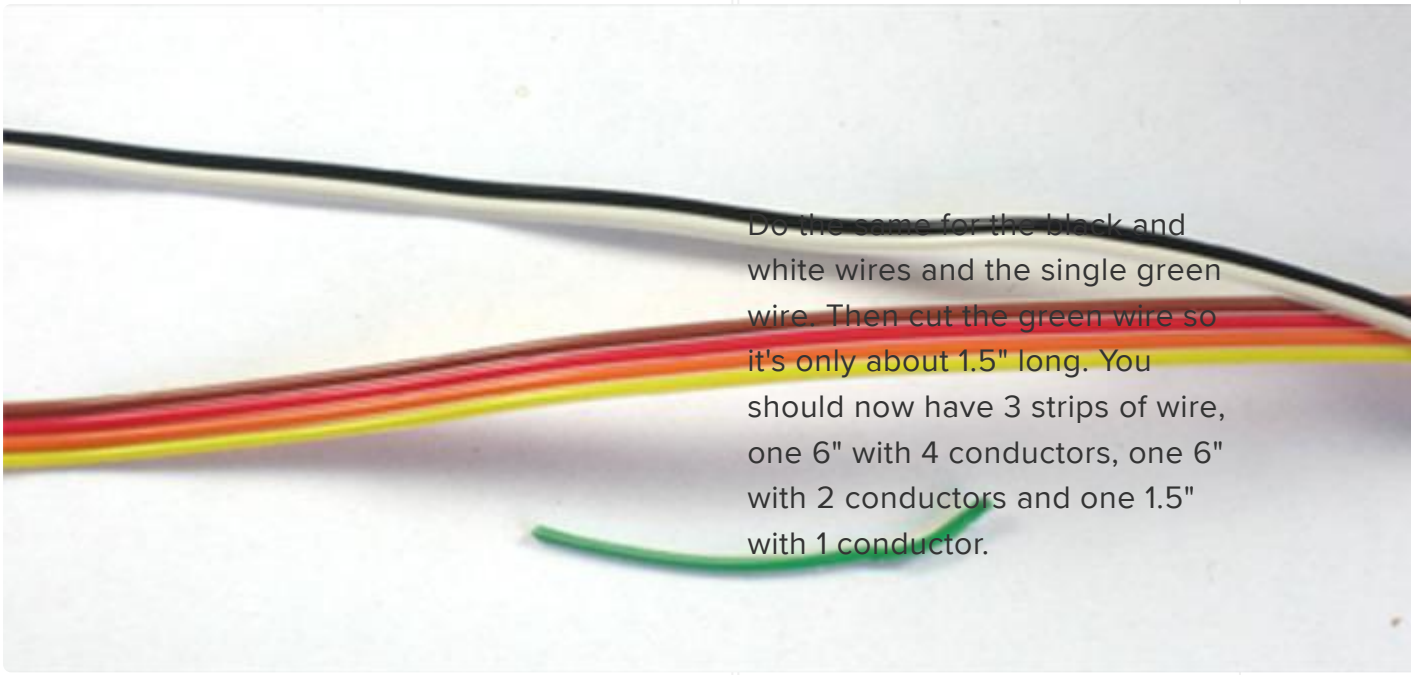

R1
R3
IC2
P91373
XBEE
GND
RST
DTR
RTS
CTS
TX
RX
+5V



ribbonclip.jpg

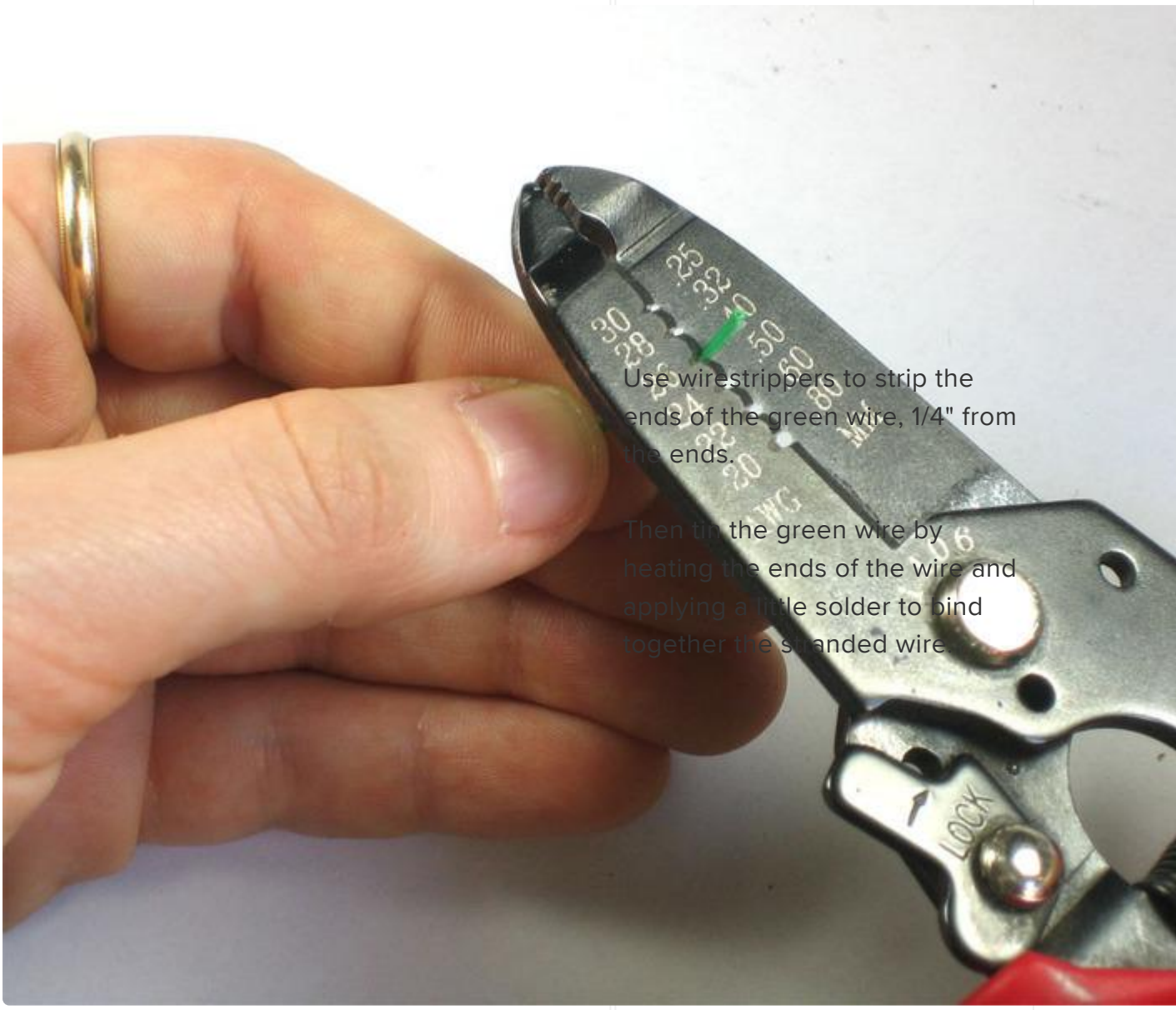


ribbontear.jpg



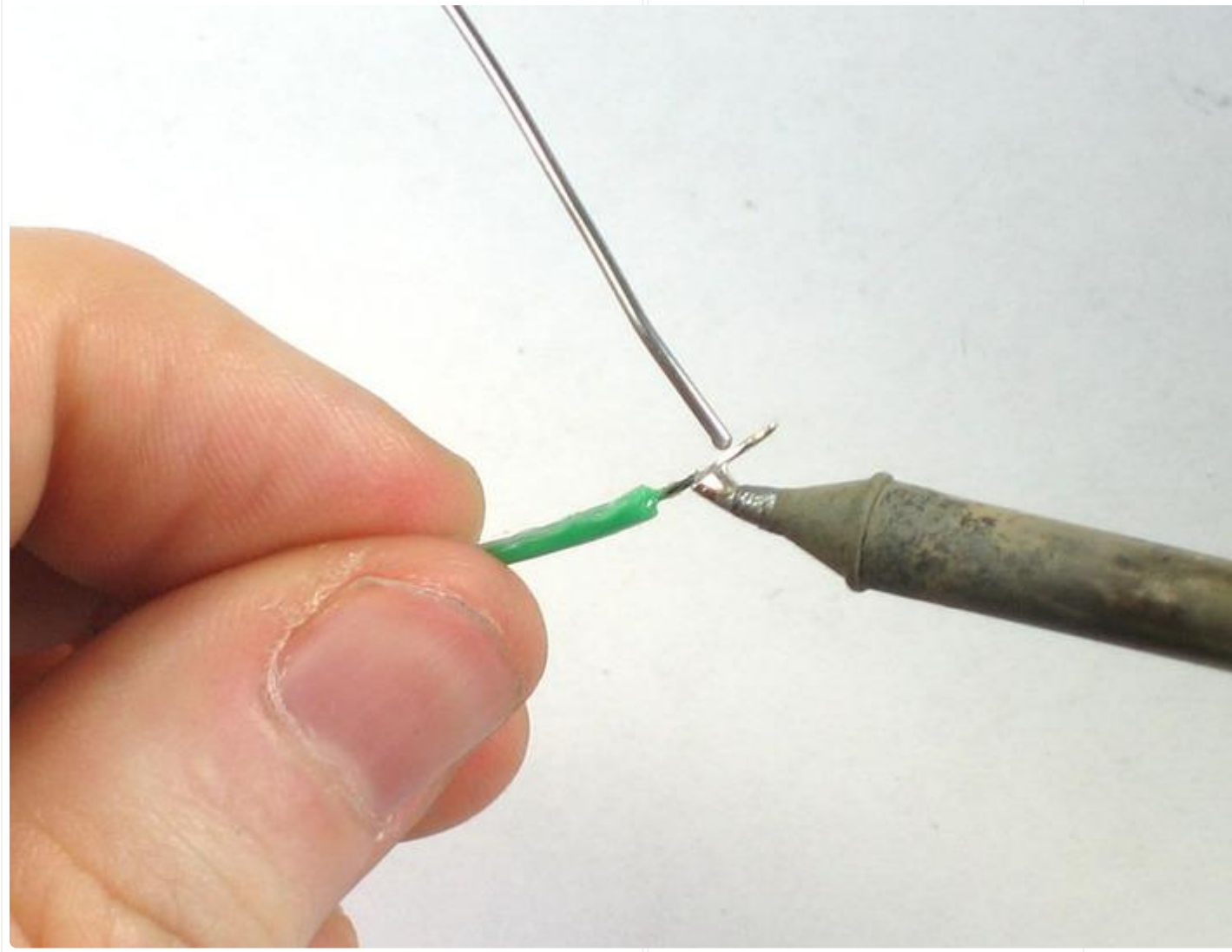
Do the same for the black and white wires and the single green wire. Then cut the green wire so it's only about 1.5" long. You should now have 3 strips of wire, one 6" with 4 conductors, one 6" with 2 conductors and one 1.5" with 1 conductor.

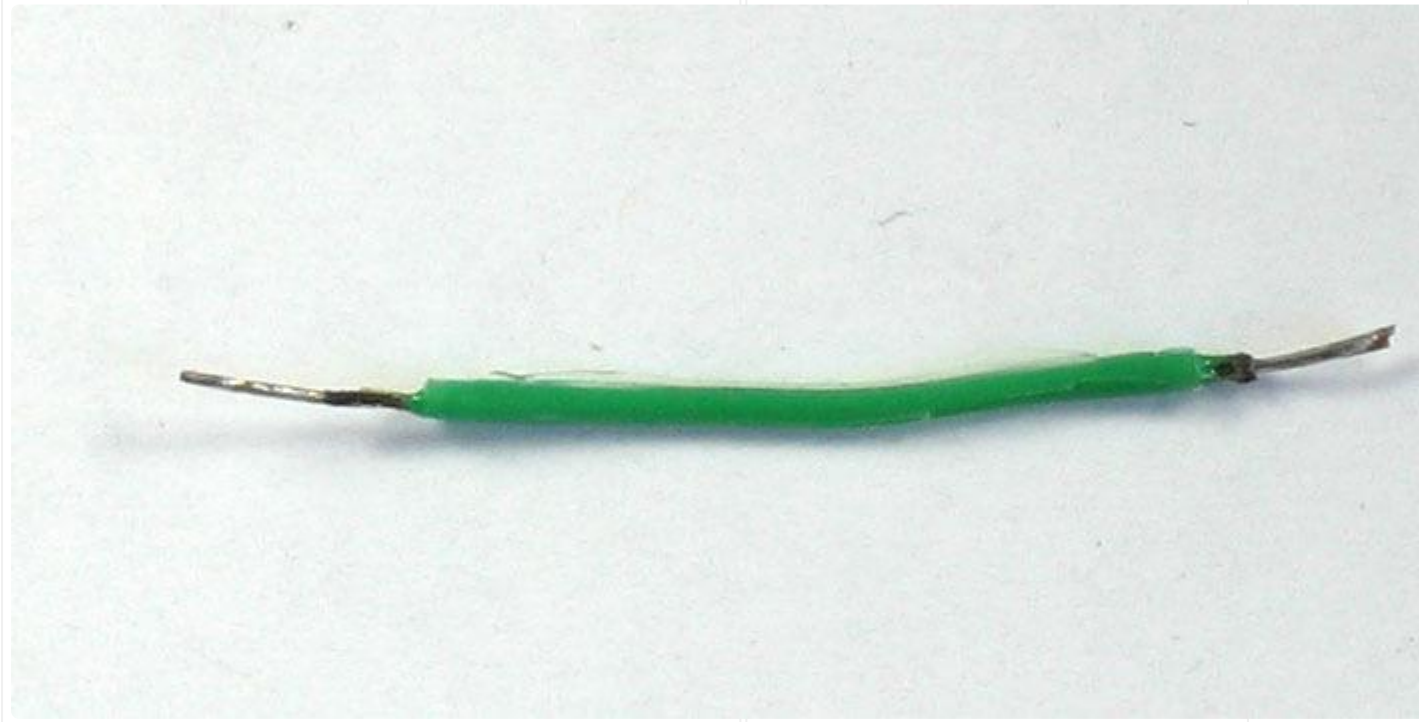
ribbonpieces.jpg

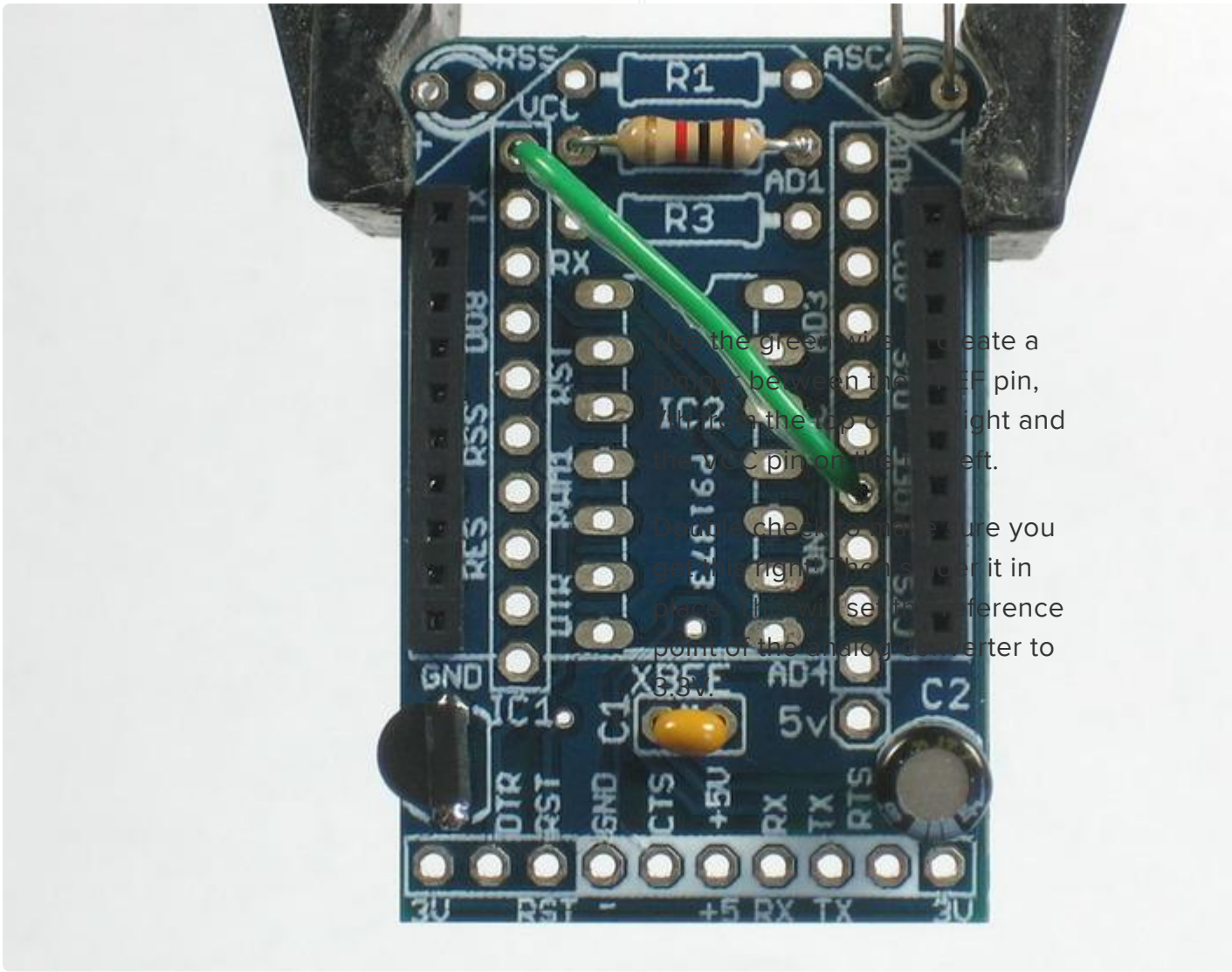


Use wirestrippers to strip the ends of the green wire, 1/4" from the ends.

Then tin the green wire by heating the ends of the wire and applying a little solder to bind together the stranded wire.



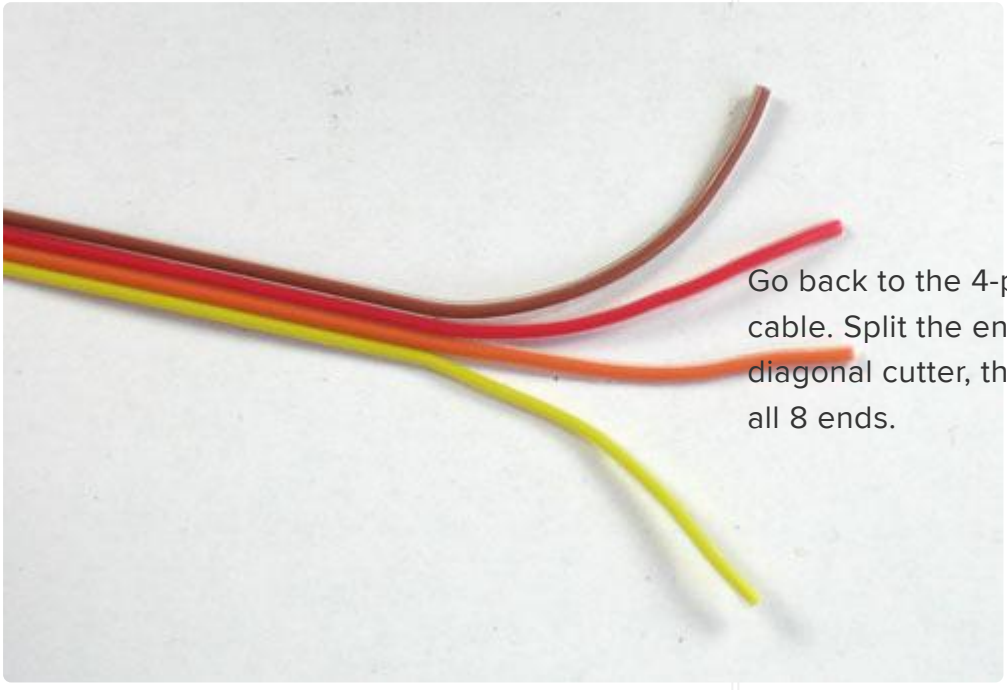




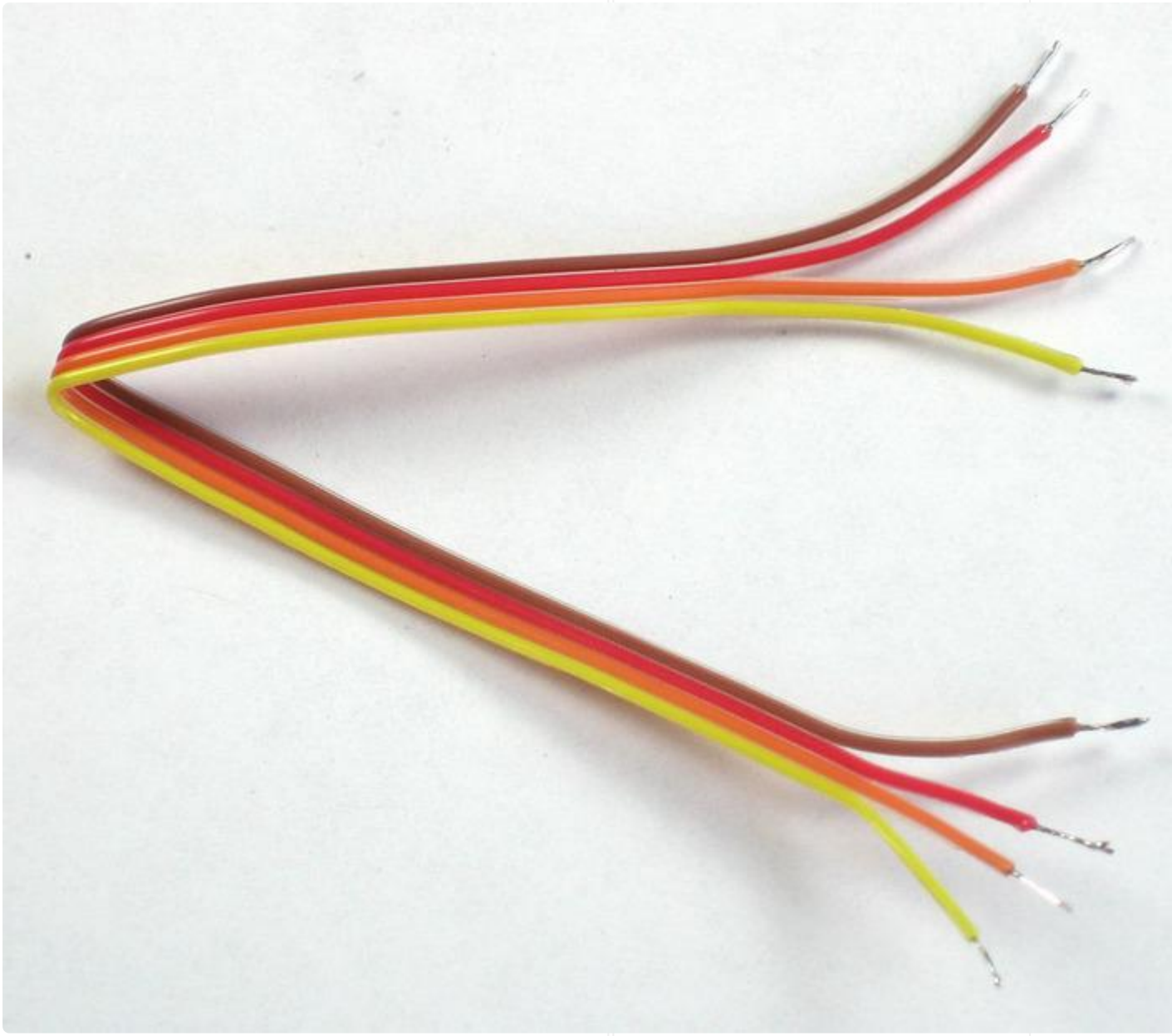
Use the green wire to create a jumper between the VREF pin, which is the top one on the right and the VCC pin on the left.

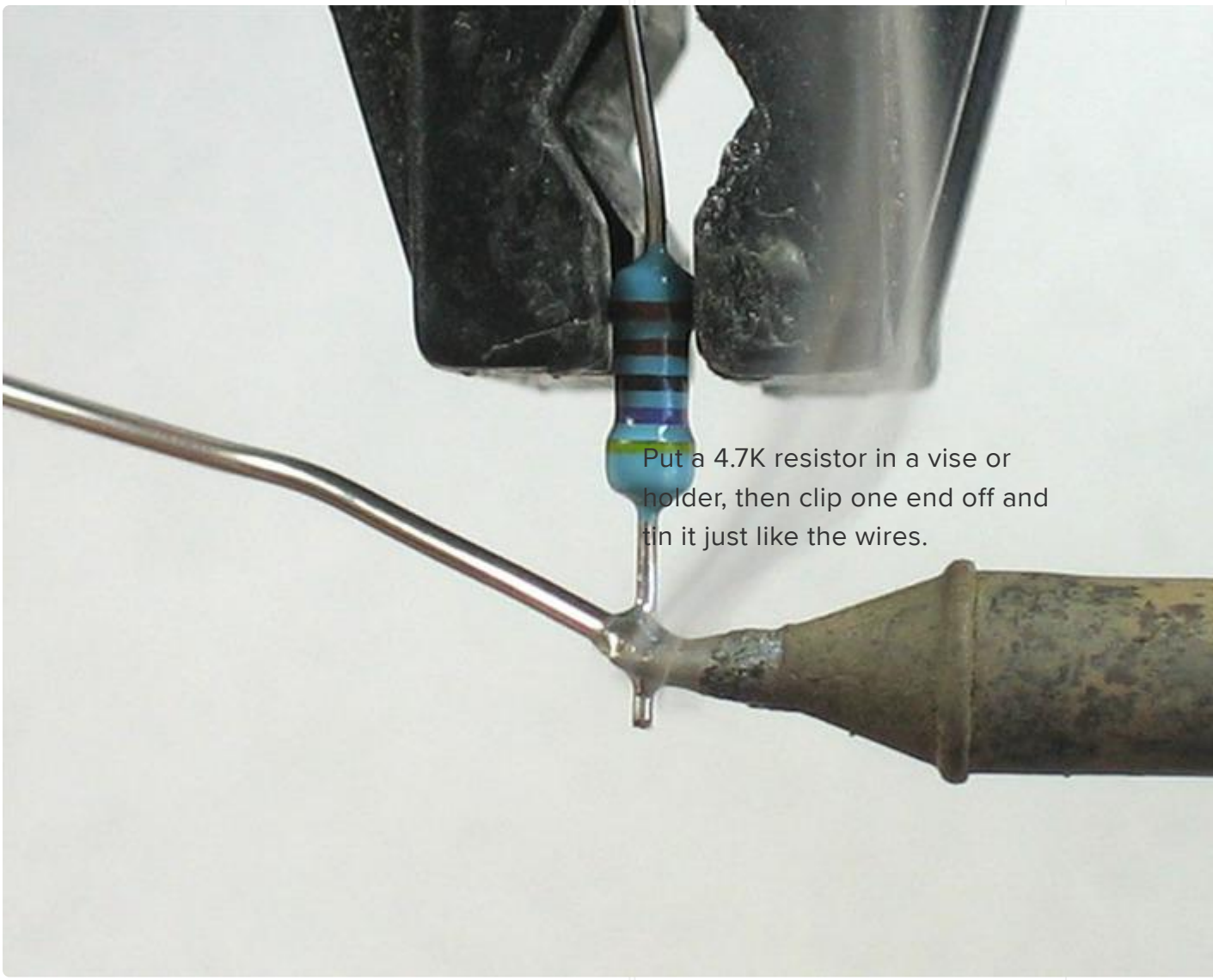
Double check to make sure you get that right before soldering it in place. This will set the reference point of the analog converter to 3.3V.

vrefjumper.jpg

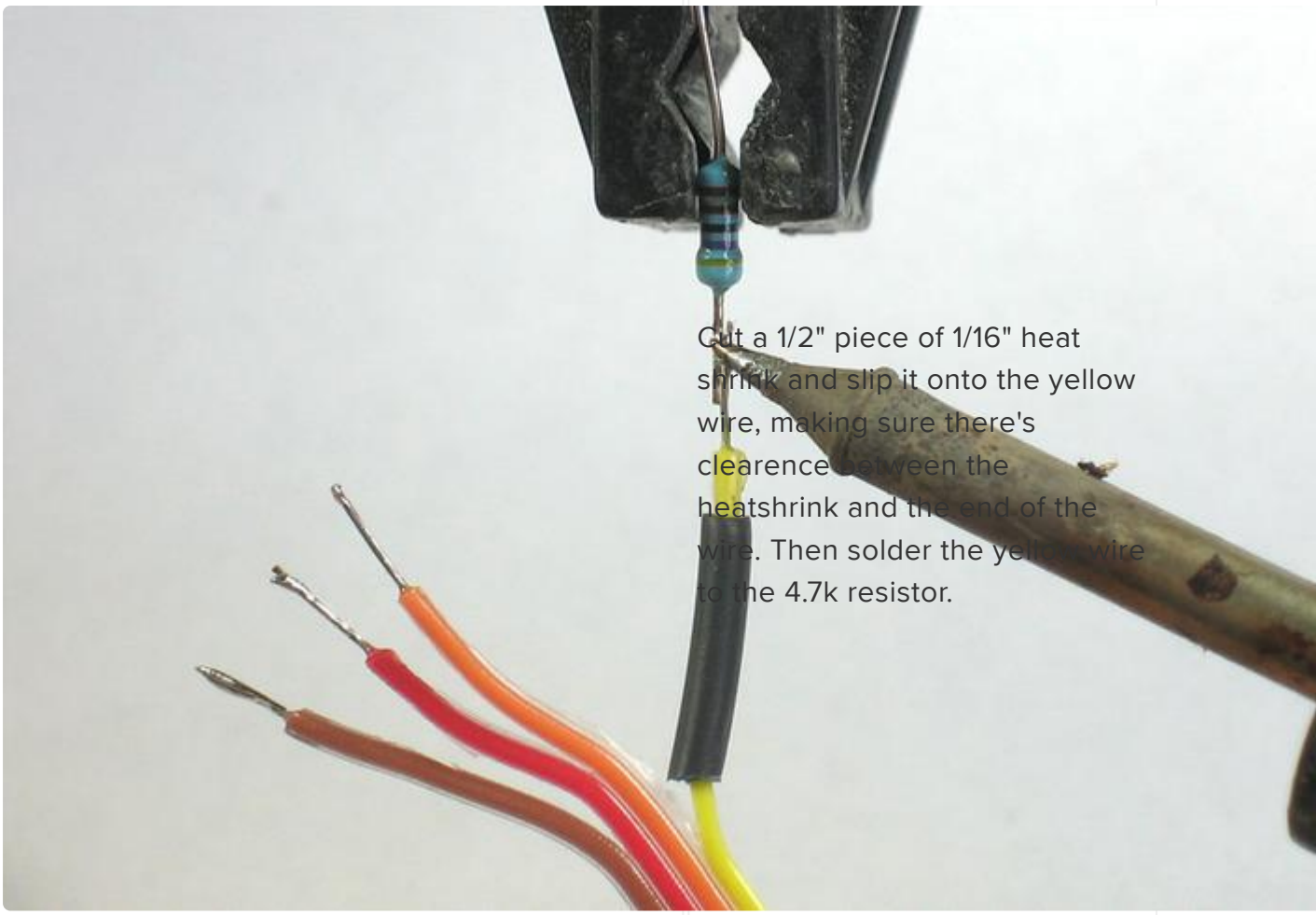


Go back to the 4-piece ribbon cable. Split the ends with the diagonal cutter, then strip and tin all 8 ends.

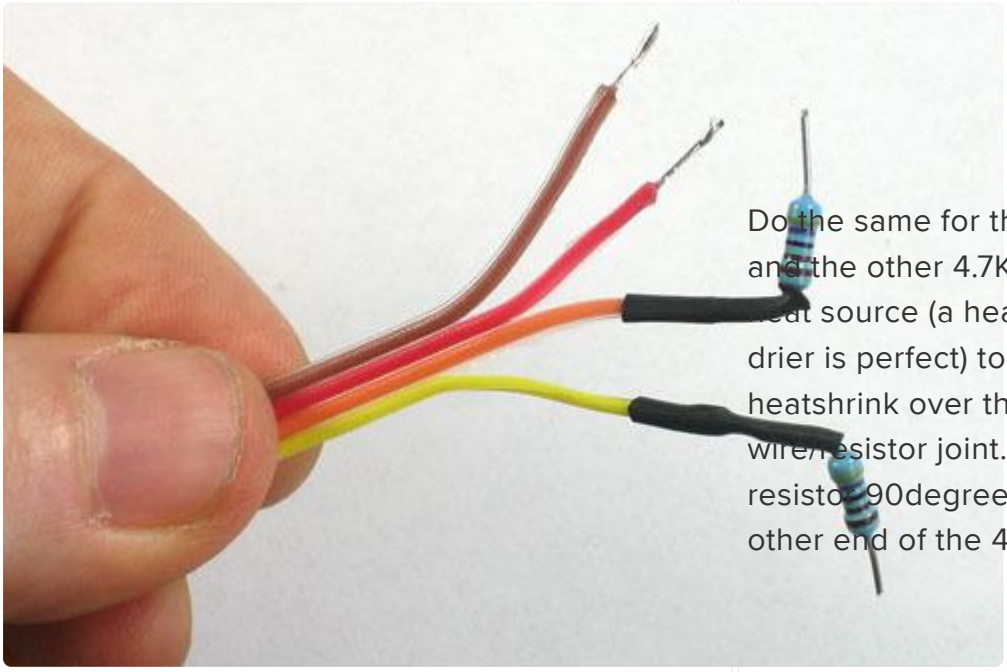




Put a 4.7K resistor in a vise or holder, then clip one end off and tin it just like the wires.

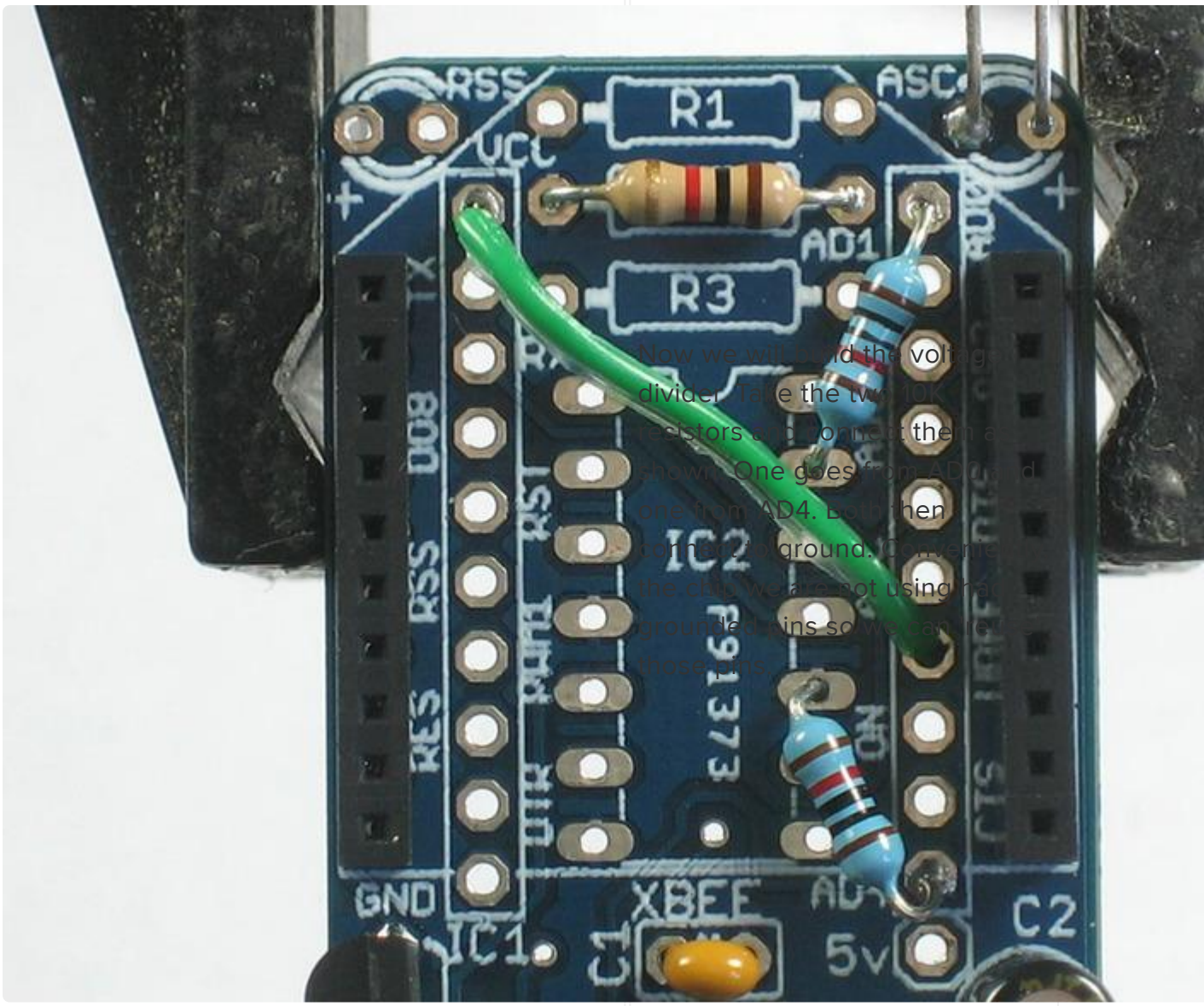


Cut a 1/2" piece of 1/16" heat shrink and slip it onto the yellow wire, making sure there's clearance between the heatshrink and the end of the wire. Then solder the yellow wire to the 4.7k resistor.

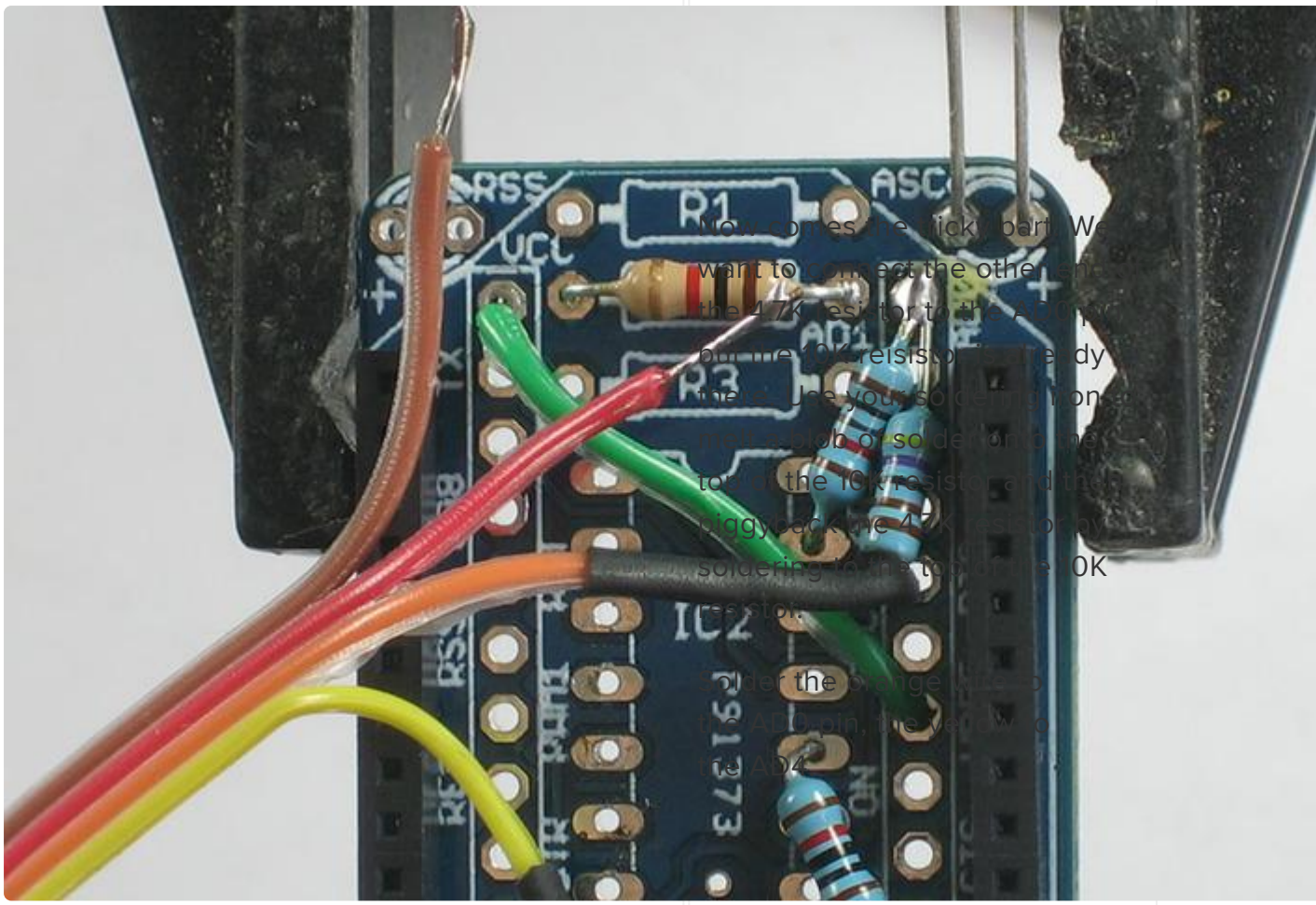


Do the same for the orange wire and the other 4.7K resistor. Use a heat source (a heat gun or hair drier is perfect) to shrink the heatshrink over the soldered wire/resistor joint. Then bend the resistor 90degrees and clip the other end of the 4.7k resistors

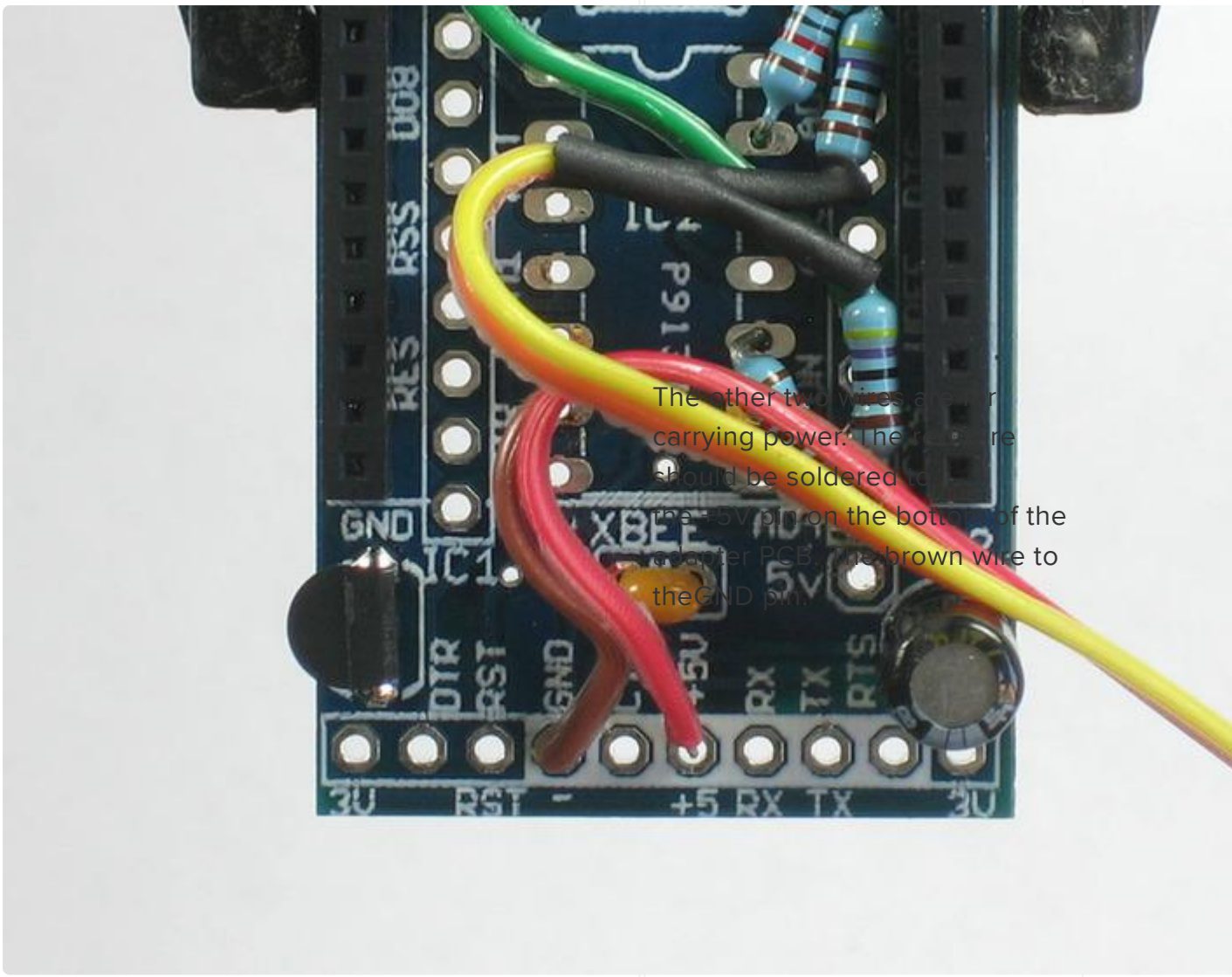
47kheat_t.jpg



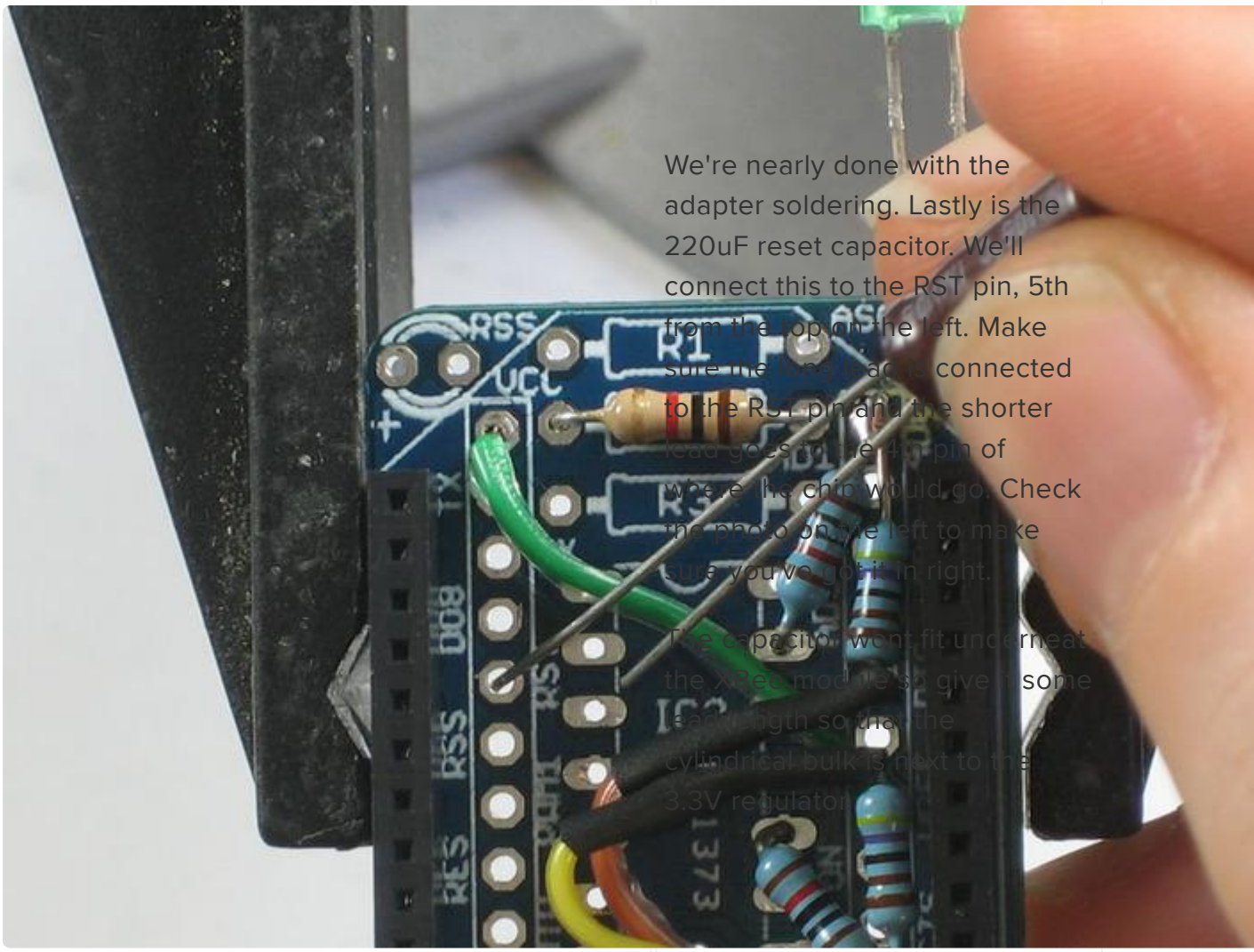
Now we will build the voltage divider. Take the two 10K resistors and connect them as shown. One goes from AD3 and one from AD4. Both then connect to ground. To prevent the chip we are not using has grounded pins so we can ignore those pins.



47kpiggy.jpg

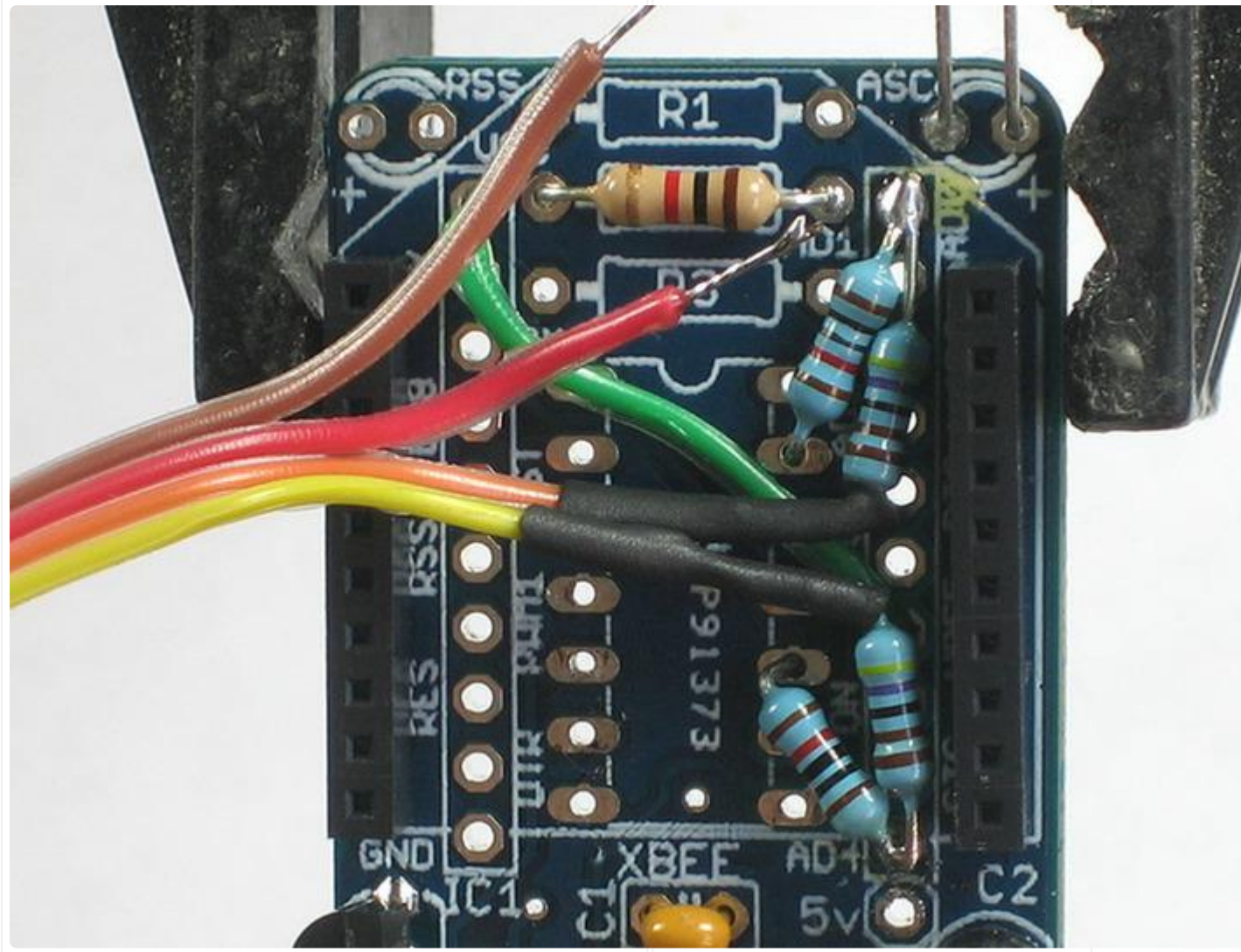


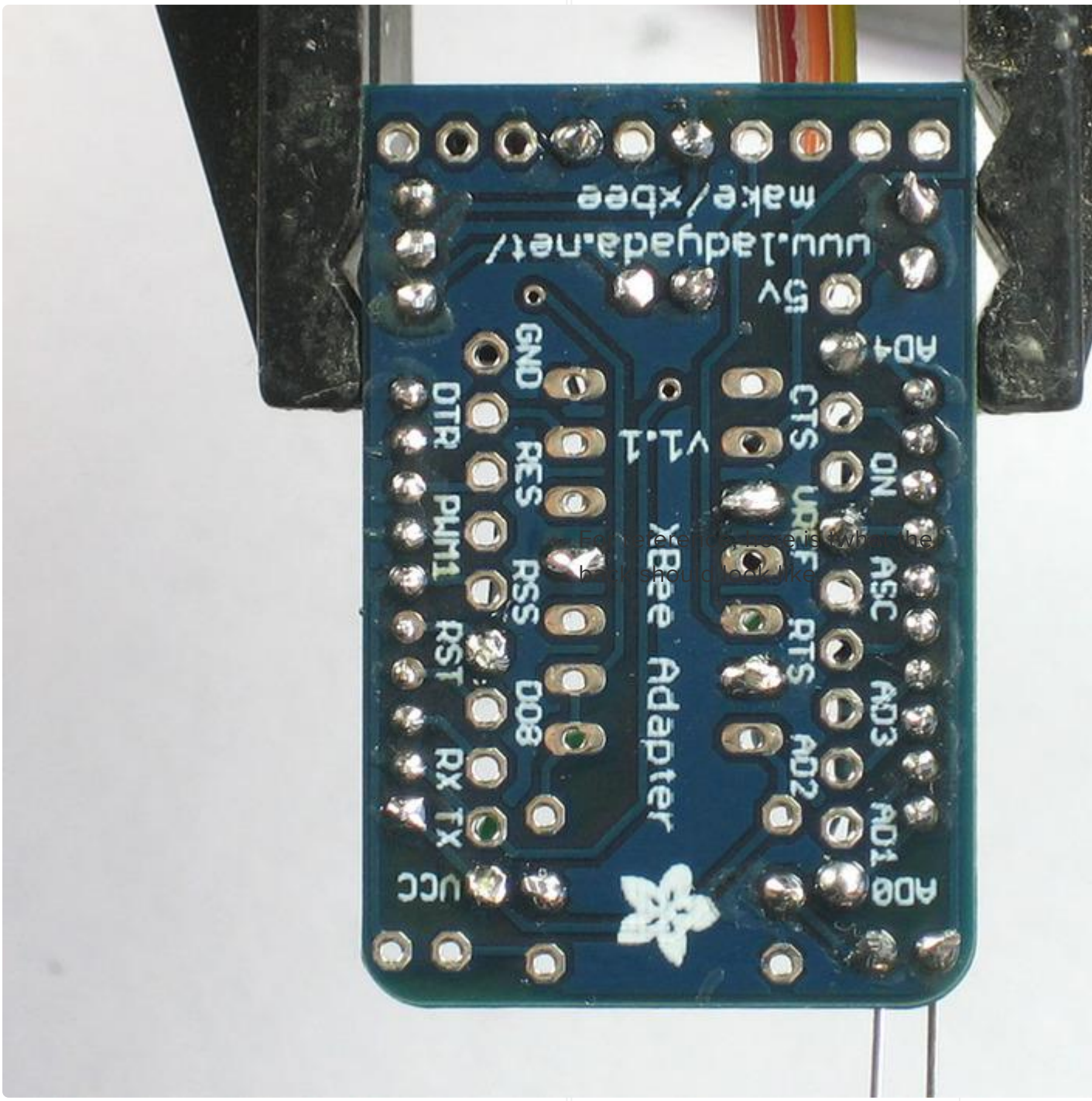
The other two wires are for carrying power. The [illegible] should be soldered to the 5V pin on the bottom of the adapter PCB. The brown wire to the GND pin.



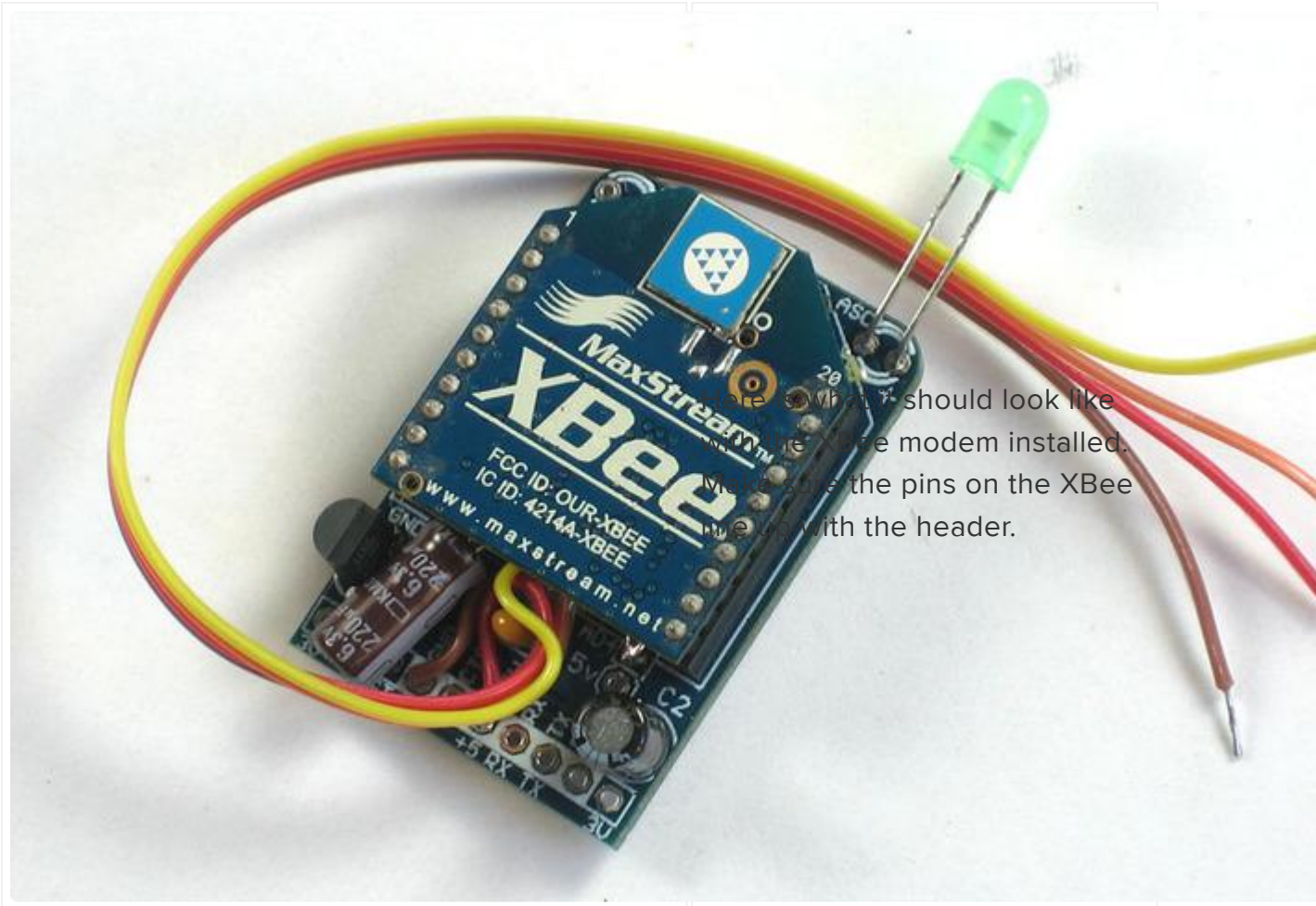
We're nearly done with the adapter soldering. Lastly is the 220uF reset capacitor. We'll connect this to the RST pin, 5th from the top on the left. Make sure the long lead is connected to the RST pin and the shorter lead goes to the 4th pin of where the chip would go. Check the photo on the left to make sure you've got it in right.

The capacitor won't fit underneat the XBee module so give it some lead length so that the cylindrical bulk is next to the 3.3V regulator.

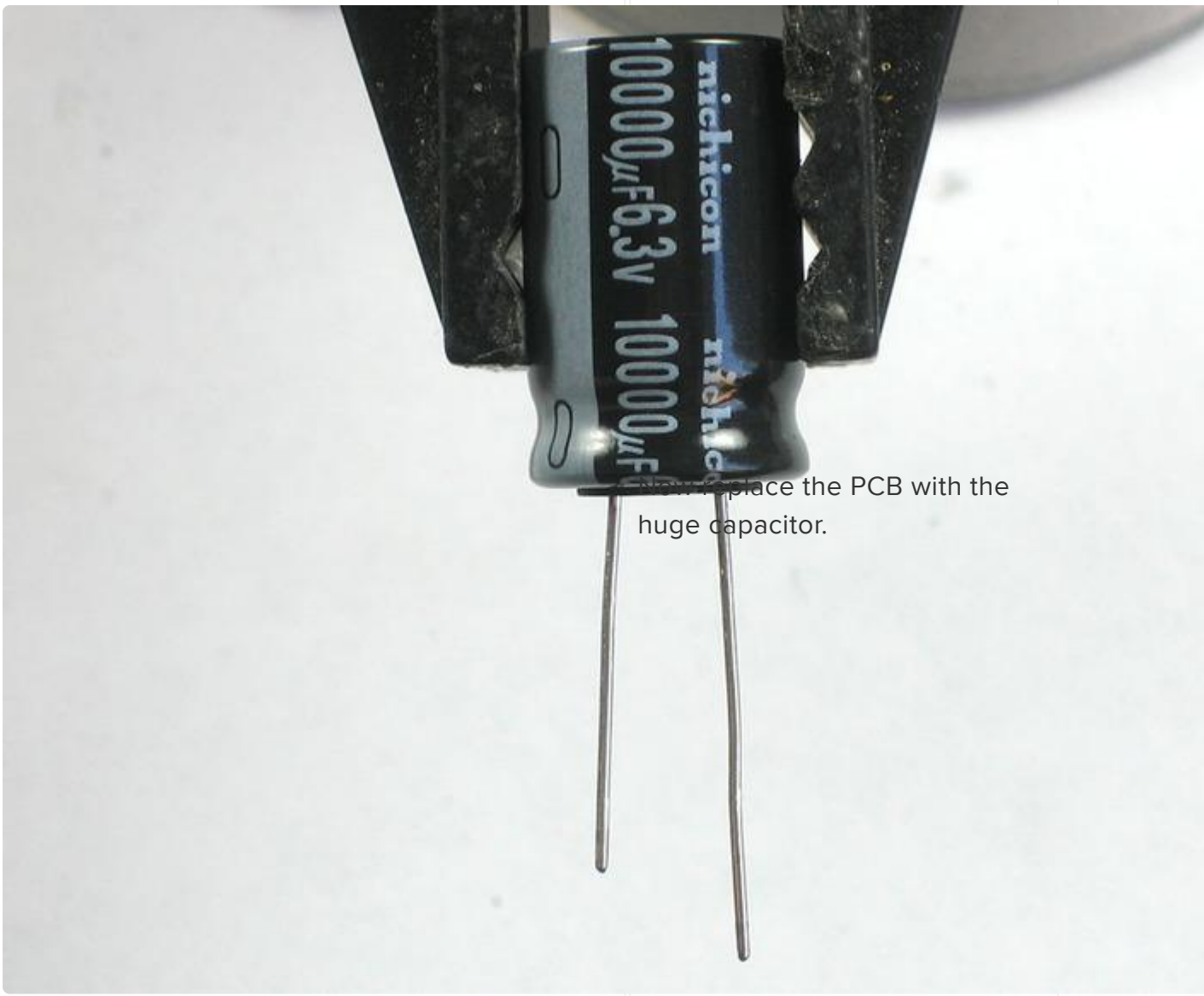

R1
RSS
GND
XBEE
5v
C1
C2
IC1
P91373



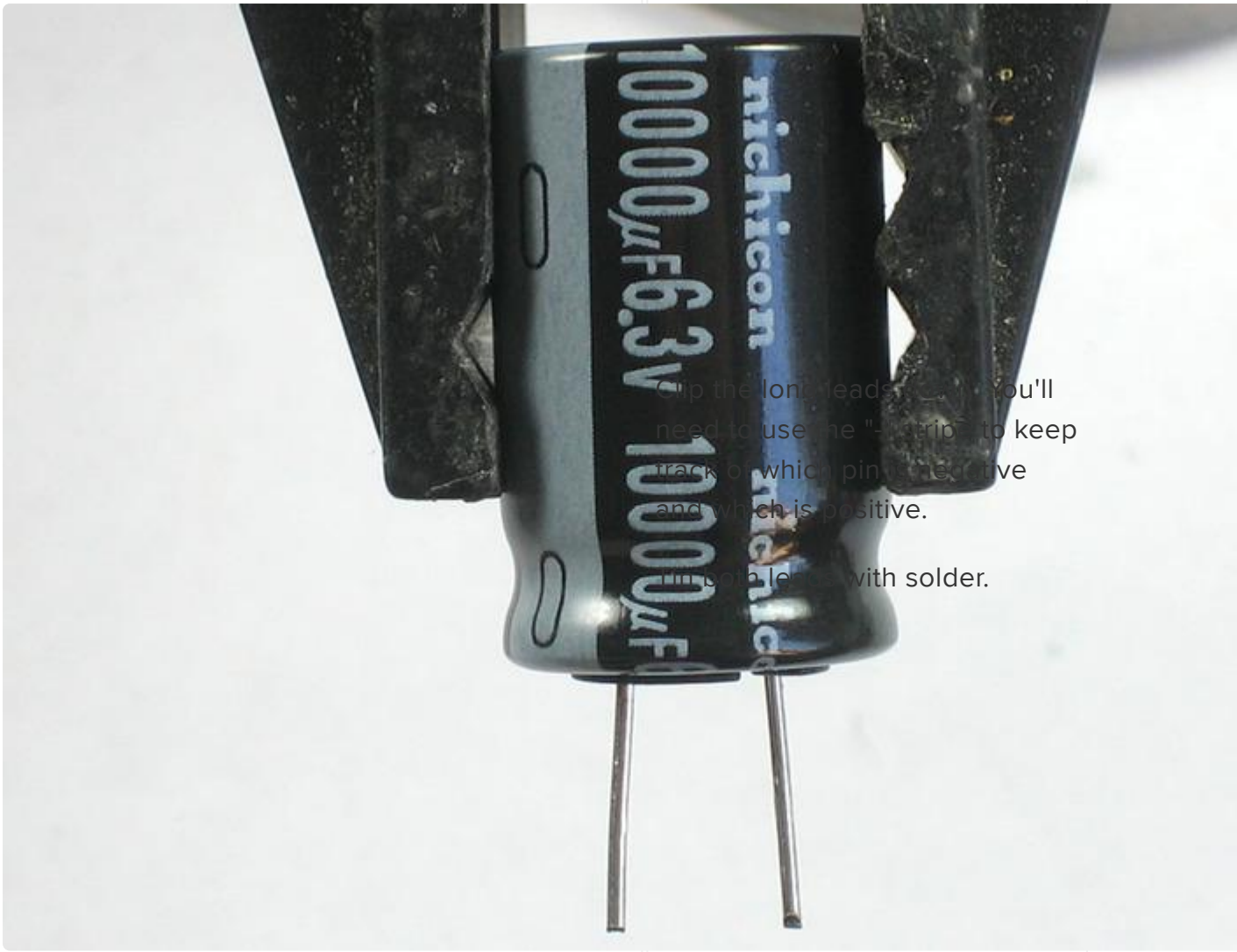
For reference, here's what the board should look like:



Here's what it should look like with the XBee modem installed. Make sure the pins on the XBee line up with the header.

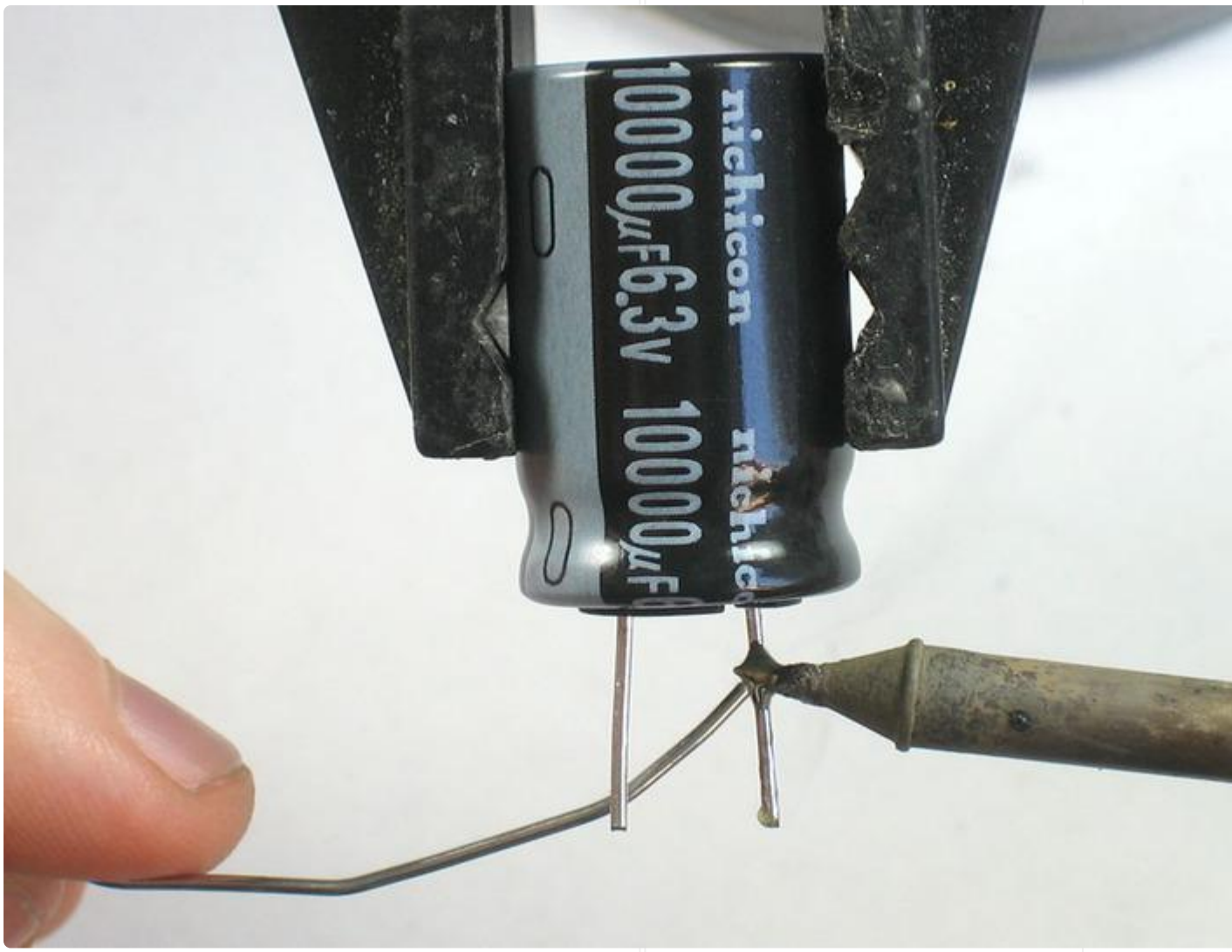


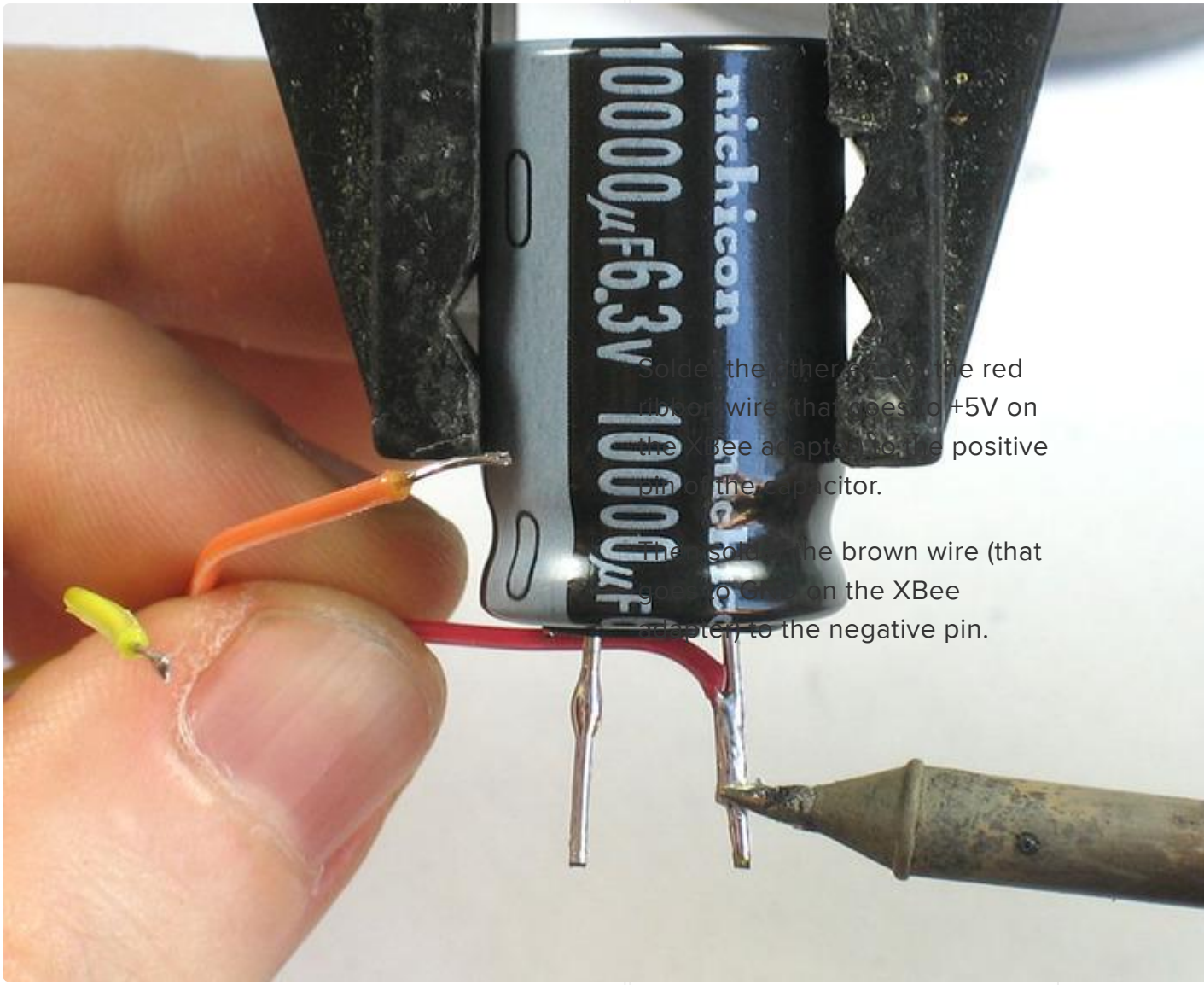
Finally, place the PCB with the huge capacitor.



Clip the long leads so you'll need to use the "stripe" to keep track of which pin is negative and which is positive.

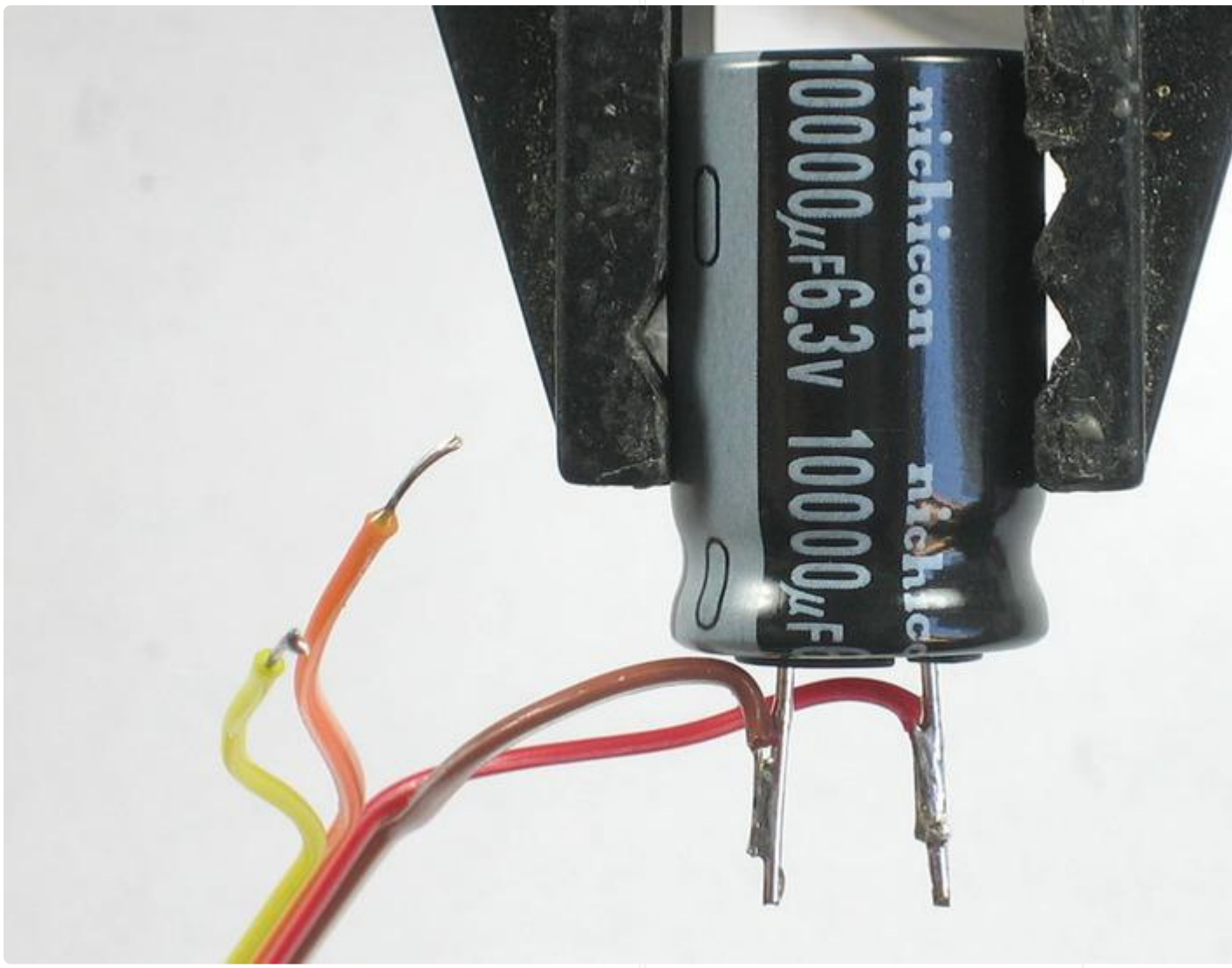
Solder both leads with solder.

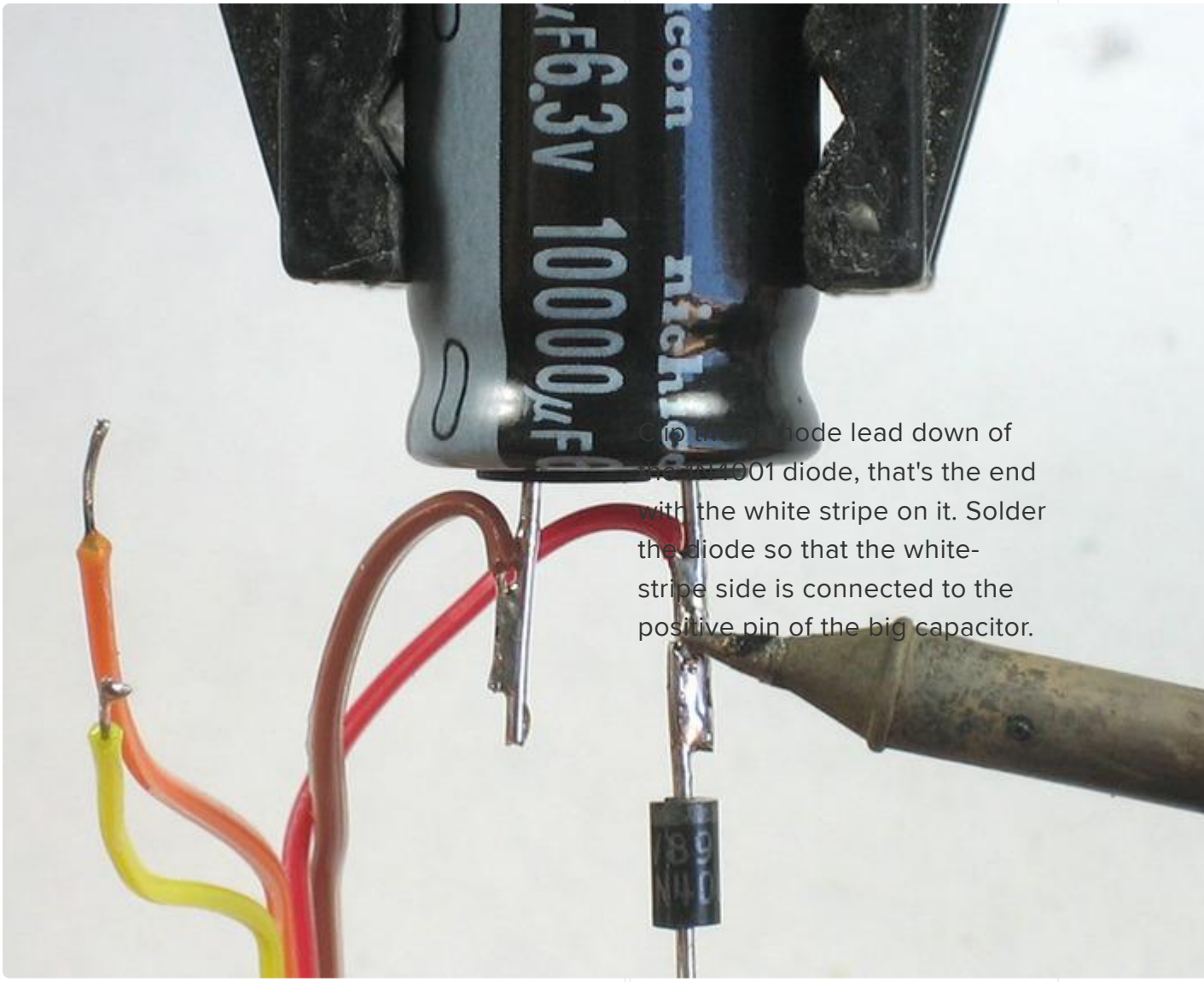

nichicon
10000µF6.3v 10000µF



Solder the other end of the red ribbon wire (that goes to +5V on the XBee adapter) to the positive pin of the capacitor.

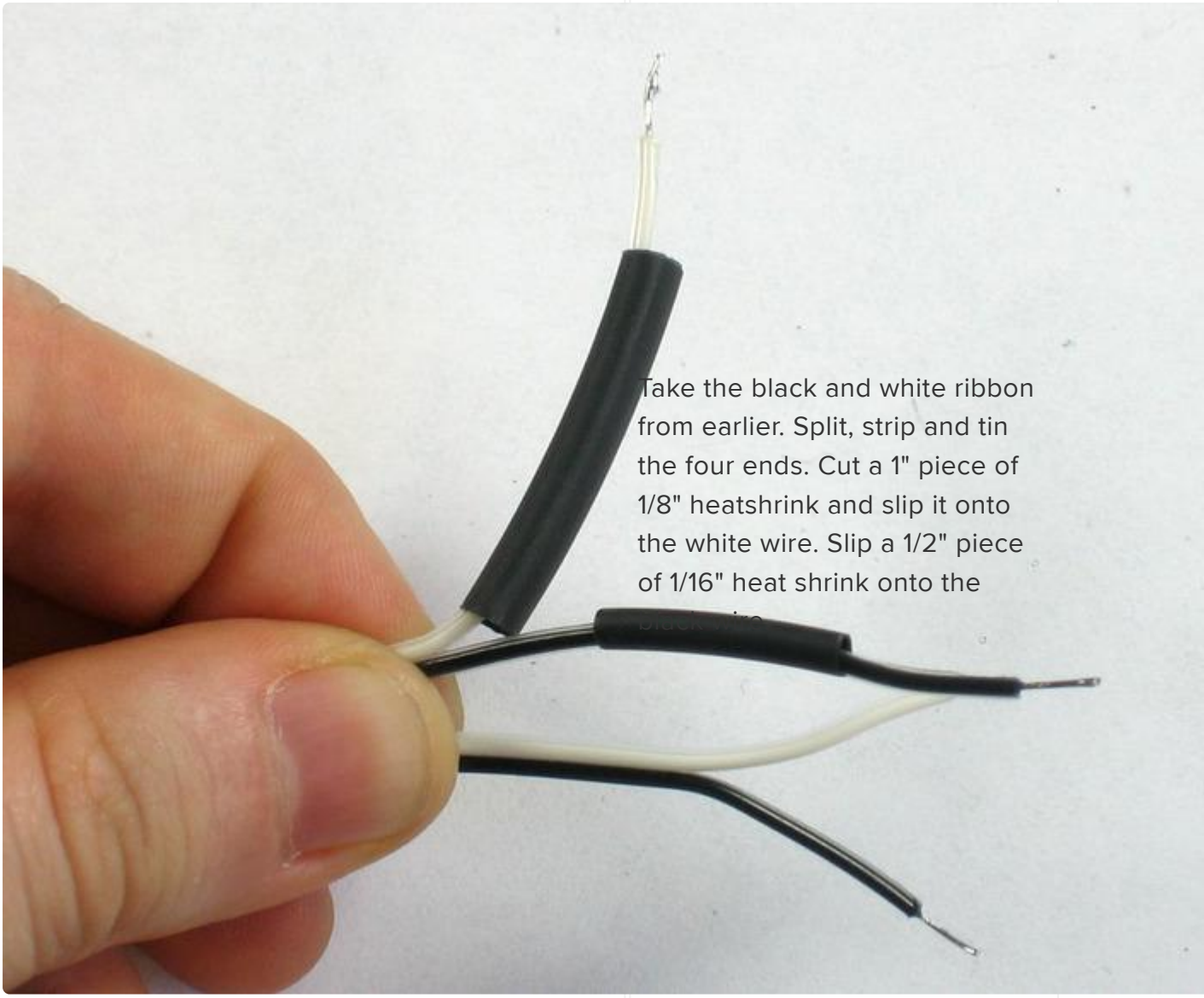
Then solder the brown wire (that goes to GND on the XBee adapter) to the negative pin.

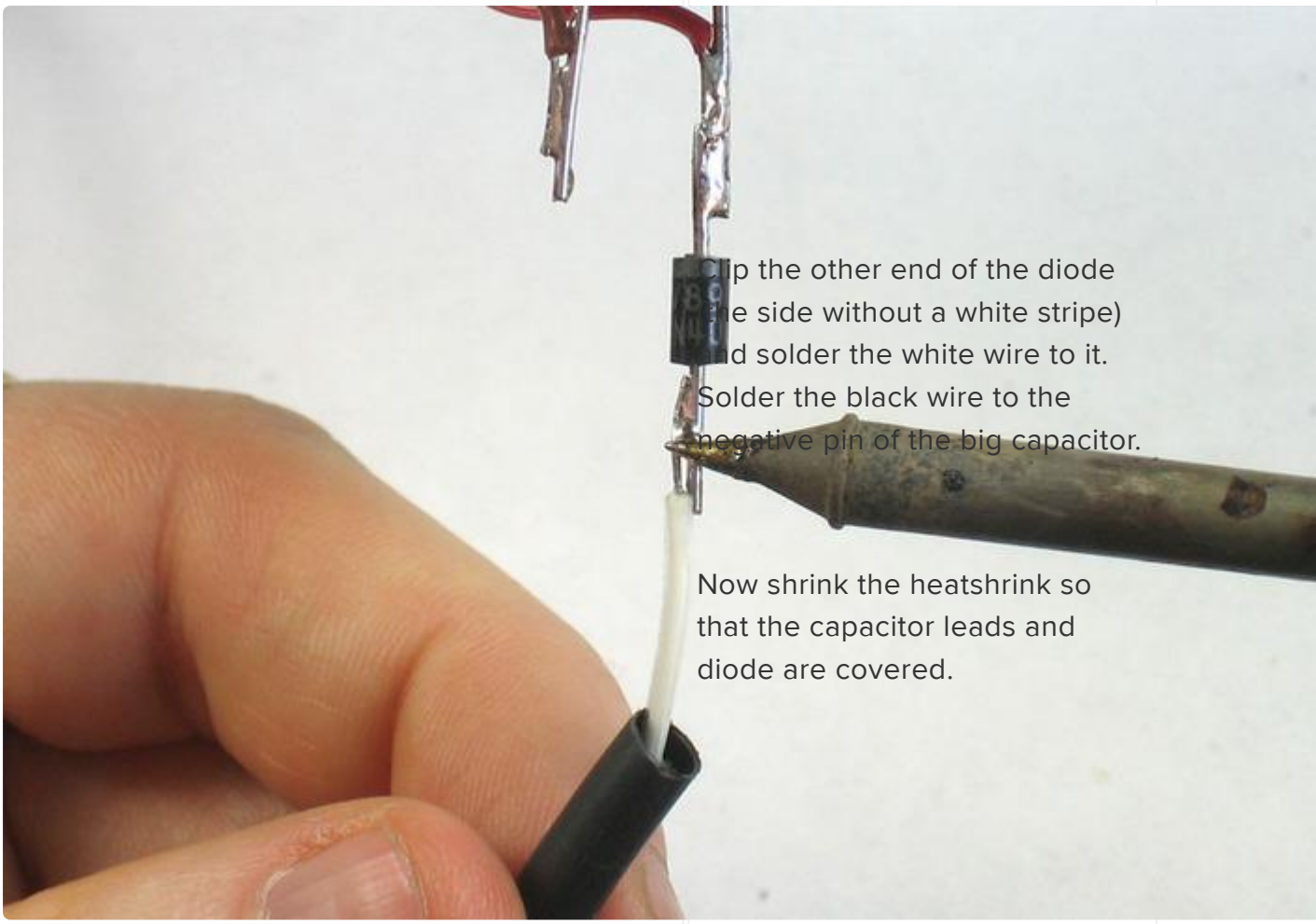

nichicon
10000µF6.3v 10000µF



bfcdiode.jpg

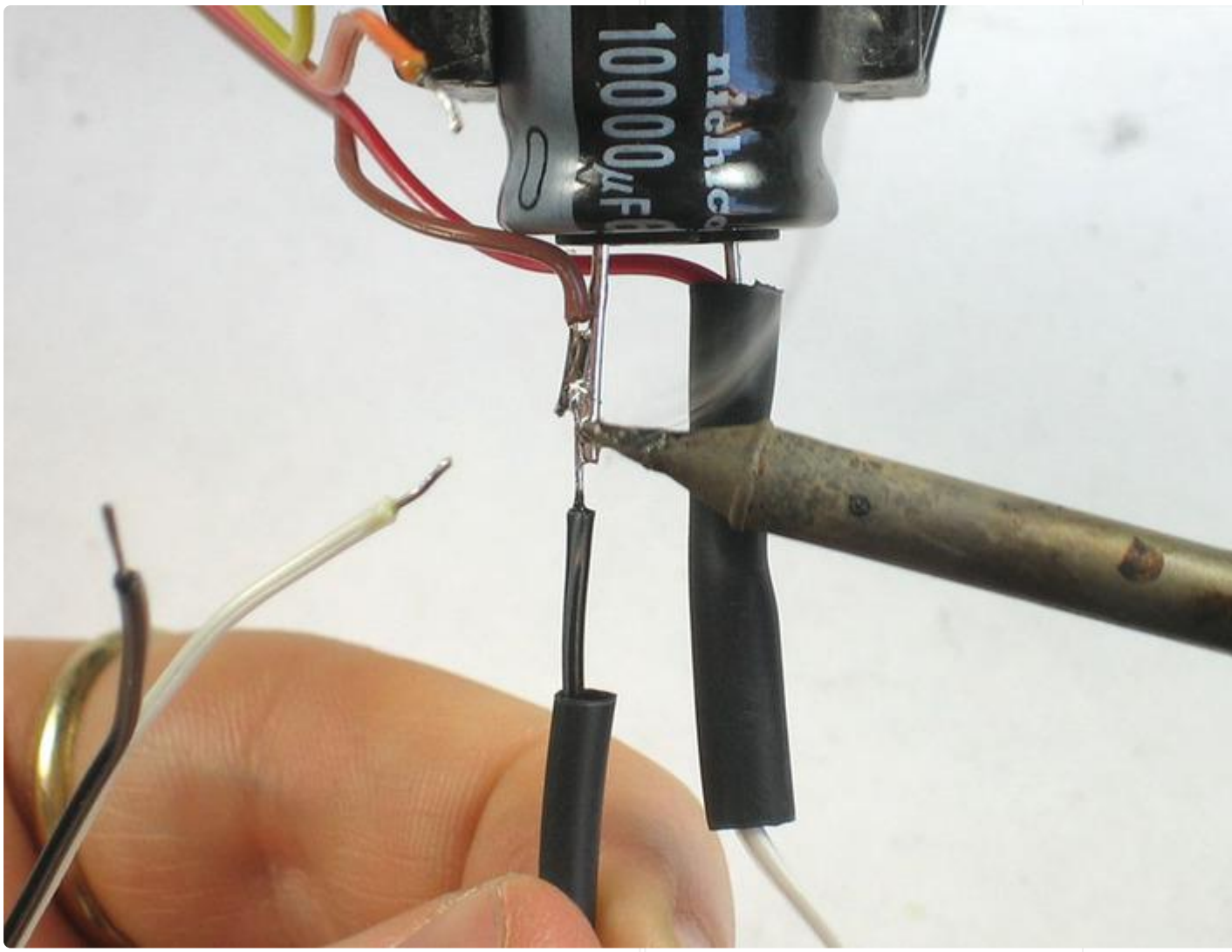
Clip the cathode lead down of the 1N4001 diode, that's the end with the white stripe on it. Solder the diode so that the white-stripe side is connected to the positive pin of the big capacitor.

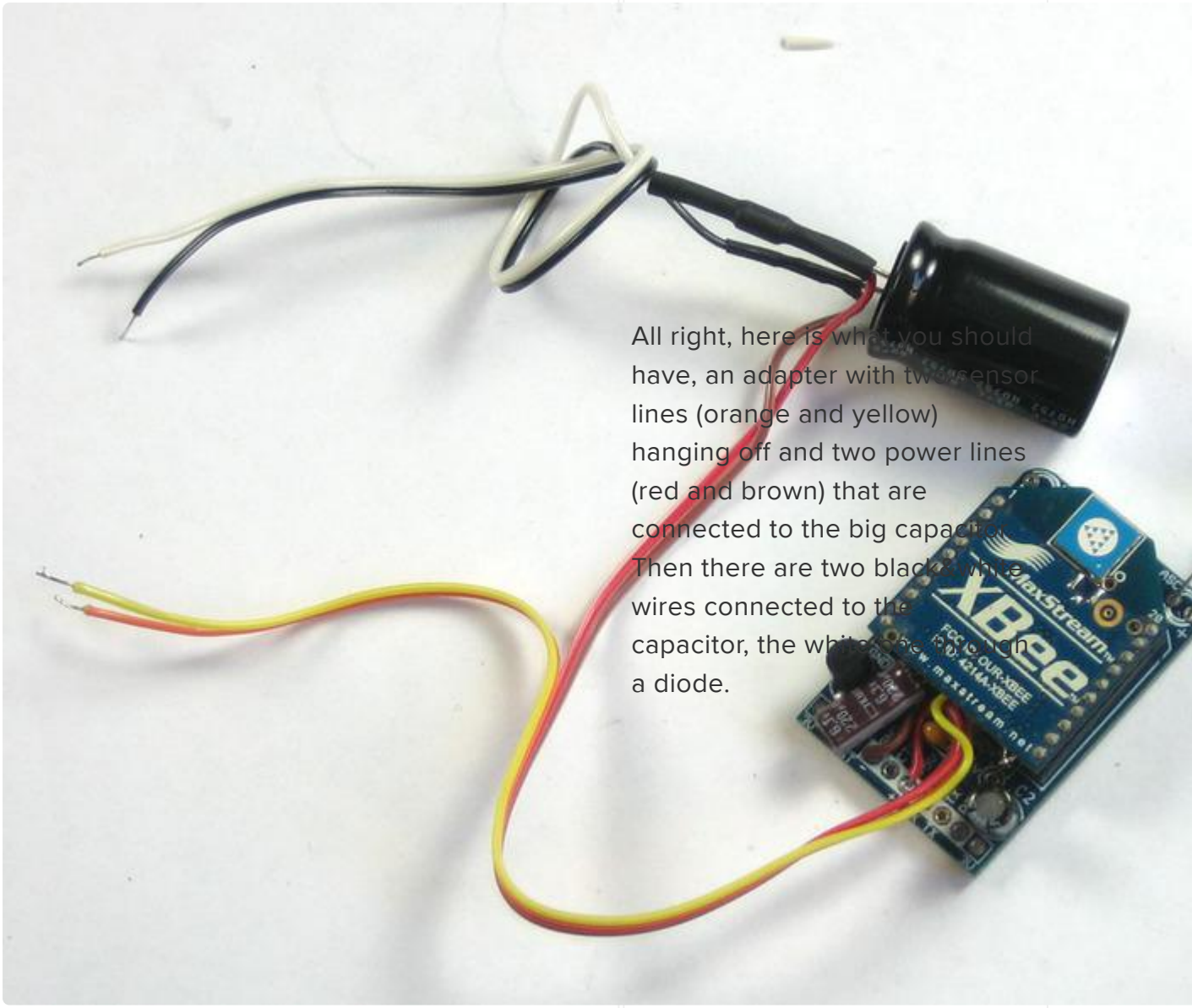

Take the black and white ribbon from earlier. Split, strip and tin the four ends. Cut a 1" piece of 1/8" heatshrink and slip it onto the white wire. Slip a 1/2" piece of 1/16" heat shrink onto the



Clip the other end of the diode (the side without a white stripe) and solder the white wire to it. Solder the black wire to the negative pin of the big capacitor.

Now shrink the heatshrink so that the capacitor leads and diode are covered.


10000µF



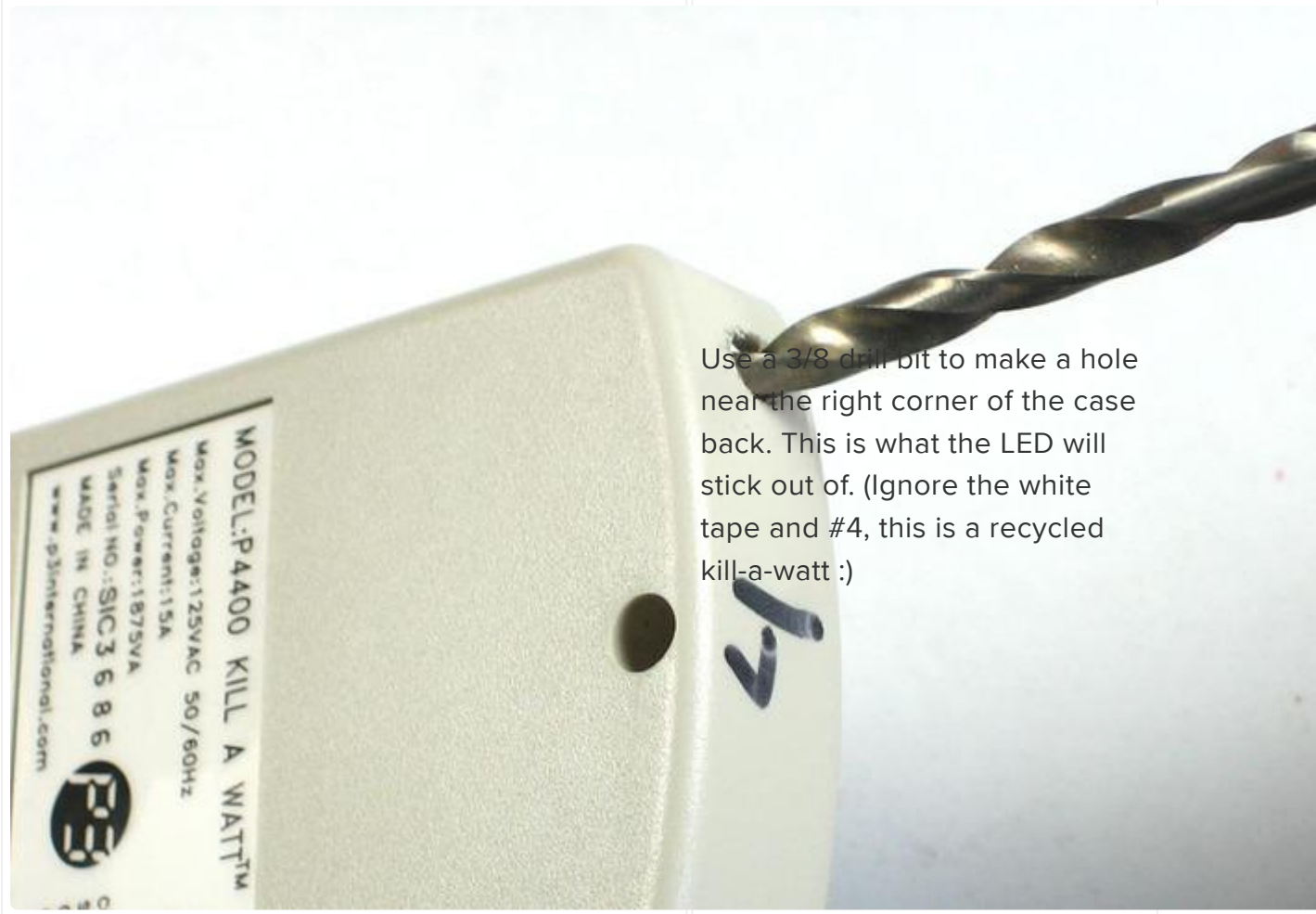
All right, here is what you should have, an adapter with two sensor lines (orange and yellow) hanging off and two power lines (red and brown) that are connected to the big capacitor. Then there are two black/white wires connected to the capacitor, the white one through a diode.

openkaw.jpg

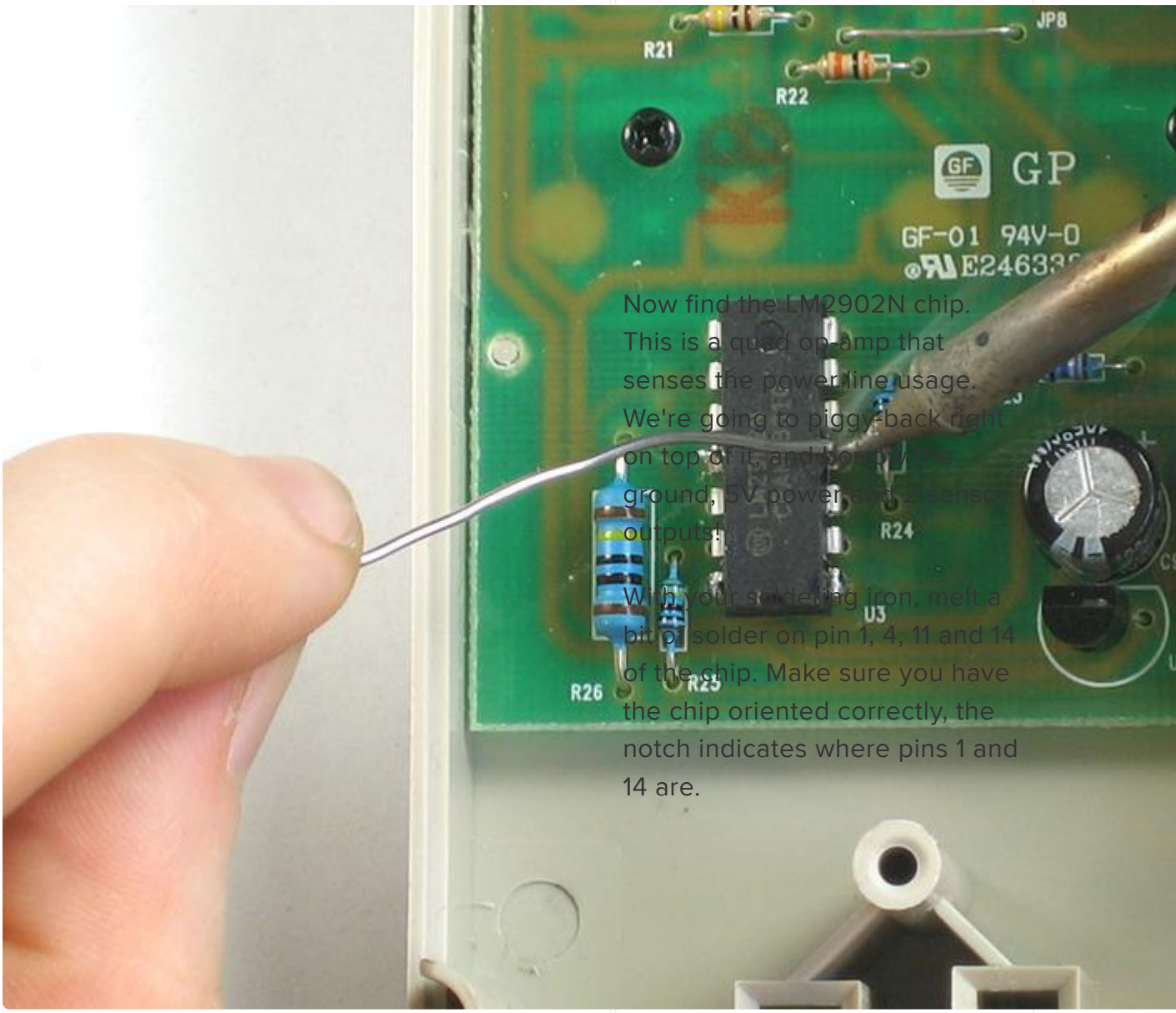
Now it's time to open the Kill-a-Watt! There are only 3 screws that hold it together, and they are found on the back.

Be gentle when opening as there's a delicate ribbon cable inside. Carefully pry off the back cover.

MODEL:P4400 KILL A WATT™
Max.Voltage:125VAC 50/60Hz
Max.Current:15A
Max.Power:1875VA
Serial NO.:SIC3686
MADE IN CHINA
www.p3international.com
3039508
CONFORMS TO UL STD.UL 610108-1
CERTIFIED TO CSA STD.C22.2 NO 1010.1
GP
94V-0
E246338



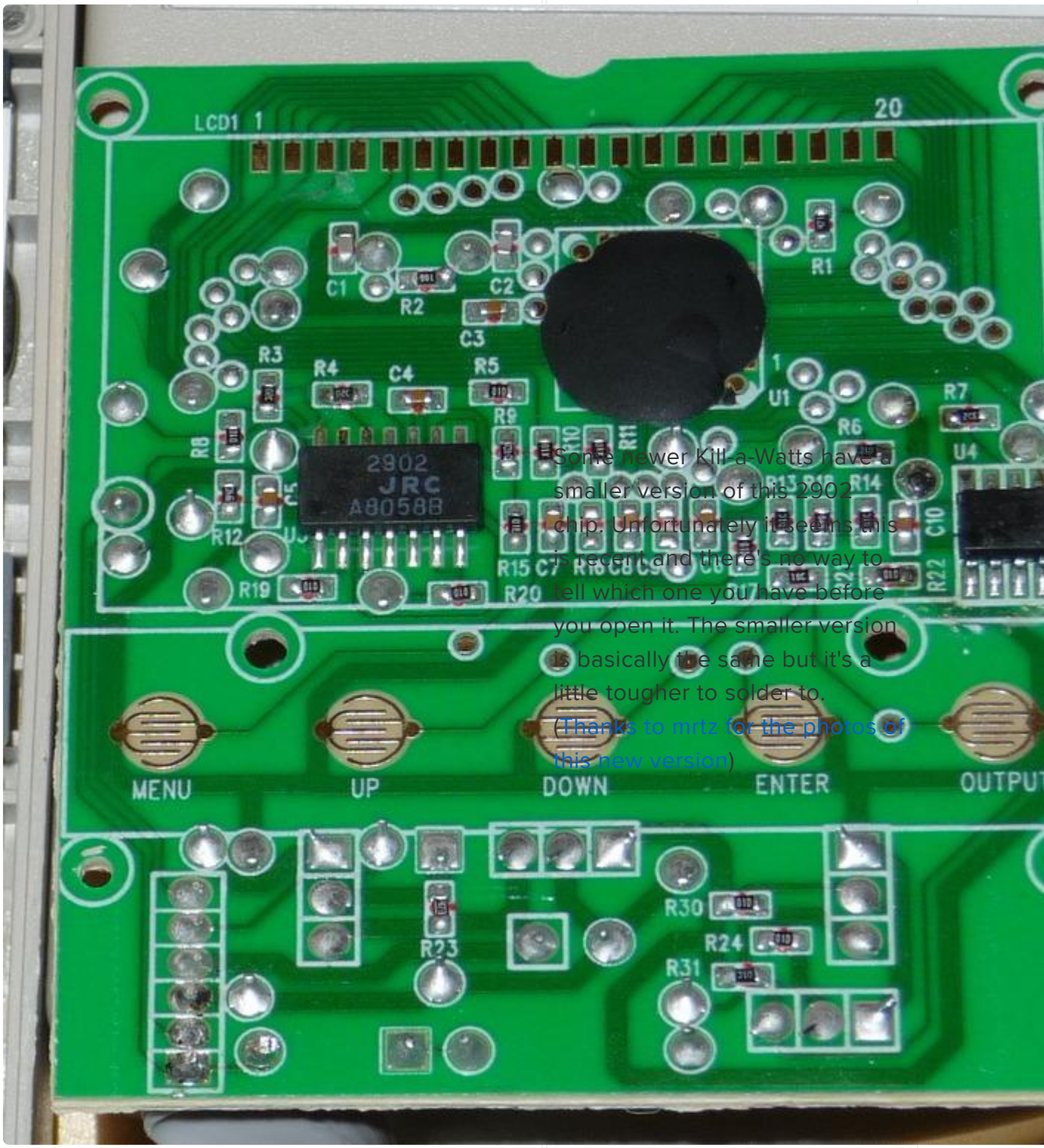
Use a 3/8 drill bit to make a hole near the right corner of the case back. This is what the LED will stick out of. (Ignore the white tape and #4, this is a recycled kill-a-watt :)



chipsolder.jpg

Now find the LM2902N chip. This is a quad op-amp that senses the power line usage. We're going to piggy-back right on top of it, and solder to the ground, 5V power and 2 sensor outputs.

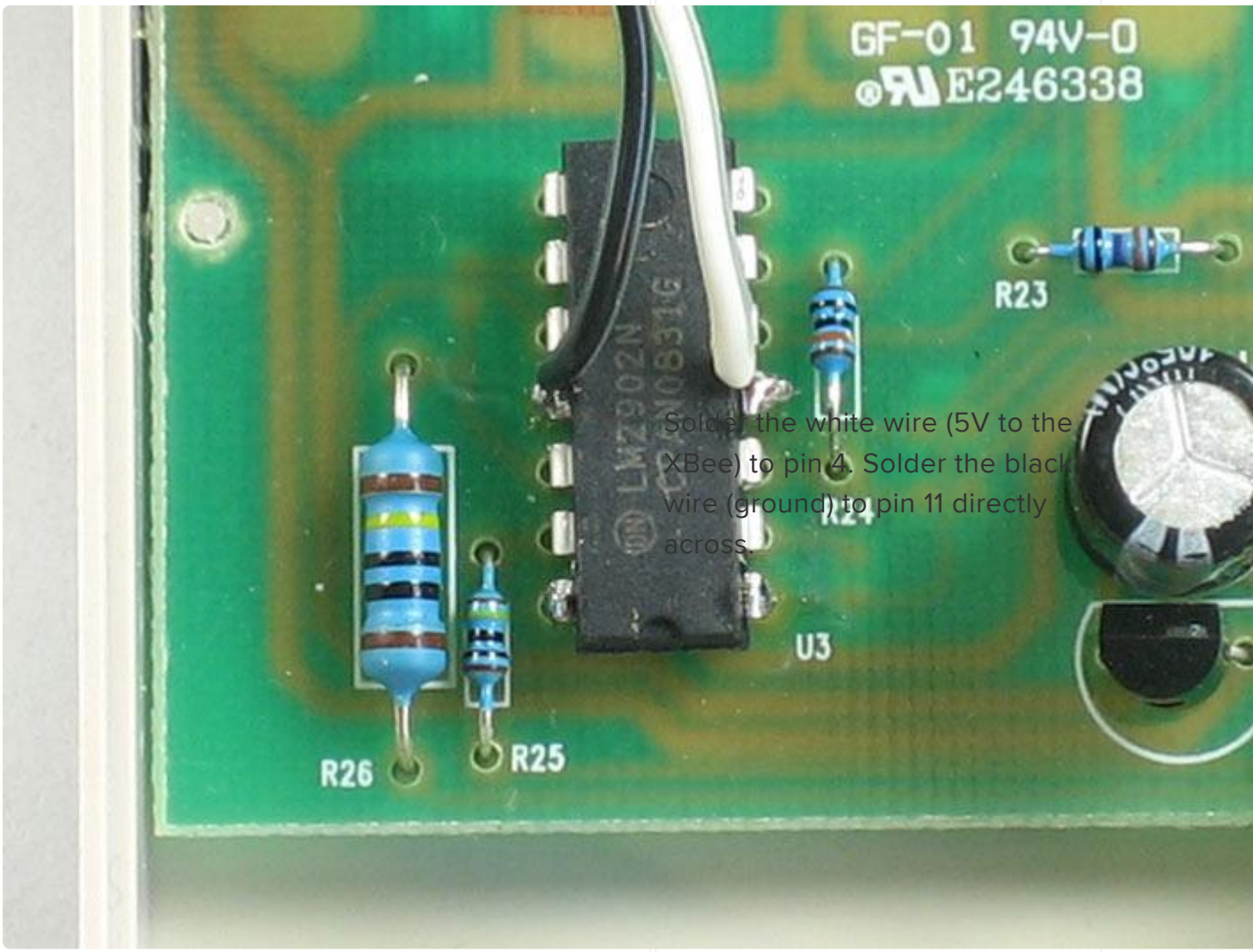
With your soldering iron, melt a bit of solder on pin 1, 4, 11 and 14 of the chip. Make sure you have the chip oriented correctly, the notch indicates where pins 1 and 14 are.



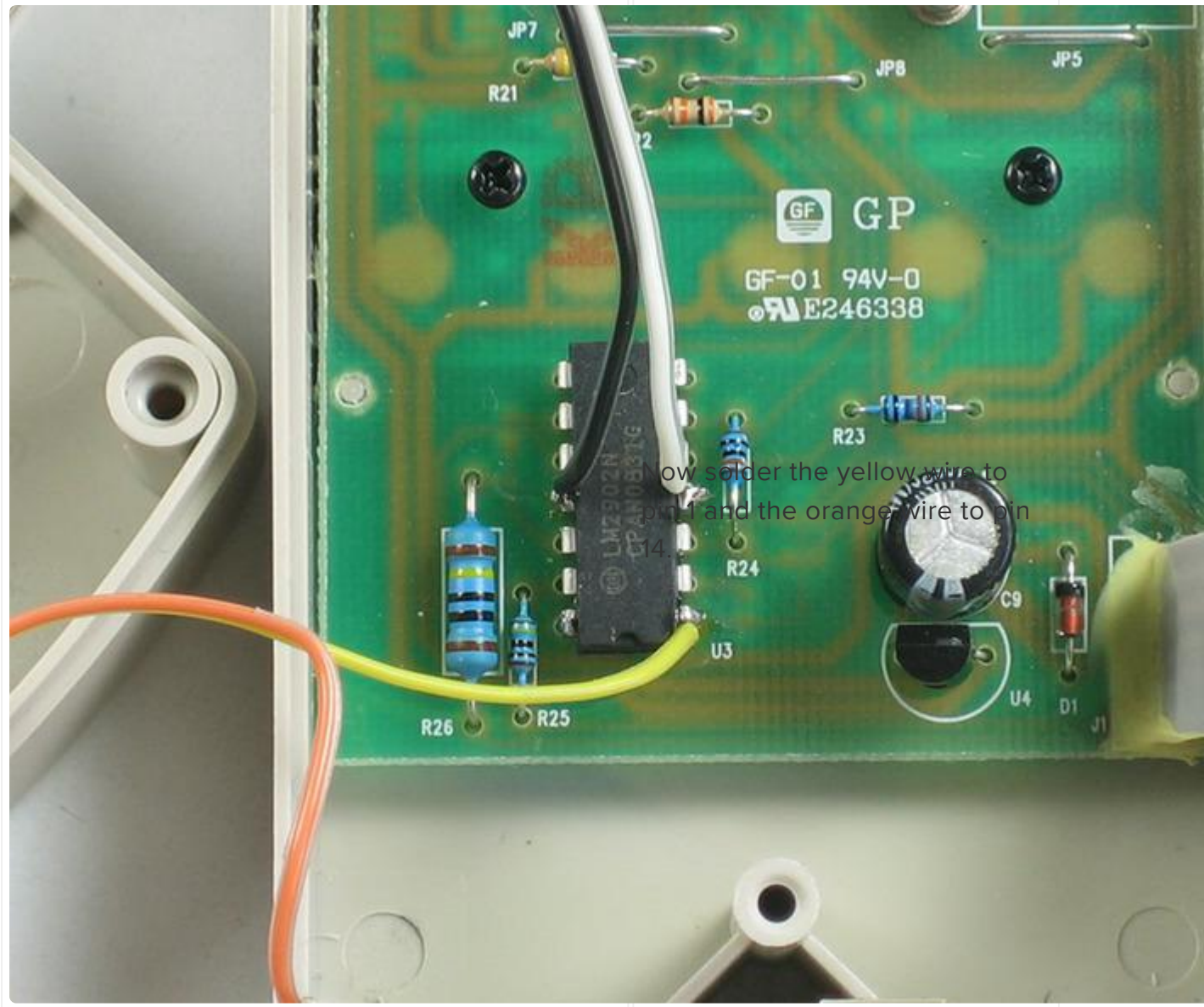
newkaw.jpg

Some newer Kill-a-Watts have a smaller version of this 2902 chip. Unfortunately it seems this is recent and there's no way to tell which one you have before you open it. The smaller version is basically the same but it's a little tougher to solder to.

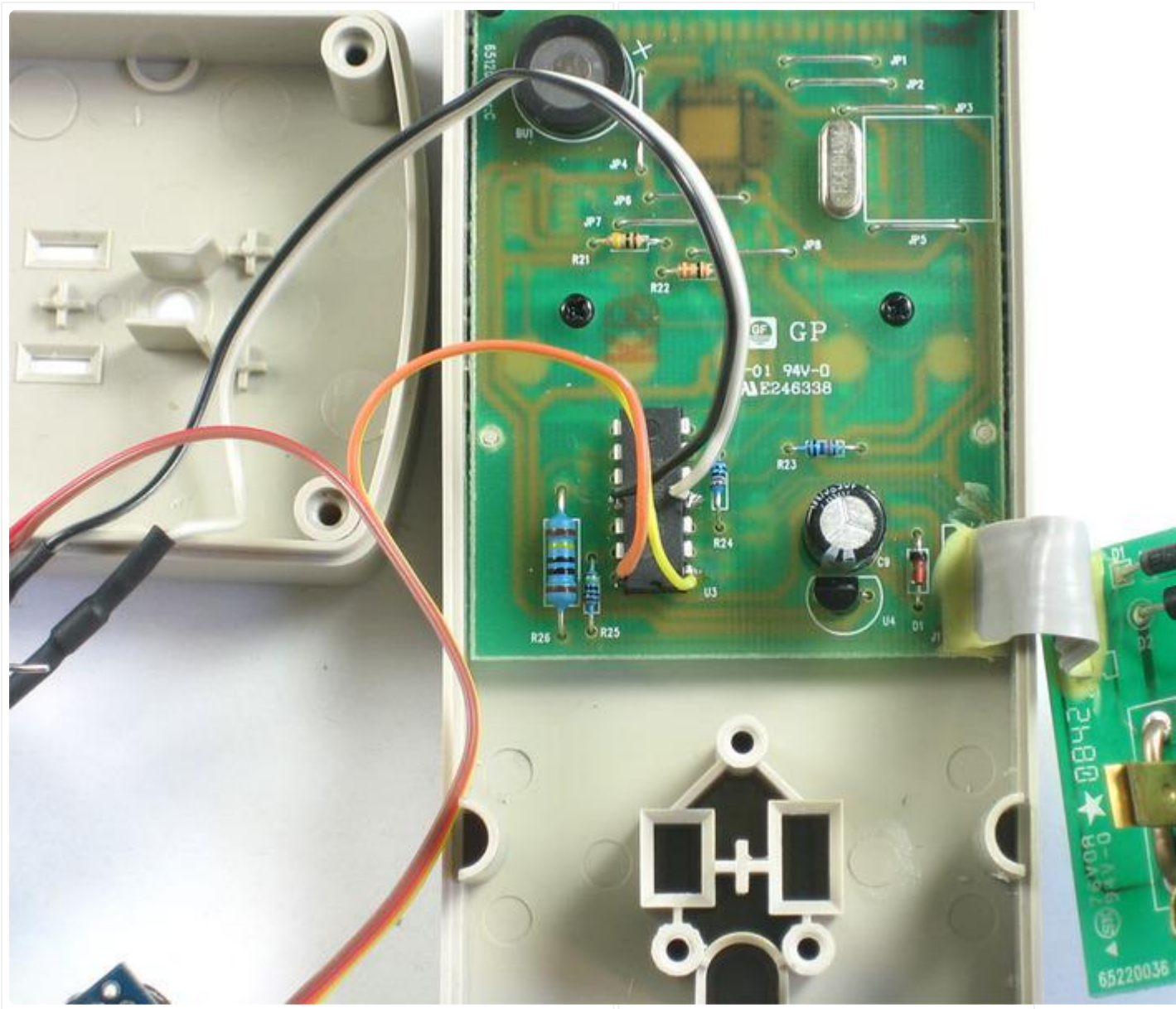
(Thanks to mrtz for the photos of this new version)

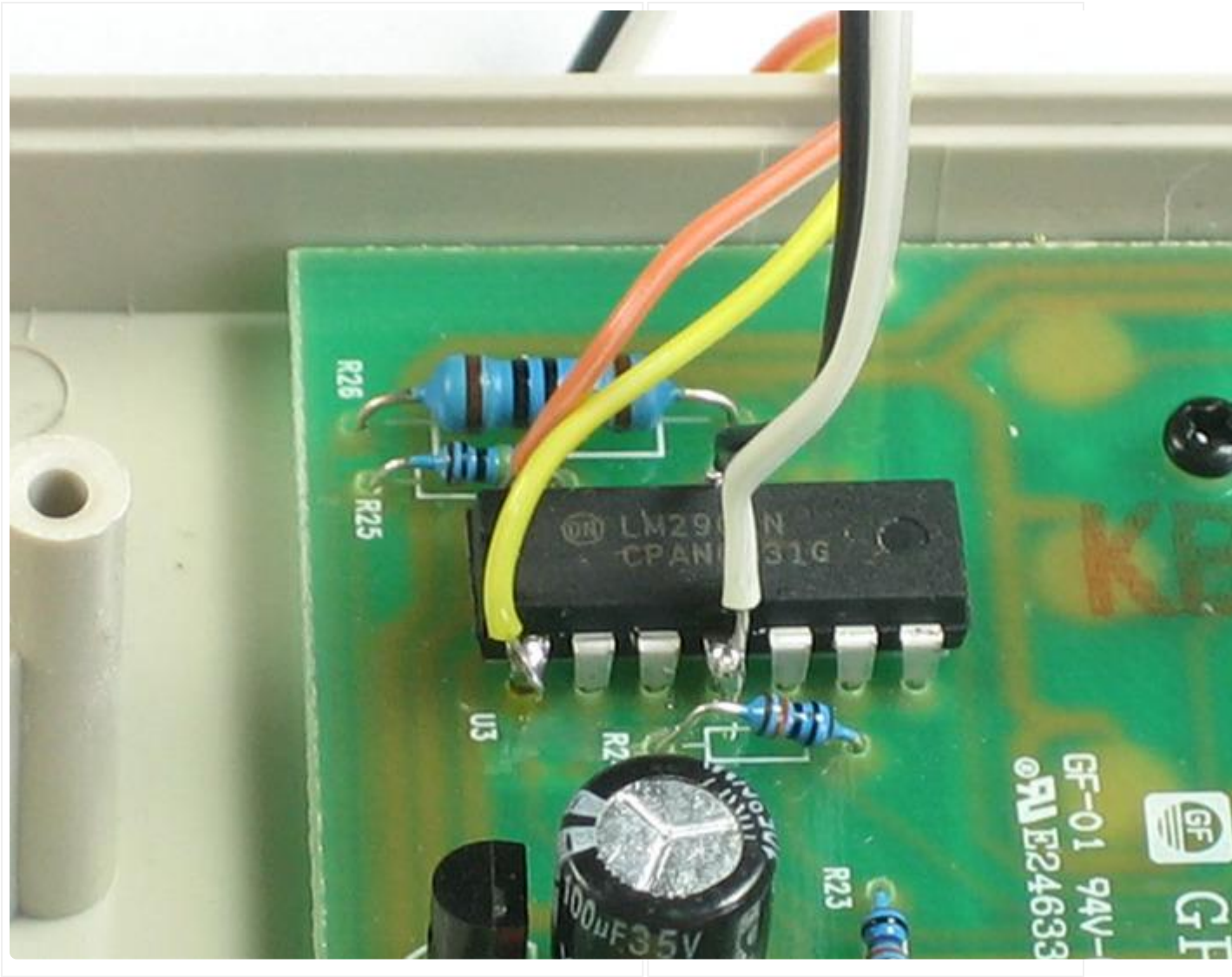


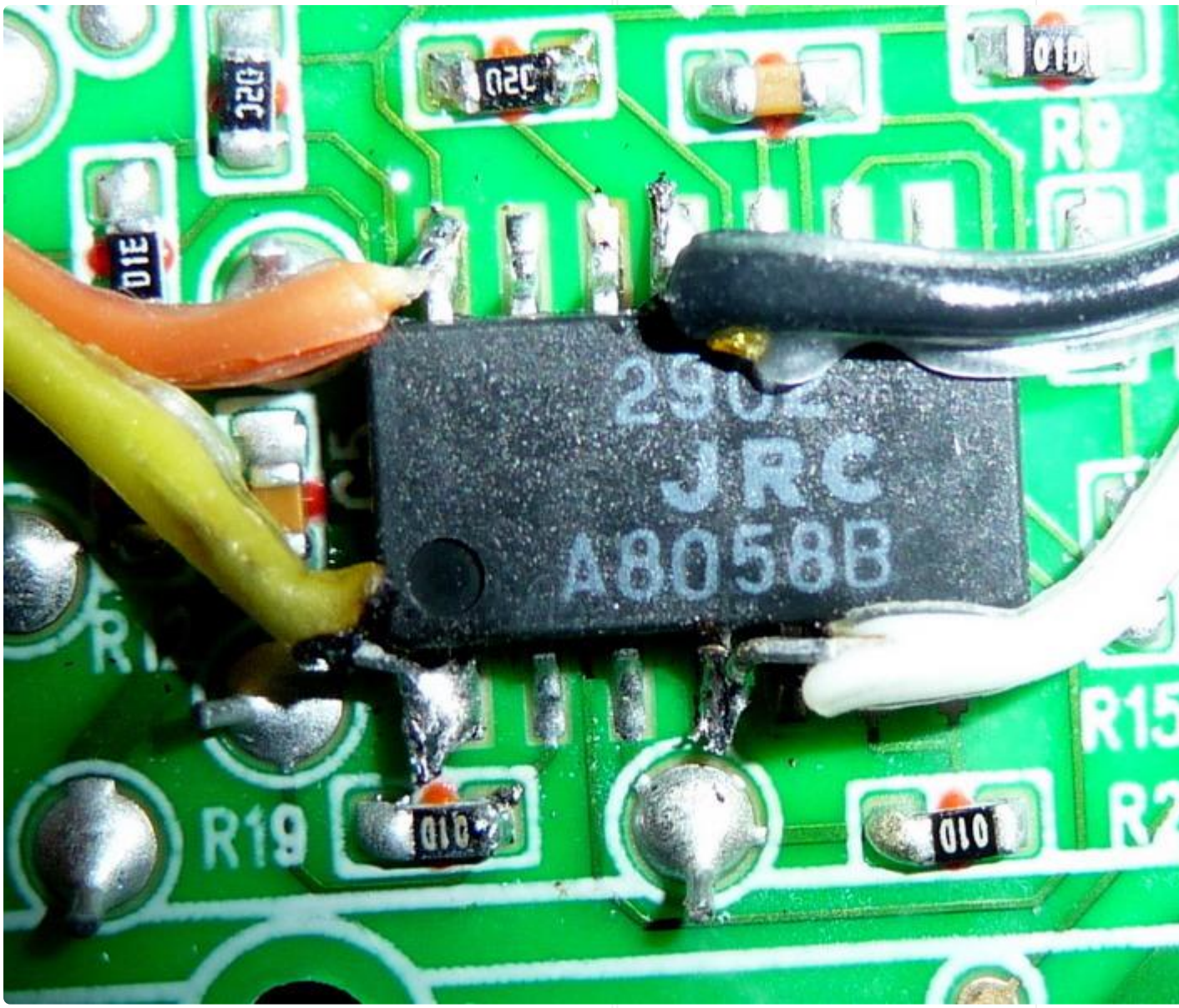
Solder the white wire (5V to the XBee) to pin 4. Solder the black wire (ground) to pin 11 directly across.



Now solder the yellow wire to pin 1 and the orange wire to pin 14.

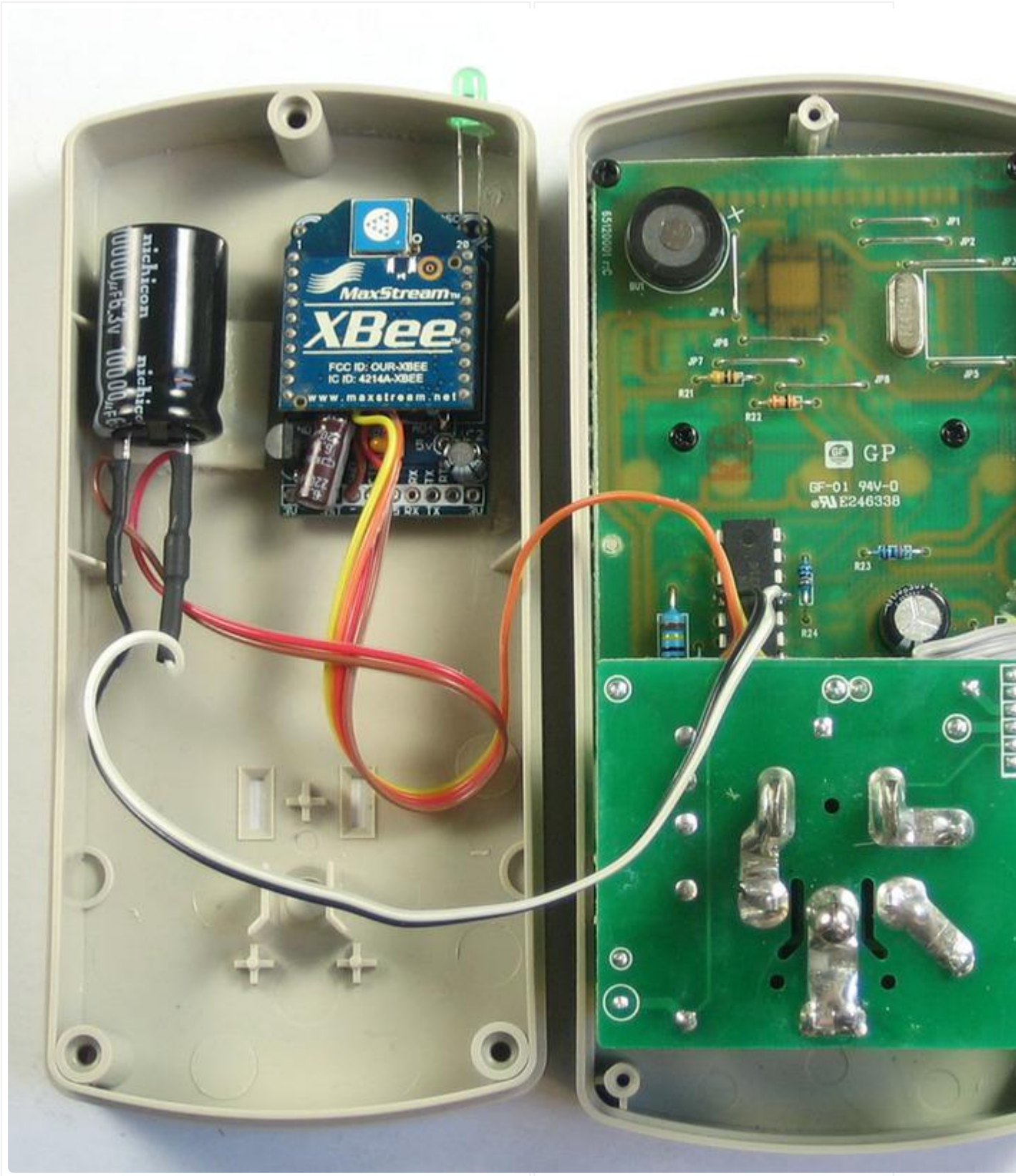

JP1
JP2
JP3
JP4
JP6
JP7
JP8
JP5
R21
R22
GP
-01 94V-0
E246338
R23
R24
C9
U3
U4
D1
R26
R25
65220036


R26
R25
U3
R23
100µF35V
GF-01 94V-
E24633


JRC
A8058B
R19
R15

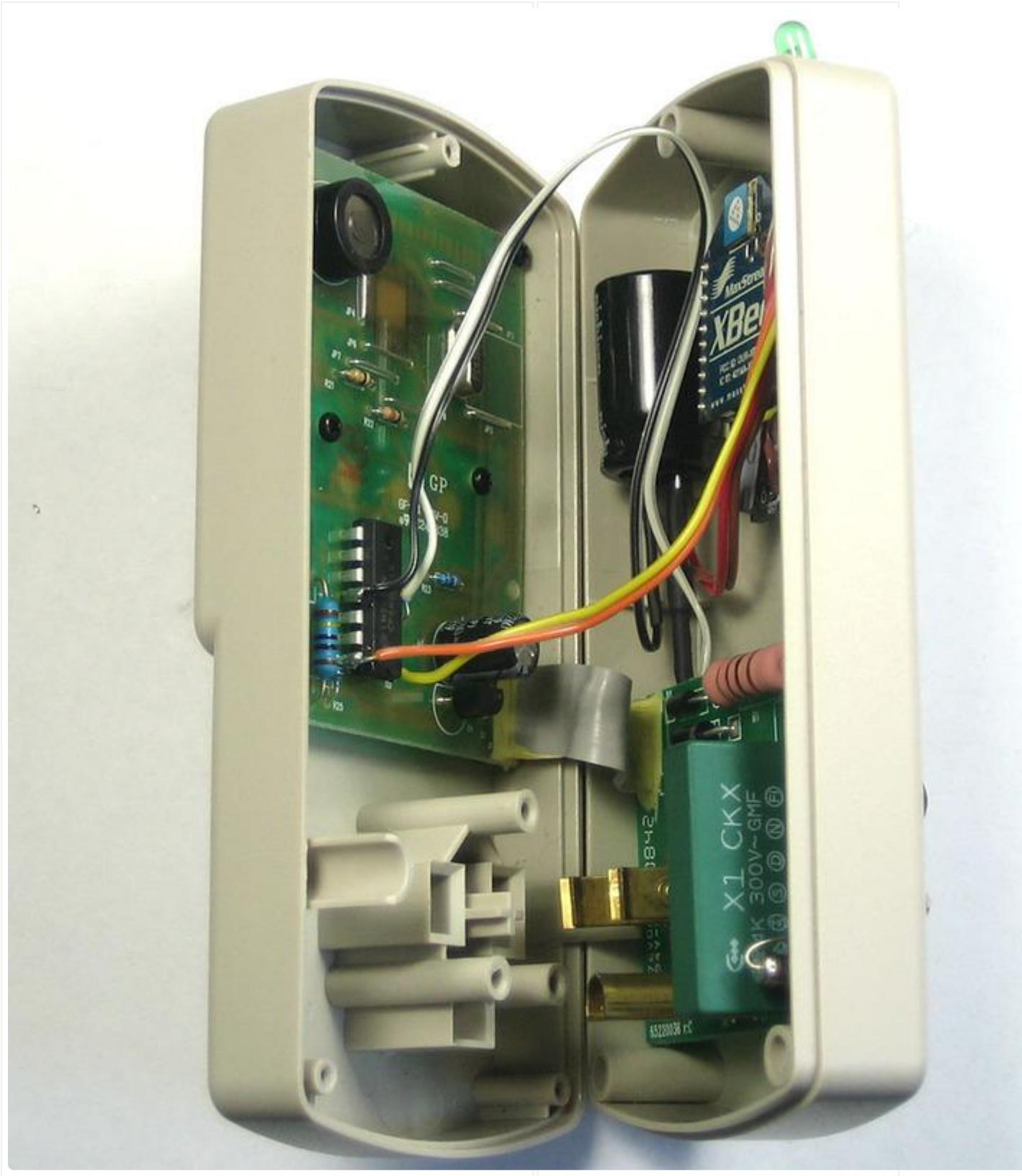
Use two small pieces of sticky foam and stick them on to the back of the case.
Then place the XBee adapter and capacitor on the tape so that the LED sticks out of the hole drilled earlier.

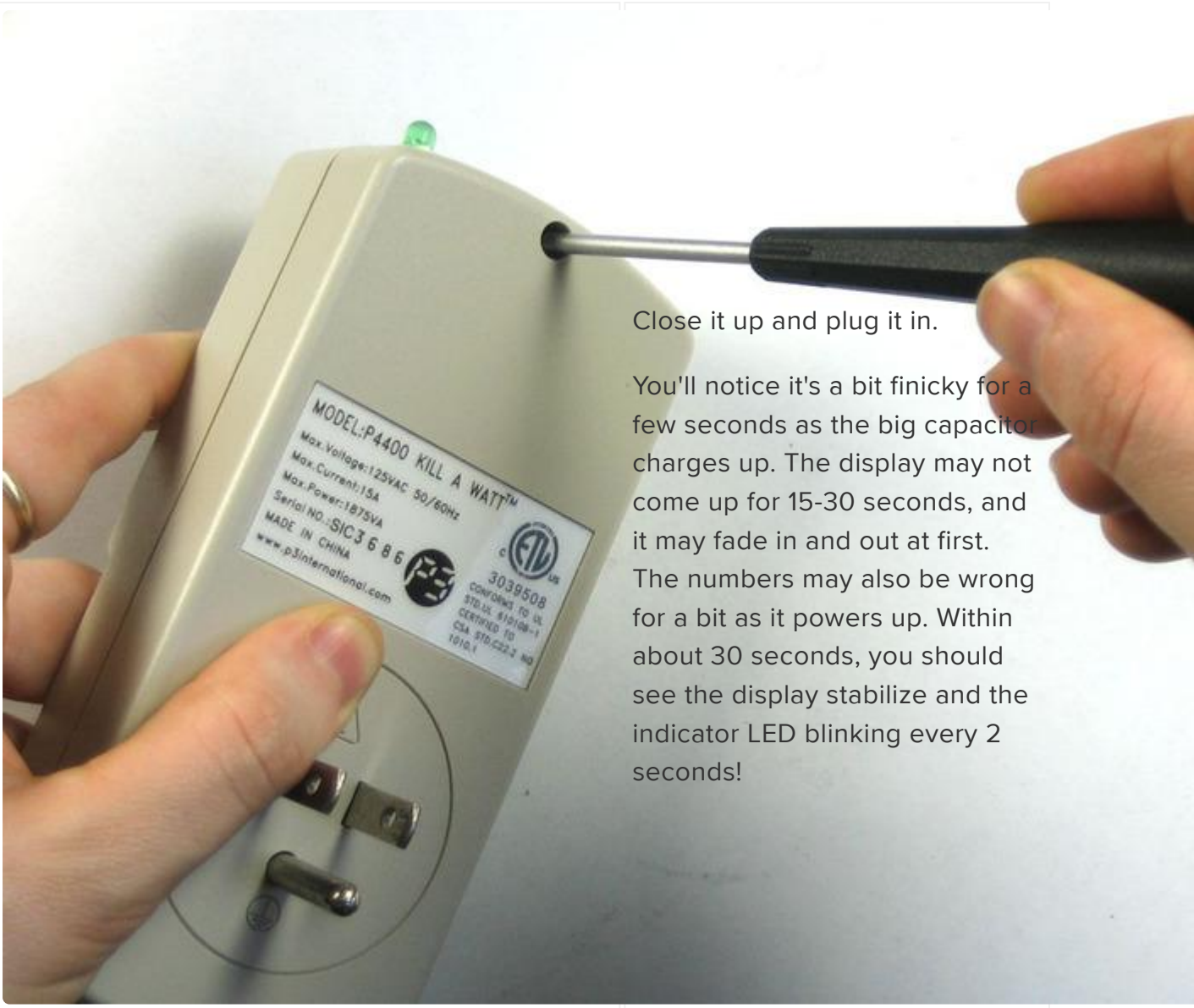
taped.jpg


MaxStream
XBee
FCC ID: OUR-XBEE
IC ID: 4214A-XBEE
www.maxstream.net
nichicon
GP
GF-01 94V-0
E246338

MaxStream
XBee

Tuck the excess ribbon cable out of the way so that it's not near the 120V connections which could make them go poof.


X1 CKX
300V~GMF

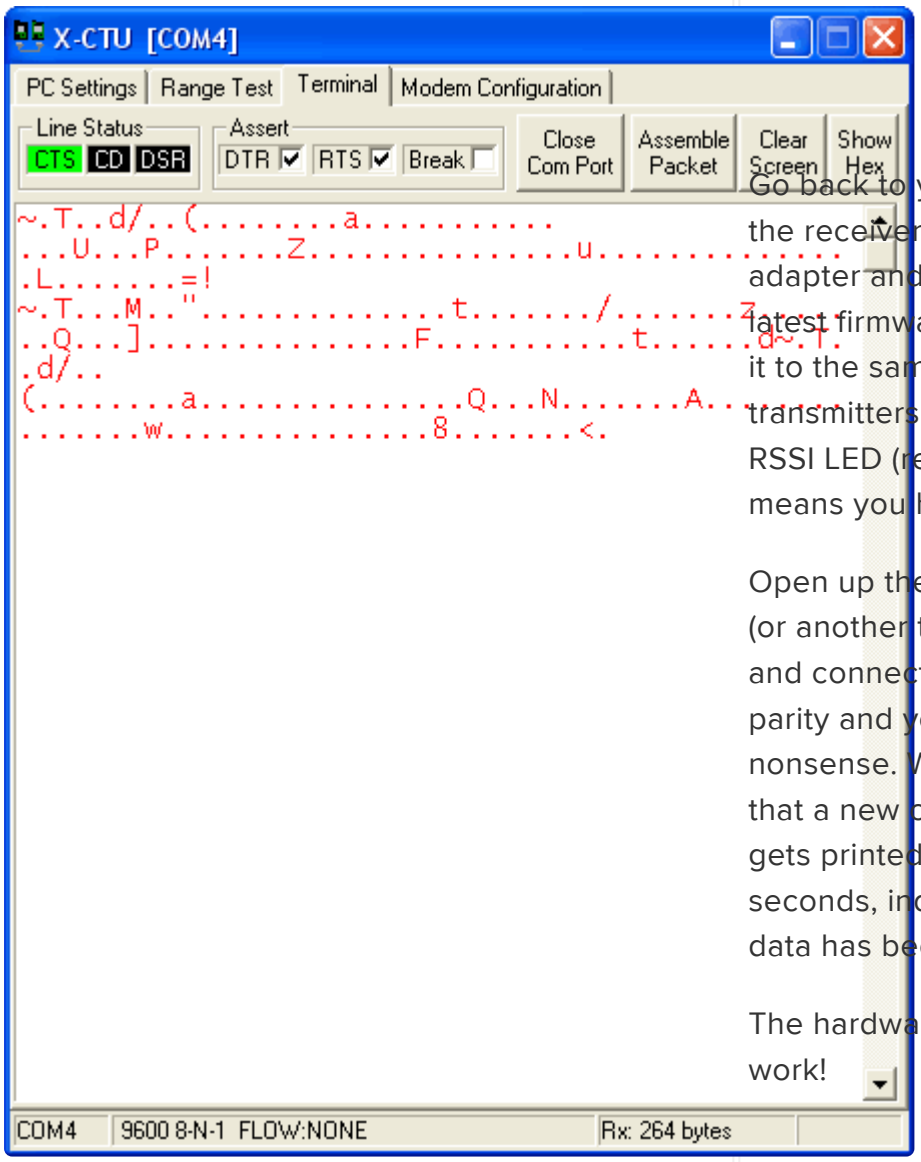


reassemble.jpg

Close it up and plug it in.

You'll notice it's a bit finicky for a few seconds as the big capacitor charges up. The display may not come up for 15-30 seconds, and it may fade in and out at first. The numbers may also be wrong for a bit as it powers up. Within about 30 seconds, you should see the display stabilize and the indicator LED blinking every 2 seconds!

KILL A WATT™
METER
123.7
Volt
Volt
Amp
Watt
VA
Hz
PF
KWH
Hour



terminaljunk.gif

Go back to your computer, plug the receiver XBee into the USB adapter and make sure it has the latest firmware uploaded and set it to the same PAN ID as the transmitters. You will see the RSSI LED (red LED) light up. That means you have a good link!

Open up the Terminal in X-CTU (or another terminal program) and connect at 9600 baud 8N1 parity and you'll see a lot of nonsense. What's important is that a new chunk of nonsense gets printed out once every 2 seconds, indicating a packet of data has been received.

The hardware is done. Good work!

Software

Introduction

Now that the hardware is complete, we come to the exciting part: running the software that retrieves the data from our receiver XBee and saves it to our computer or uploads it to a database or updates our twitter feed or....whatever you'd like!

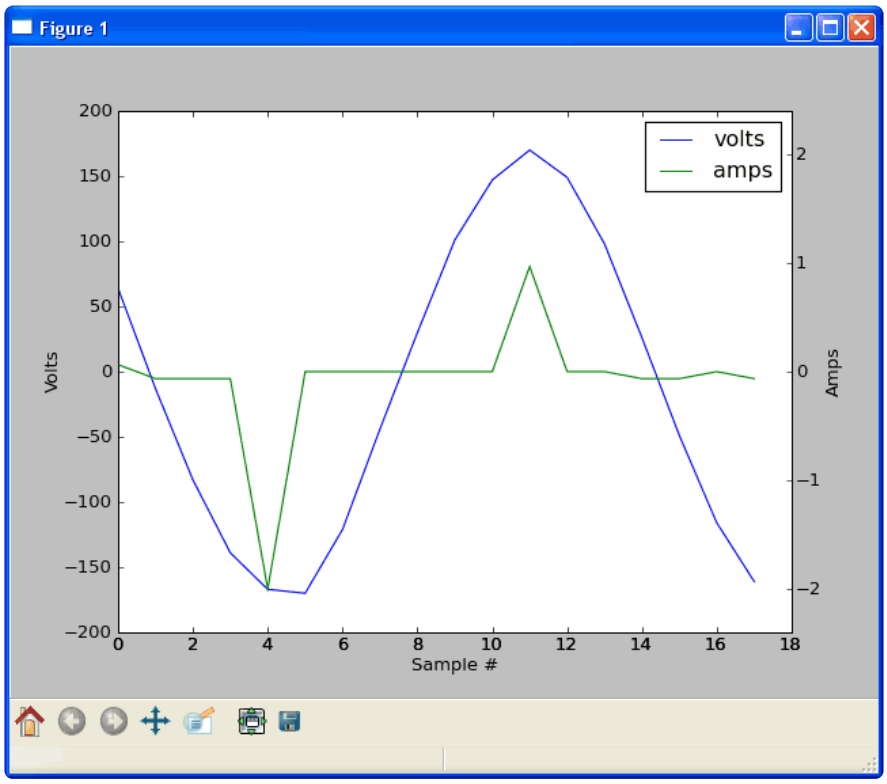
Here is how it works, the XBee inside the Kill-a-Watt is hooked up to two analog signals. One is the voltage signal which indicates the AC voltage read. In general this is a sine wave that is 120VAC. One tricky thing to remember is that 120V is the 'RMS' voltage, and the 'true voltage' is +-170VDC. (You can read more about RMS voltage at wikipedia () basically it's a way to indicate how much 'average' voltage there is.) The

second reading is the AC current read. This is how much current is being drawn through the Kill-a-Watt. If you multiply the current by the voltage, you'll get the power (in Watts) used!

The XBee's Analog/Digital converter is set up to take a 'snapshot' of one sine-cycle at a time. Each double-sample (voltage and current) is taken 1ms apart and it takes 17 of them. That translates to a 17ms long train of samples. One cycle of power-usage is 1/60Hz long which is 16.6ms. So it works pretty well!

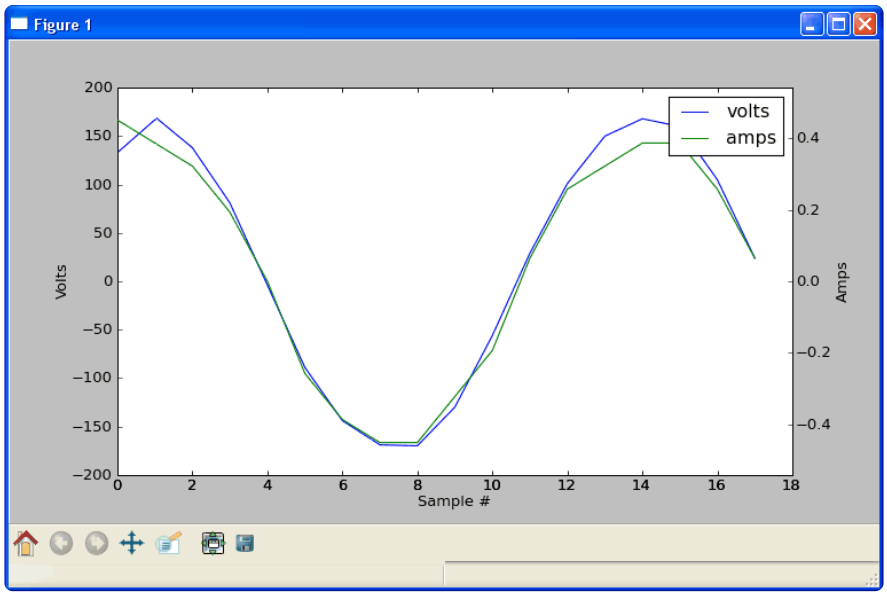
Lets look at some examples of voltage and current waveforms as the XBee sees them.

For example this first graph is of a laptop plugged in. You'll see that it's a switching supply, and only pulls power during the peak of the voltage curve.



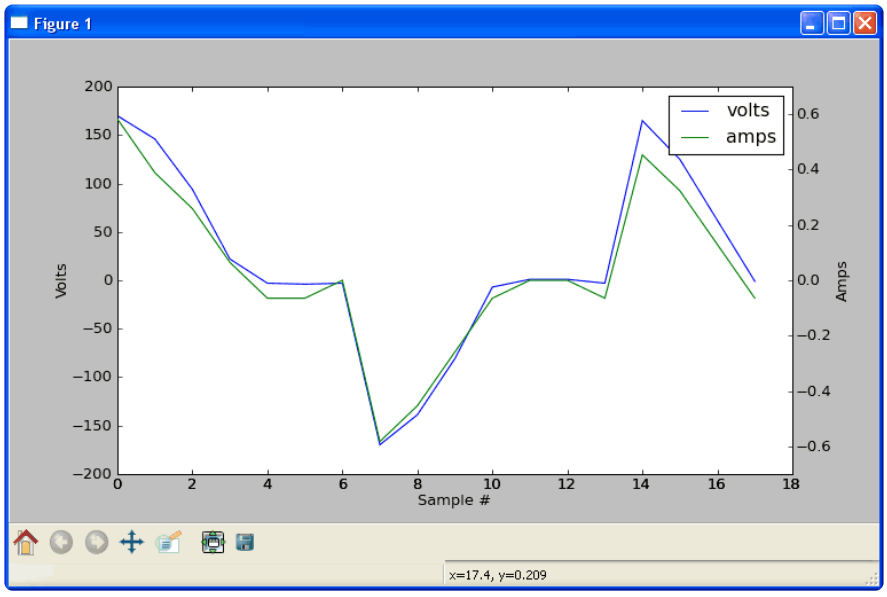
A laptop plugged in, switching power supply

Now let's try plugging in a 40W incandescent light bulb. You'll notice that unlike the switching supply, the current follows the voltage almost perfectly. That's because a lightbulb is just a resistor!



40W lightbulb

Finally, let's try sticking the meter on a dimmable switch. You'll see that the voltage is 'chopped' up, no longer sinusoidal. And although the current follows the voltage, it's still matching pretty well.



Light bulb on dimmer switch

The XBee sends the raw data to the computer which, in a python script, figures out what the (calibrated) voltage and amperage is at each sample and multiplies each point together to get the Watts used in that cycle. Since there's almost no device that changes the power-usages from cycle-to-cycle, the snapshot is a good indicator of the overall power usage that second. Then once every 2 seconds, a single snapshot is sent to the receiver XBee

Install python & friends

The software that talks to the XBee is written in python. I used python because it's quick to develop in, has multi-OS support and is pretty popular with software and hardware hackers. The XBees talk over the serial port so literally any programming language can/could be used here. If you're a software geek and want to use perl, C, C#, tcl/tk, processing, java, etc. go for it! You'll have to read the serial data and parse out the packet but it's not particularly hard.

However, most people just want to get on with it and so for you we'll go through the process of installing python and the libraries we need.

1. Download and install python 2.5 from http://www.python.org/download/ () I suggest 2.5 because that seems to be stable and well supported at this time. If you use another version there may be issues.
2. Download and install pyserial from the package repository () (this will let us talk to the XBee thru the serial port).
3. If you're running windows download and install win32file for python 2.5 () (this will add file support).
4. Download and install the simplejson python library () (this is how the twitter api likes to be spoken to) you'll need to uncompress the tar.gz file and then run the command "python setup.py install" to install.

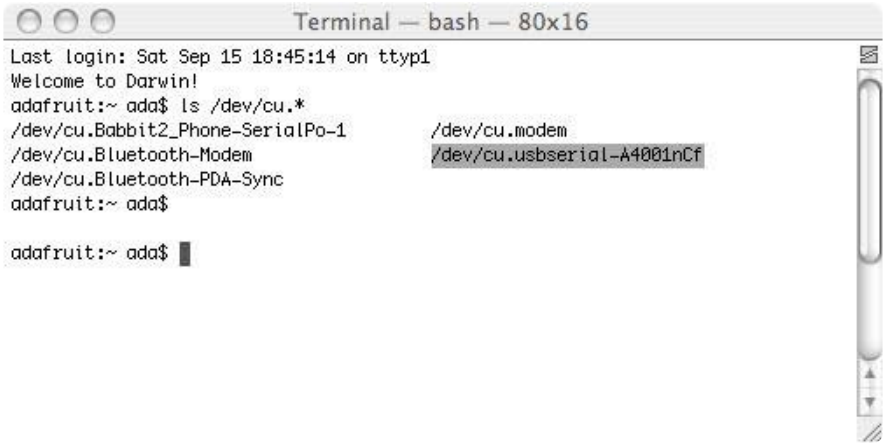
Now you can finally download the Wattcher script () we will demonstrate here! We're going to download it into the C:\wattcher directory, for other OS's you can of course change this directory.

Basic configure

We'll have to do a little bit of setup to start, open up the wattcher.py script with a text editor and find the line:

```
SERIALPORT = "COM4" # the com/serial port the XBee is connected to.
```

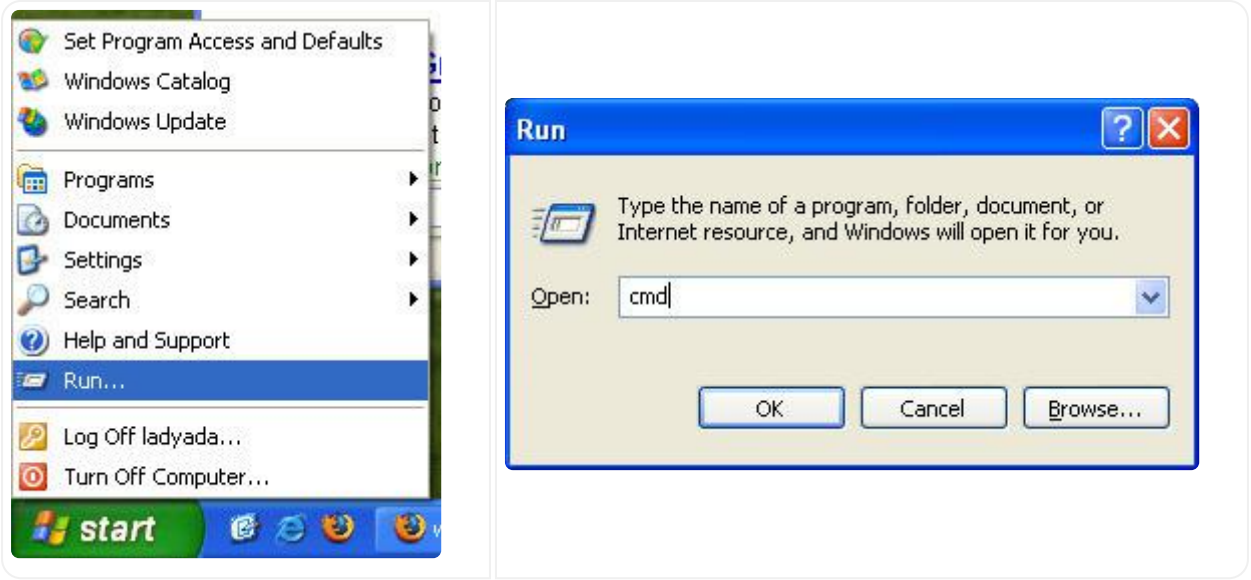
Change COM4 into whatever the serial port you will be connecting to the XBee with is called. Under windows it's some COMx port, under linux and mac it's something like /dev/cu.usbserial-xxxx check the /dev/ directory and/or dmesg.



Save the script with the new serial port name.

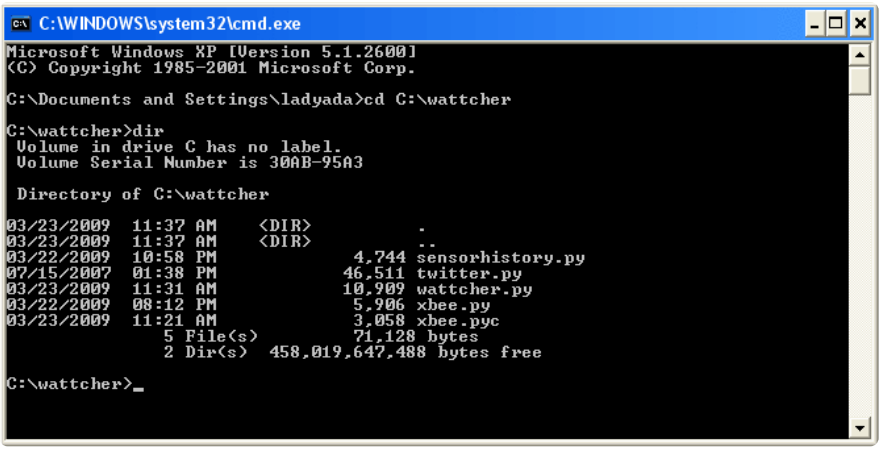
Test it out

Once you have installed python and extracted the scripts to your working directory, start up a terminal (under linux this is just rxvt or xterm, under mac it's Terminal, under windows, it's a cmd window).



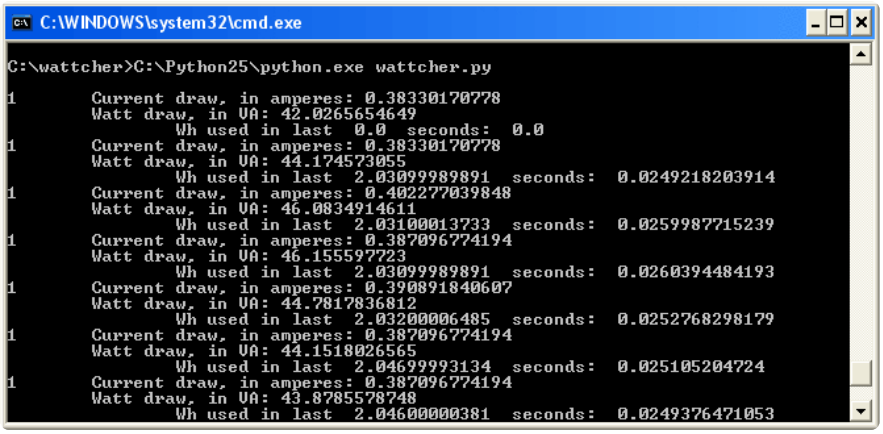
I'm going to assume you're running windows from now on, it shouldn't be tough to adapt the instructions to linux/mac once the terminal window is open.

Run the command cd C:\wattcher to get to the place where you uncompressed the files. By running the dir command you can see that you have the files in the directory.



Make sure your transmitter (Kill-a-Watt + Xbee) is plugged in, and blinking once every 2 seconds. Remember it takes a while for the transmitter to charge up power and start transmitting. The LCD display should be clear, not fuzzy. Make sure that there's nothing plugged into the Kill-a-Watt, too. The RSSI (red) LED on the receiver connected to the computer should be lit indicating data is being received. Don't continue until that is all good to go!

Now run python by running the command C:\python25\python.exe wattcher.py

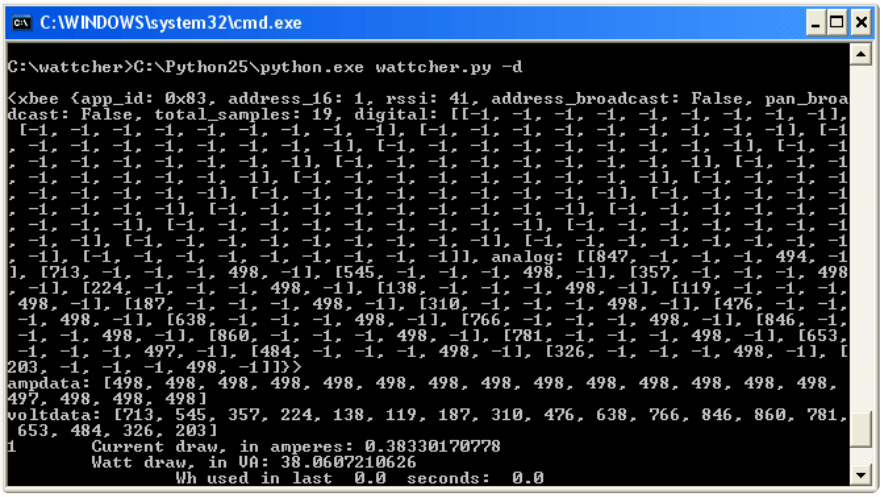


You should get a steady print out of data. The first number is the XBee address from which it received data, following is the estimated current draw, wattage used and the Watt-hours consumed since the last data came in. Hooray! We have wireless data!

Calibrating

Now that we have good data being received, it's time to tweak it. For example, it's very likely that even without an appliance or light plugged into the Kill-a-Watt, the script thinks that there is power being used. We need to calibrate the sensor so that we know where 'zero' is. In the Kill-a-Watt there is an autocalibration system but unfortunately the XBee is not smart enough to do it on its own. So, we do it in the python script. Quit the script by typing in Control-C and run it again this time as C:\python25\python.exe wattcher.py -d note the -d which tells the script to print out

debugging information.

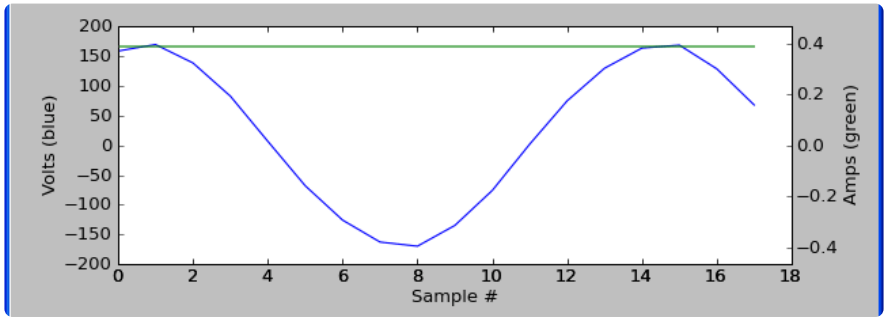


Now you can see the script printing out a whole mess of data. The first chunk with lots of -1's in it is the raw packet. While it's interesting we want to look at the line that starts with ampdata:

> ampdata: [498, 498, 498, 498, 498, 498, 498, 498, 498, 498, 498, 498, 498, 498, 497, 498, 498, 498]

Note: If you're getting -1's instead of nice ~500 numbers, check the XBee is set up right and has the correct wires going to the correct A/D pins. If you're getting 1024's that probably means you forgot to tie VREF to VCC in the last step. Remove the Kill-a-Watt from power and go back to repair it.

Now you'll notice that the numbers are pretty much all the same. That's because there's nothing plugged into the tweetawatt and so each 1/60 Hz cycle has a flat line at 'zero'. The A/D in the XBee is 10 bits, and will return values between 0 and 1023. So, in theory, if the system is perfect the value at 'zero' should be 512. However, there are a bunch of little things that make the system imperfect and so zero is only close to 512. In this case the 'zero' calibration point is really 498. When it's off there is a 'DC offset' to the Amp readings, as this graph shows:



See how the Amp line (green) is steady but it's not at zero, it's at 0.4 amps? There is a 'DC offset' of 0.4 amps.

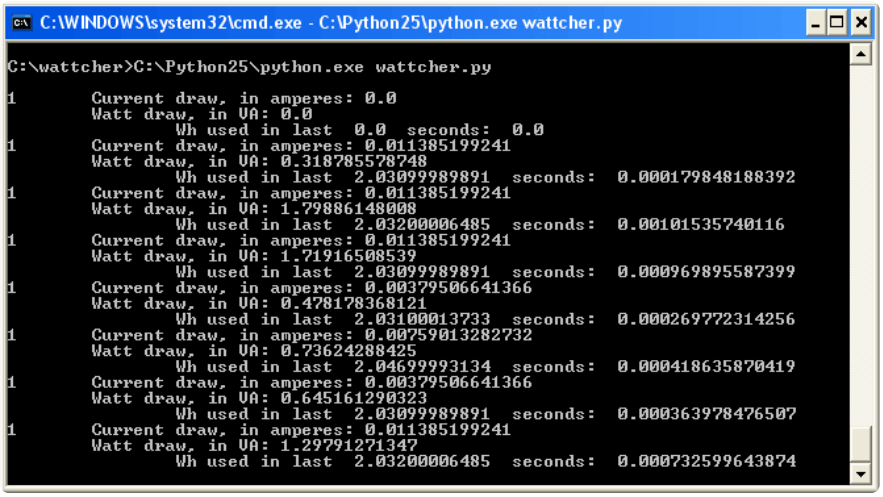
OK, open up the wattcher.py script in a text editor.

```
vrefcalibration = [492,  # Calibration for sensor #0
        492,  # Calibration for sensor #1
        489,  # Calibration for sensor #2
        492,  # Calibration for sensor #3
        501,  # Calibration for sensor #4
        493]  # etc... approx ((2.4v * (10Ko/14.7Ko)) / 3
```

See the line that says # Calibration for sensor #1? Change that to 498.

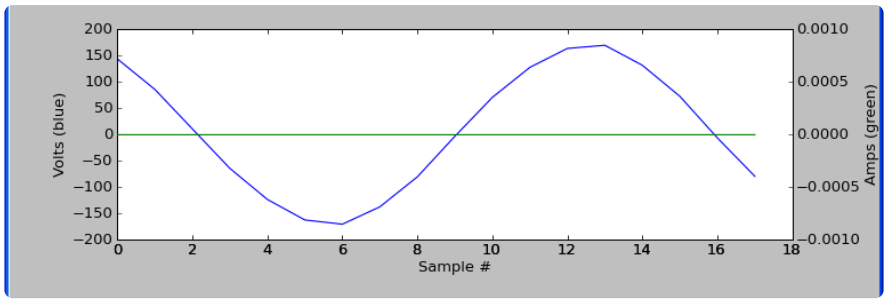
```
vrefcalibration = [492,  # Calibration for sensor #0
        498,  # Calibration for sensor #1
        489,  # Calibration for sensor #2
        492,  # Calibration for sensor #3
        501,  # Calibration for sensor #4
        493]  # etc... approx ((2.4v * (10Ko/14.7Ko)) / 3
```

Save the file and start up the script again, this time without the -d



Now you'll see that the Watt draw is 2W or less, instead of 40W (which was way off!) The reason it's not 0W is that, first off, there's a little noise that we're reading in the A/D lines, secondly there's power draw by the Kill-a-Watt itself and finally, the XBee doesn't have a lot of samples to work with. However <2W is pretty good considering that the full sensing range is 0-1500W.

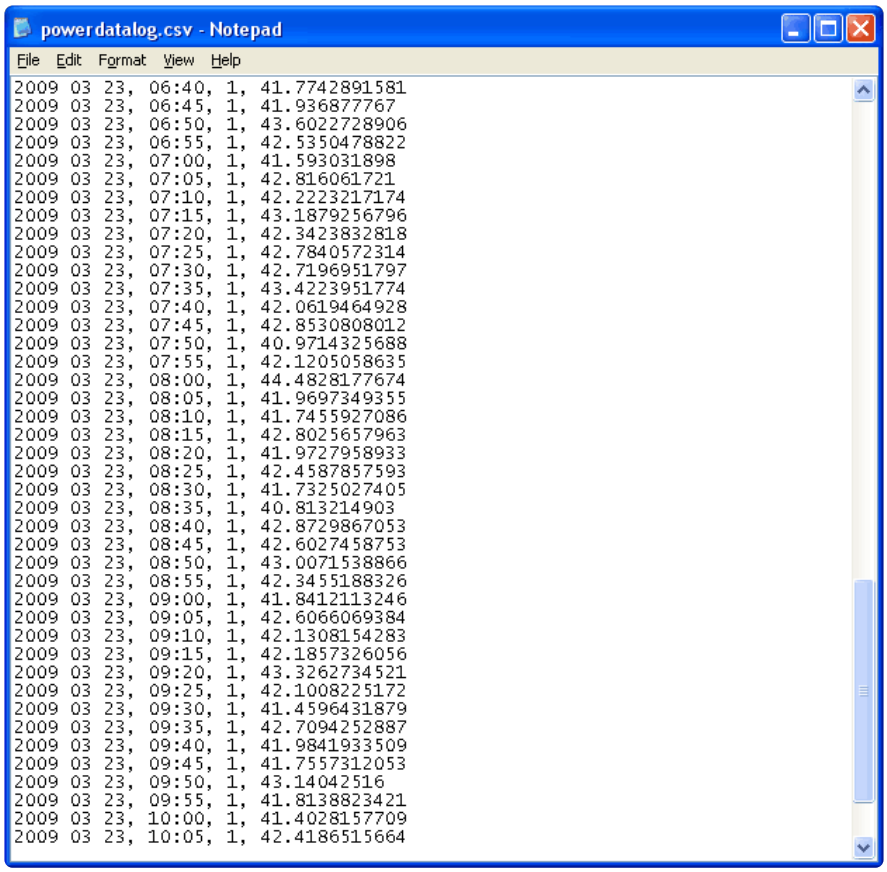
Here is the graph with the calibrated sensor:



See how the Amps line is now at 0 steady, there is no DC offset.

Logging data

It's nice to have this data but it would be even nicer if we could store it for use. Well, that's automatically done for you! You can set the name of the log file in the wattcher.py script. By default it's powerdatalog.csv. The script collects data and every 5 minutes writes a single line in the format Year Month Day, Time, Sensor#, Watts for each sensor. As you can see, this is an example of a 40W incandescent lightbulb plugged in for a few hours. Because of the low sample rate, you'll see some minor variations in the Watts recorded. This data can be easily imported directly into any spreadsheet program.



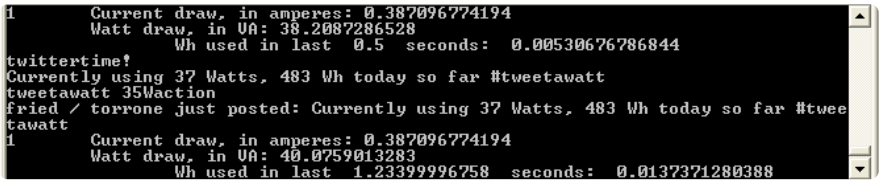
Tweeting

Finally we get to the tweeting part of the tweet-a-watt. First open up the wattcher.py script and set:

```
# Twitter username & password
twitterusername = "username"
twitterpassword = "password"
```

to your username and password on twitter. You can make an account on twitter.com () if you don't have one.

Then run the script as usual. Every 8 hours (midnight, 8am and 4pm) the script will send a tweet using the Twitter API.



Then check it out at your account:

Powermeter

Tweet-a-Watt + Google Powermeter!

Once you have the Tweet-a-Watt working with the python scripts, we can add a cool extension and have beautiful graphs generated thanks to Google Powermeter!

This code, based on the original T-a-W, was created by the super-rad Devlin Thyne ()! You can download it from our source repository on github which now includes this script ().

Please note that this mod is new, and is still under development, if you are having difficulties, try posting to the forums.

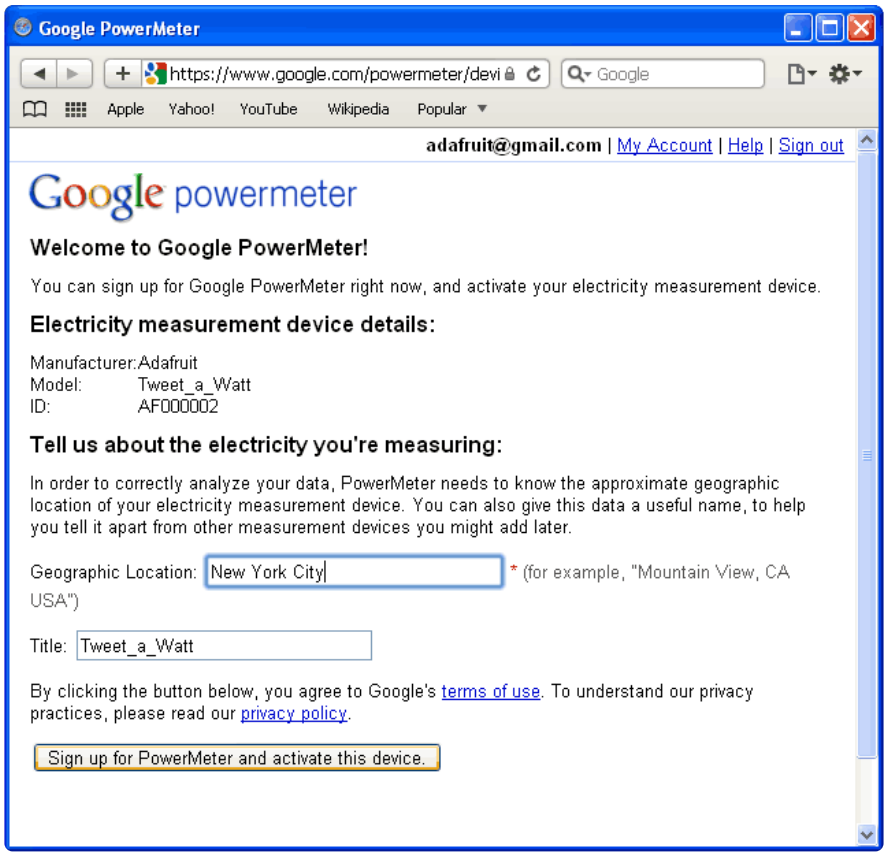
Right now it's only tested to work with a single T-a-W!

Log in + Activate

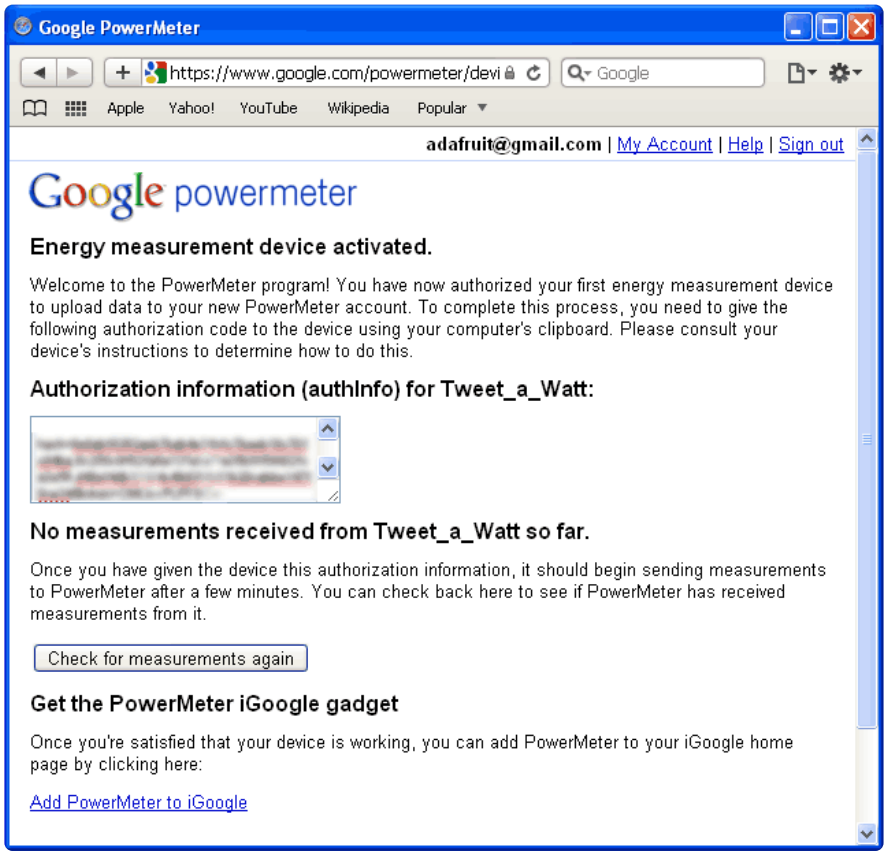
Visit http://www.google.com/powermeter/about/ () and log in with your google account.

Next, go to https://www.google.com/powermeter/device/activate?mfg=Adafruit&model=Tweet_a_Watt&did=AF000002&cvars=0&dvars=1 ()

To activate your Tweet-a-Watt:



Once you activate, you'll get a chunk of text in a box. Copy and paste this into a text file to save it:



Out of this text, there are two things we need, the token and the path. The token is between the text token= and the next &. The path is everything after path=.

hash=0123456789abcdef0123456789abcdef,0123456789abcdef0123456789abcdef,0123456789abcdef0123456789abcdef&token=ABCdefGHI-_JkLMNoP&path=/user/0123456789123456/7891234567819/variable/Adafruit.Tweet_a_Watt.AF000002

So in this case, the token is ABCdefGHI-_JkLMNoP and the path is: /user/0123456789123456/7891234567819/variable/Adafruit.Tweet_a_Watt.AF000002

Save this in a text file, you'll need it for the next step!

Python code

Download the Google Powermeter python API scripts () and place all the files in the same directory as the Wattcher.py scripts.

Now edit gmeter-wattcher.py to have the correct COM/Serial port and calibration data that you figured out in the previous steps. Also, find the two lines:

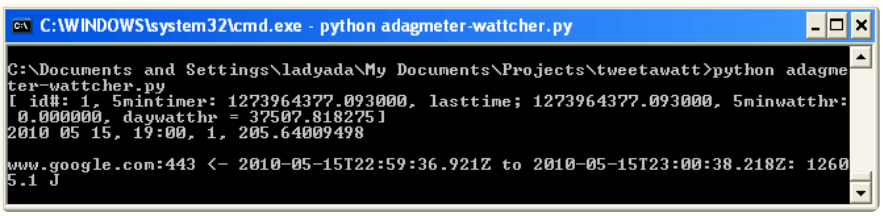
```
#Google PowerMeter stuff:
powerMeterToken = "paste your token info here"
powerMeterPath = "paste your path info here"
```

Edit them so that the token and path you got from google go in between the quotes. If you want lots of data spit out, edit:

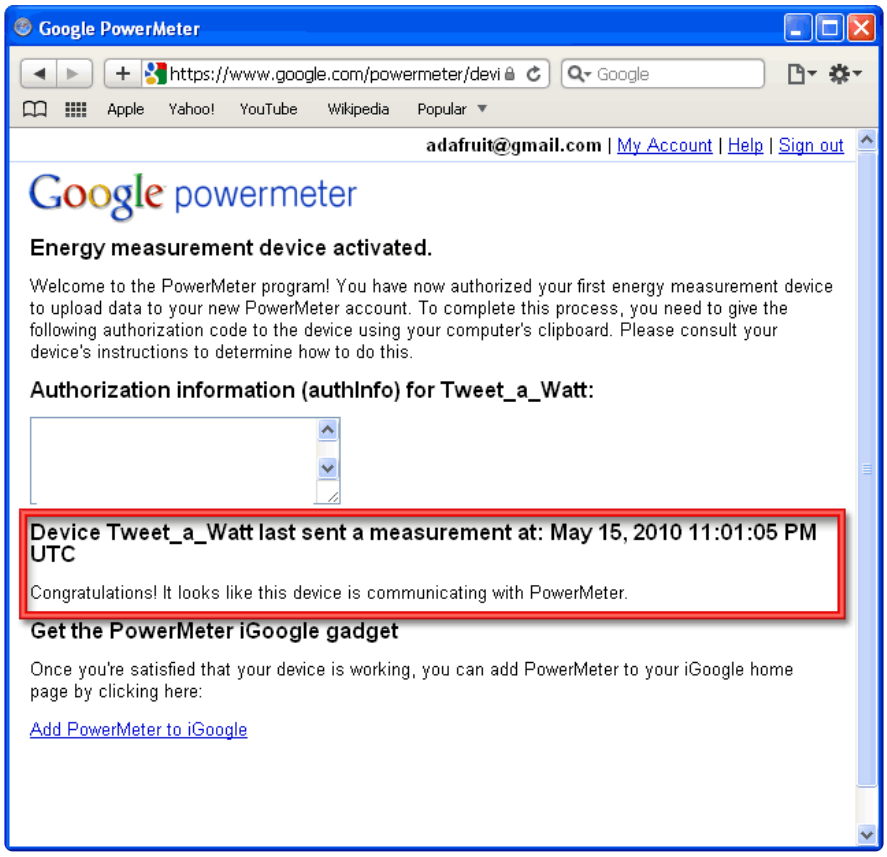
```
QUIET = True
```

To be False

Save gmeter-wattcher.py, Open up a command line as before and go to the wattcher directory and run python gmeter-wattcher.py after a few minutes you shoould see data sent to google:



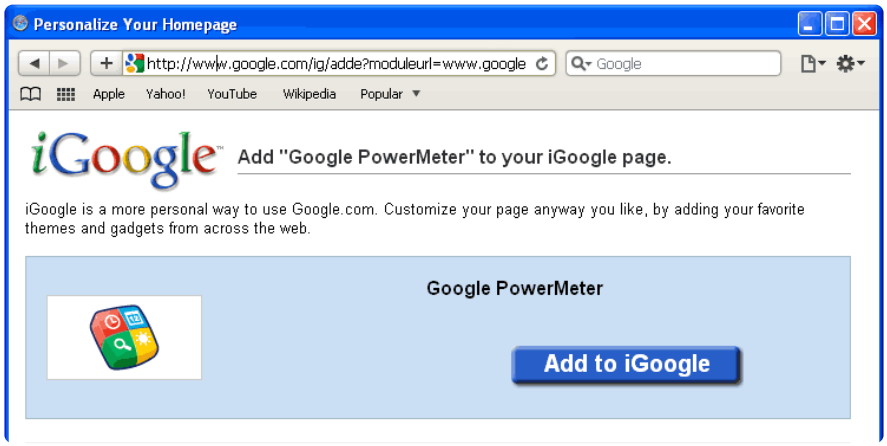
Now go back to your browser and click on "Check for measurements again"



Success!

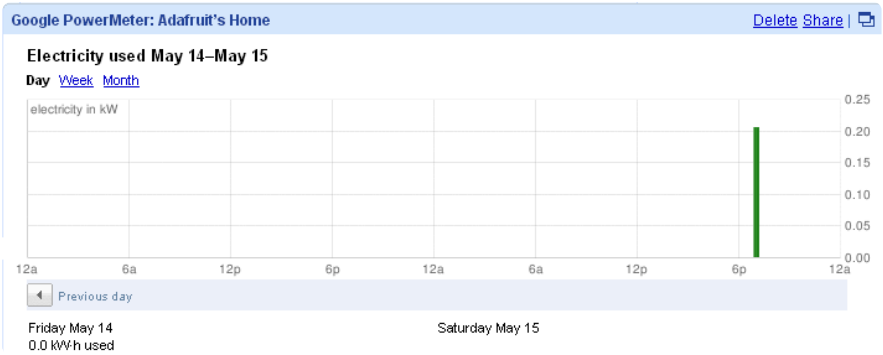
Widgetty

Now we can get the graphs, Click "Add Powermeter to iGoogle"



And then, of course, the big shiny blue button.

Now you gots widgets!



Expand

Improve on your system

Once you've got your base system up and running here are some ideas for how to extend, improve or expand it!

Add more outlets

So you can track more rooms, of course.

Graphing

If you'd like to play some more with the script, there's some extras built in. For example, you can graph the data as it comes in from the XBee, both Watts used and the actual 'power line' waveform. Simply set GRAPHIT = True you'll need to install a mess of python libraries though, including wxpython, numpy and pylab

Remove the computer

It took a few hours, but I hacked my Asus wifi router to also log data for me. There'll be more documentation soon but here's some hints:

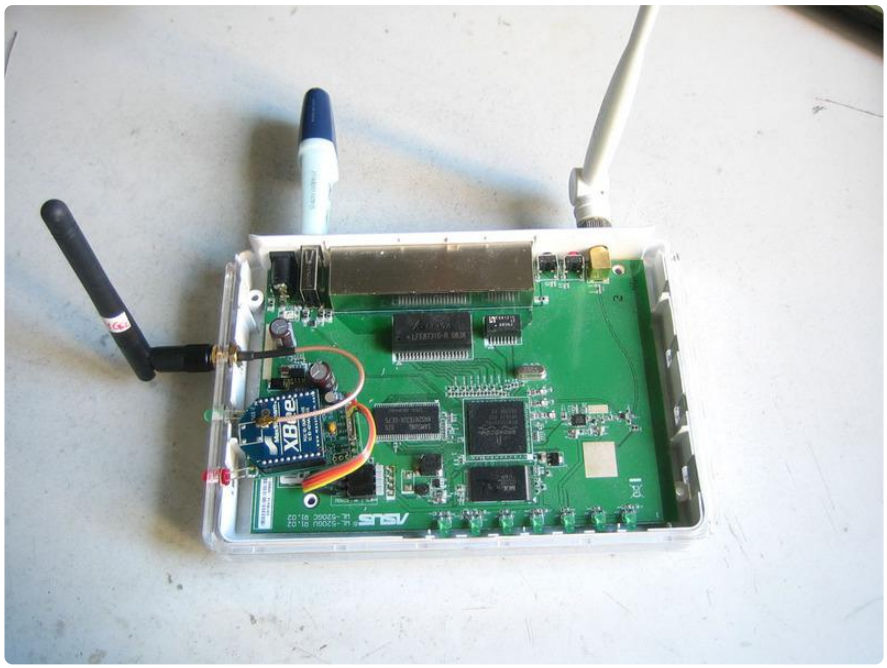
Do basically everything in MightyOhm's tutorial (). You can use the FTDI cable to reprogram the router, just move the pins around. Then add a 16mb USB key (I was given one as schwag so look in your drawers) and install python and the openssl library as well as the other libraries needed like pyserial. The code should pretty much just run! (I'll put up more detailed notes later.)

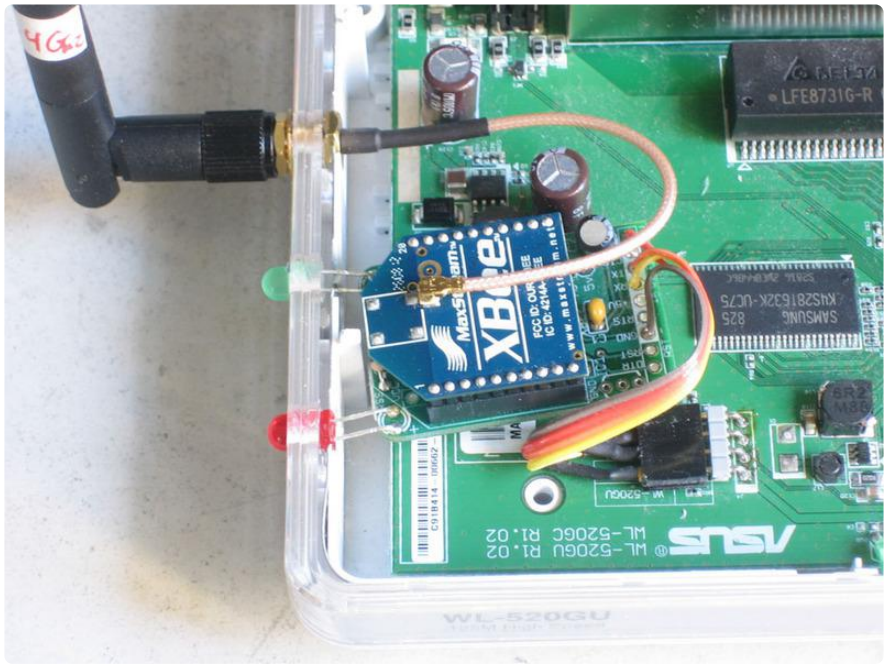
The router still works as my wifi gateway to the cablemodem, and only uses 5W.

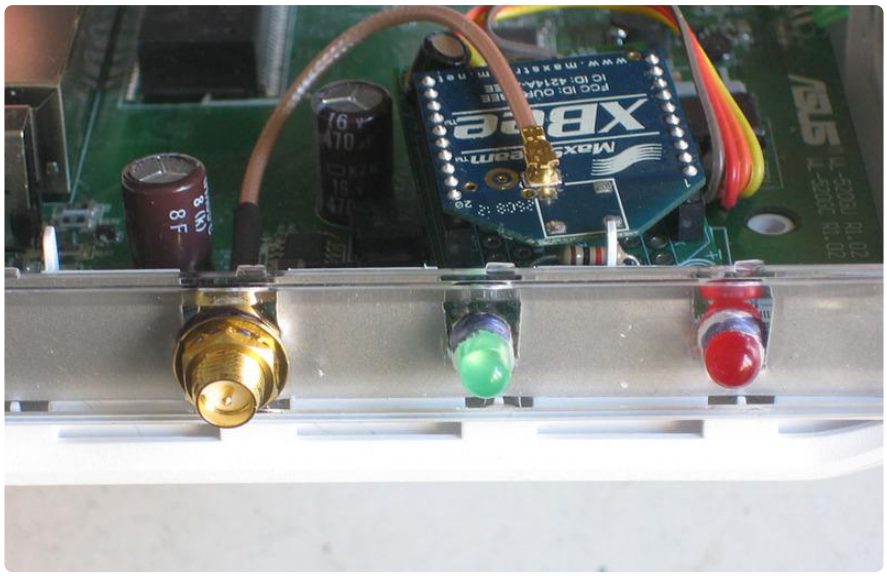
Get better range

If you have a big place, you may find that the XBee's don't have enough range, especially through doors. Unfortunately, the tweet-a-watt system as published here can't use the Pro models with much more powerful transmitters. That's because we need a huge capacitor just for the 50mA-using XBees!

However, we can make the receiver much better. To do that, replace the XBee on the receiving computer with part #XB24-AUI-001 (available from Digikey ()or Mouser ()), which is an XBee with a U.FL connector. That connector is for attaching external antennas. You'll also want a u.fl to RPSMA cable, such as Digi part# JF1R6-CR3-4I (Digi key () or Mouser ()) and a RPSMA 2.4GHz antenna. These are the same antennas used for WiFi so you may be able to scrounge one for cheap. The bigger the better!




4Gz
LFE8731G-R
WL-520GU
WL-520GC R1.02
WL-520GU R1.02
ASUS



Design

Design overview

For those interested in how to build a sensor node system with a Google Appengine backend, here is the process by which I created it. Of course, you should have the hardware part () done first!

1. Listen () - designing the parser for the computer that grabs XBee packets, and extracts the useful data
2. Store () - how to use GAE to store the data in 'the cloud'
3. Graph () - using Google Visualizations to make pretty graphs

Listen

Data listening & parsing

In this section we will work on the receiver software, that will talk to a receiver XBee and figure out what the sensor data means. I'll be writing the code in python which is a fairly-easy to use scripting language. It runs on all OS's and has tons of tutorials online. Also, Google AppEngine uses it so its a good time to learn!

This whole section assumes that you only have 1 transmitter and 1 receiver, mostly to make graphing easier to cope with. In the next section we'll tie in more sensors when we get to the datalogging part!

Raw analog input

We'll start by just getting raw data from the XBee and checking it out. The packet format for XBees is published but instead of rooting around in it, I'll just use the handy XBee library written for python. With it, I can focus on the data instead of counting bytes and calculating checksums.

To use the library, first use the pyserial module to open up a serial port (ie COM4 under windows, /dev/ttyUSB0 under mac/linux/etc) You can look at the XBee project page for information on how to figure out which COM port you're looking for. We connect at the standard default baudrate for XBees, which is 9600 and look for packets

```
from xbee import xbee
import serial

SERIALPORT = "COM4"    # the com/serial port the XBee is connected to
BAUDRATE = 9600      # the baud rate we talk to the xbee

# open up the FTDI serial port to get data transmitted to xbee
ser = serial.Serial(SERIALPORT, BAUDRATE)
ser.open()

while True:
    # grab one packet from the xbee, or timeout
    packet = xbee.find_packet(ser)
    if packet:
        xb = xbee(packet)

        print xb
```

Running this code, you'll get the following output:

```
&lt;xbee {app_id: 0x83, address_16: 1, rssi: 85, address_broadcast: False,
pan_broadcast: False, total_samples: 19, digital: [[-1, -1, -1, -1, -1, -1, -1, -1,
-1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1],
[-1 , -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1,
-1 , -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1,
-1 , -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1,
-1 , -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1,
-1 , -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1,
-1 , -1, -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1, -1,
-1 , -1], [-1, -1, -1, -1, -1, -1, -1, -1, -1]], analog: [[190, -1, -1, -1, 489,
-1 ], [109, -1, -1, -1, 484, -1], [150, -1, -1, -1, 492, -1], [262, -1, -1, -1,
492 , -1], [423, -1, -1, -1, 492, -1], [589, -1, -1, -1, 492, -1], [740, -1, -1,
-1, 492, -1], [843, -1, -1, -1, 492, -1], [870, -1, -1, -1, 496, -1], [805, -1, -1,
-1, 491, -1], [680, -1, -1, -1, 492, -1], [518, -1, -1, -1, 492, -1], [349, -1, -1,
-1, 491, -1], [199, -1, -1, -1, 491, -1], [116, -1, -1, -1, 468, -1], [108, -1, -1,
-1, 492, -1], [198, -1, -1, -1, 492, -1], [335, -1, -1, -1, 492, -1], [ 523, -1,
-1, -1, 492, -1]]}&gt;
```

...which we will reformat to make a little more legible:

```
&lt;xbee {
    app_id: 0x83,
    address_16: 1,
    rssi: 85,
    address_broadcast: False,
    pan_broadcast: False,
    total_samples: 19,
    digital: [  [-1, -1, -1, -1, -1, -1, -1, -1, -1],
                [-1, -1, -1, -1, -1, -1, -1, -1, -1],
                [-1, -1, -1, -1, -1, -1, -1, -1, -1],
                [-1, -1, -1, -1, -1, -1, -1, -1, -1],
                [-1, -1, -1, -1, -1, -1, -1, -1, -1],
                [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1],
               [-1, -1, -1, -1, -1, -1, -1, -1, -1]],
    analog: [   [190, -1, -1, -1, 489, -1],
             [109, -1, -1, -1, 484, -1],
                 [150, -1, -1, -1, 492, -1],
                 [262, -1, -1, -1, 492, -1],
                 [423, -1, -1, -1, 492, -1],
                 [589, -1, -1, -1, 492, -1],
                 [740, -1, -1, -1, 492, -1],
                 [843, -1, -1, -1, 492, -1],
                 [870, -1, -1, -1, 496, -1],
                 [805, -1, -1, -1, 491, -1],
                 [680, -1, -1, -1, 492, -1],
                 [518, -1, -1, -1, 492, -1],
                 [349, -1, -1, -1, 491, -1],
                 [199, -1, -1, -1, 491, -1],
                 [116, -1, -1, -1, 468, -1],
                 [108, -1, -1, -1, 492, -1],
                 [198, -1, -1, -1, 492, -1],
                 [335, -1, -1, -1, 492, -1],
                 [523, -1, -1, -1, 492, -1]]

}&gt;
```

OK now its clear whats going on here. First off, we get some data like the transmitter ID (address_16) and signal strength (RSSI). The packet also tells us how many sample are available (19). Now, the digital samples are all -1 because we didn't request any to be sent. The library still fills them in tho so thats why the non-data is there. The second chunk is 19 sets of analog data, ranging from 0 to 1023. As you can see, the first sample (#0) and fifth sample (#4) contain real data, the rest are -1. That corresponds to the hardware section where we setup AD0 and AD4 to be our voltage and current sensors.

We'll tweak our code so that we can extract this data only and ignore the rest of the packet.

This code creates two arrays, voltagedata and ampdata where we will stick the data. We throw out the first sample because usually ADCs are a bit wonky on the first sample and then are good to go after that. It may not be necessary, though.

```
#!/usr/bin/env python
import serial
from xbee import xbee

SERIALPORT = "COM4"    # the com/serial port the XBee is connected to
BAUDRATE = 9600      # the baud rate we talk to the xbee
CURRENTSENSE = 4       # which XBee ADC has current draw data
VOLTSENSE = 0          # which XBee ADC has mains voltage data

# open up the FTDI serial port to get data transmitted to xbee
ser = serial.Serial(SERIALPORT, BAUDRATE)
ser.open()


while True:
    # grab one packet from the xbee, or timeout
    packet = xbee.find_packet(ser)
    if packet:
        xb = xbee(packet)

        #print xb
        # we'll only store n-1 samples since the first one is usually messed up
        voltagedata = [-1] * (len(xb.analog_samples) - 1)
        ampdata = [-1] * (len(xb.analog_samples ) -1)
        # grab 1 thru n of the ADC readings, referencing the ADC constants
        # and store them in nice little arrays
        for i in range(len(voltagedata)):
            voltagedata[i] = xb.analog_samples[i+1][VOLTSENSE]
            ampdata[i] = xb.analog_samples[i+1][CURRENTSENSE]
        print voltagedata
        print ampdata
```

Now our data is easier to see:

```
Voltage: [672, 801, 864, 860, 755, 607, 419, 242, 143, 108, 143, 253, 433, 623,
760, 848, 871, 811]
Current: [492, 492, 510, 491, 492, 491, 491, 491, 492, 480, 492, 492, 492, 492,
492, 492, 497, 492]
```

Note that the voltage swings from about 100 to 900, sinusoidally.

Normalizing the data

Next up we will 'normalize' the data. The voltage should go from -170 to +170 which is the actual voltage on the line, instead of 100 to 900 which is just what the ADC reads. To do that we will get the average value of the largest and smallest reading and

subtract it from all the samples. After that, we'll normalize the Current measurements as well, to get the numbers to equal the current draw in Amperes.

```
#!/usr/bin/env python
import serial
from xbee import xbee

SERIALPORT = "COM4"    # the com/serial port the XBee is connected to
BAUDRATE = 9600      # the baud rate we talk to the xbee
CURRENTSENSE = 4       # which XBee ADC has current draw data
VOLTSENSE = 0          # which XBee ADC has mains voltage data

# open up the FTDI serial port to get data transmitted to xbee
ser = serial.Serial(SERIALPORT, BAUDRATE)
ser.open()

while True:
    # grab one packet from the xbee, or timeout
    packet = xbee.find_packet(ser)
    if packet:
        xb = xbee(packet)

        #print xb
        # we'll only store n-1 samples since the first one is usually messed up
        voltagedata = [-1] * (len(xb.analog_samples) - 1)
        ampdata = [-1] * (len(xb.analog_samples ) -1)
        # grab 1 thru n of the ADC readings, referencing the ADC constants
        # and store them in nice little arrays
        for i in range(len(voltagedata)):
            voltagedata[i] = xb.analog_samples[i+1][VOLTSENSE]
            ampdata[i] = xb.analog_samples[i+1][CURRENTSENSE]

        # get max and min voltage and normalize the curve to '0'
        # to make the graph 'AC coupled' / signed
        min_v = 1024     # XBee ADC is 10 bits, so max value is 1023
        max_v = 0
        for i in range(len(voltagedata)):
            if (min_v &gt; voltagedata[i]):
                min_v = voltagedata[i]
            if (max_v &lt; voltagedata[i]):
                max_v = voltagedata[i]

        # figure out the 'average' of the max and min readings
        avgv = (max_v + min_v) / 2
        # also calculate the peak to peak measurements
        vpp =  max_v-min_v

        for i in range(len(voltagedata)):
            #remove 'dc bias', which we call the average read
            voltagedata[i] -= avgv
            # We know that the mains voltage is 120Vrms = +-170Vpp
            voltagedata[i] = (voltagedata[i] * MAINSVPP) / vpp

        # normalize current readings to amperes
        for i in range(len(ampdata)):
            # VREF is the hardcoded 'DC bias' value, its
            # about 492 but would be nice if we could somehow
            # get this data once in a while maybe using xbeeAPI
            ampdata[i] -= VREF
            # the CURRENTNORM is our normalizing constant
            # that converts the ADC reading to Amperes
            ampdata[i] /= CURRENTNORM

        print "Voltage, in volts: ", voltagedata
```

```
        print "Current, in amps:  ", ampdata
```

We'll run this now to get this data, which looks pretty good, theres the sinusoidal voltage we are expecting and the current is mostly at 0 and then peaks up and down once in a while. Note that the current is sometimes negative but thats OK because we multiply it by the voltage and if both are negative it still comes out as a positive power draw

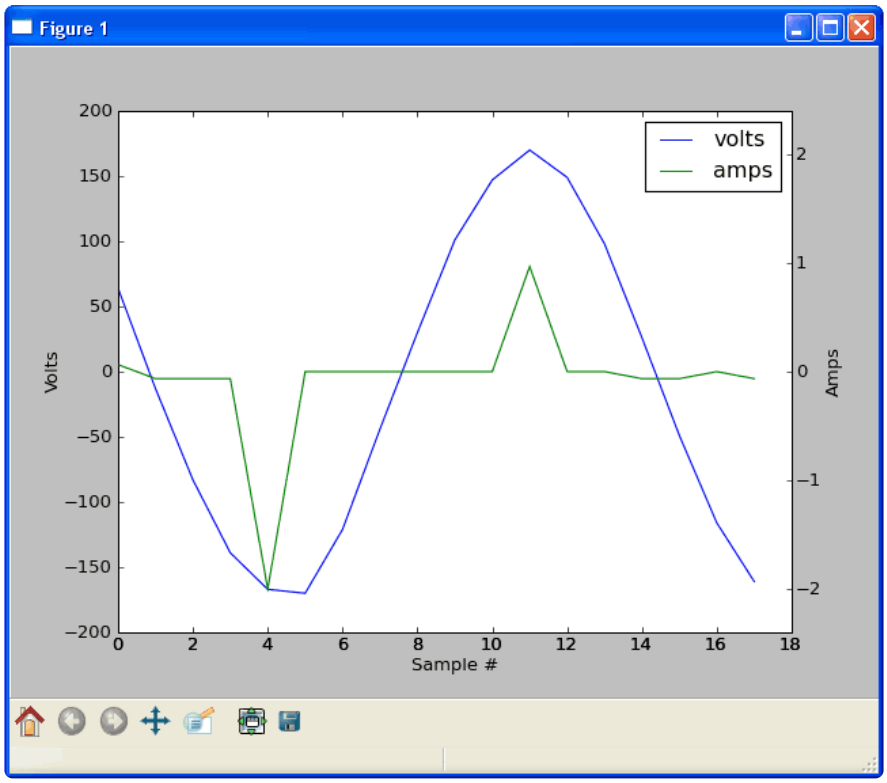
```
Voltage, in volts:  [-125, -164, -170, -128, -64, 11, 93, 148, 170, 161, 114, 46,
-39, -115, -157, -170, -150, -99]

Current, in amps:   [0.064516129032258063, -1.096774193548387, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 1.096774193548387,
                     0.0, 0.0, 0.0, -0.064516129032258063, 0.0, 0.0,
-0.70967741935483875, 0.0, 0.0]
```

Basic data graphing

Finally, I'm going to add a whole bunch more code that will use the numpy graphing modules to make a nice graph of our data. Note that you'll need to install wxpython as well as numpy, and matplotlib!

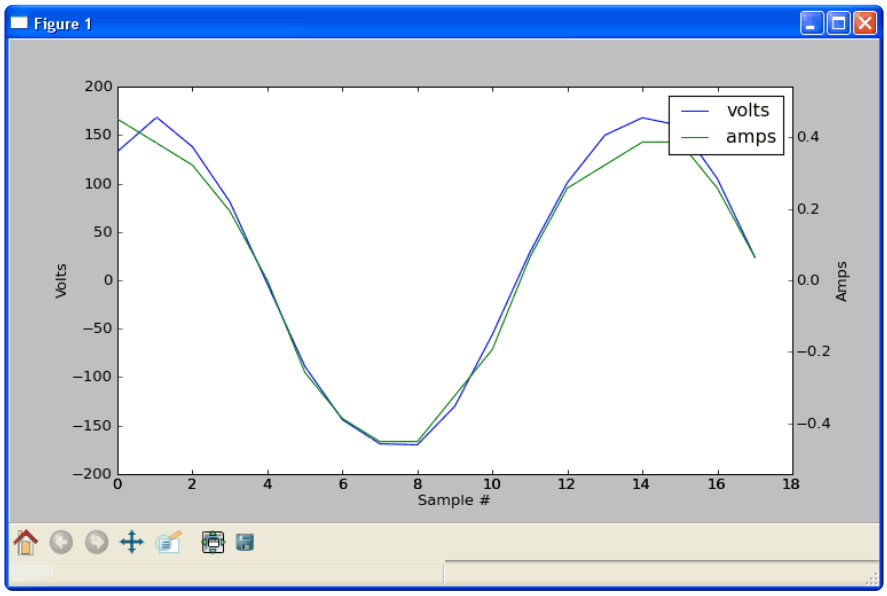
At this point, the code is getting waaay to big to paste here so grab "wattcher.py Mains graph" from the download page! ()

Run it and you should see a graph window pop up with a nice sinusoidal voltage graph and various amperage data. For example this first graph is of a laptop plugged in. You'll see that its a switching supply, and only pulls power during the peak of the voltage curve.



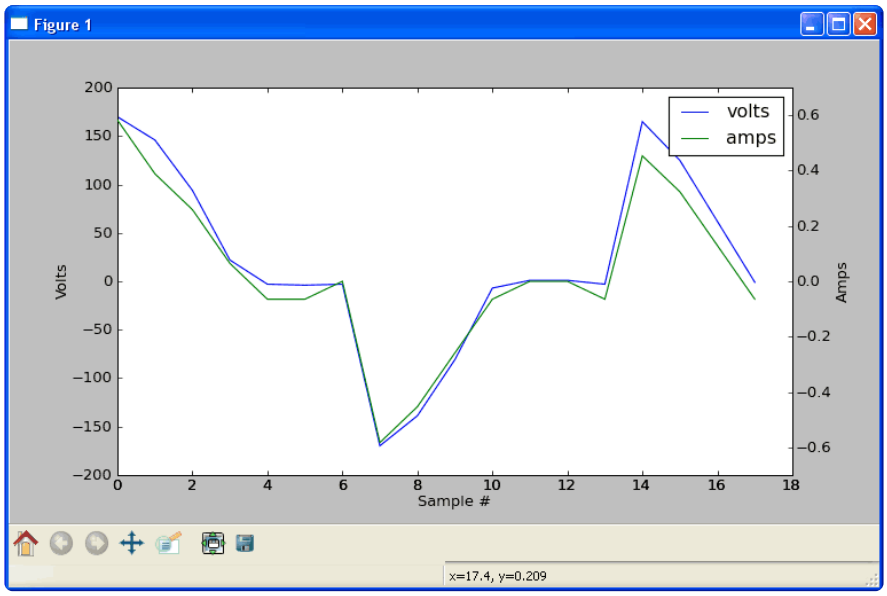
A laptop plugged in, switching power supply (above)

Now lets try plugging in a 40W incandescent light bulb. You'll notice that unlike the switching supply, the current follows the voltage almost perfectly. Thats because a lightbulb is just a resistor!



40W lightbulb (above)

Finally, lets try sticking the meter on a dimmable switch. You'll see that the voltage is 'chopped' up, no longer sinusoidal. And although the current follows the voltage, its still matching pretty well.

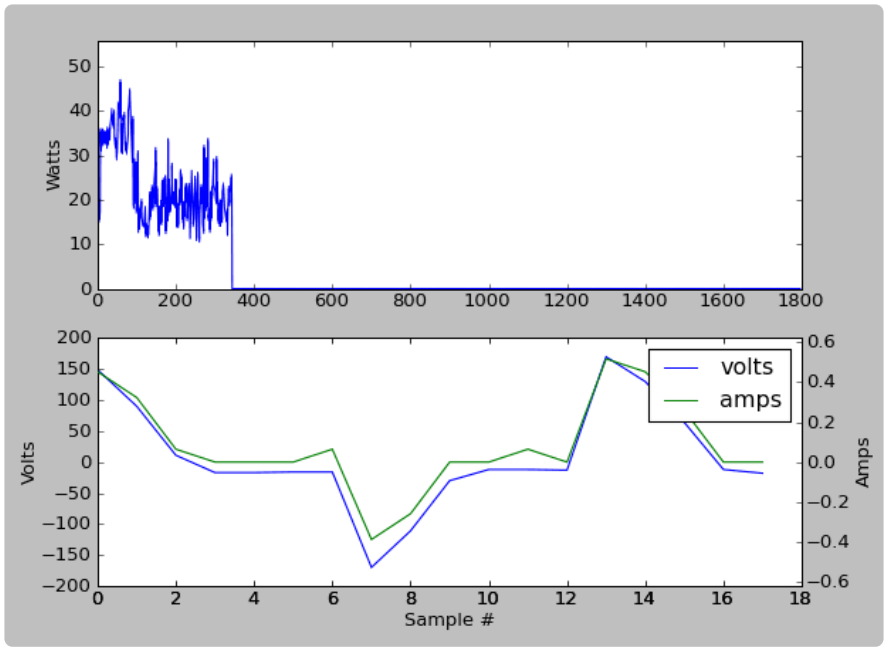


Light bulb on dimmer switch (above)

Graphing wattage!

OK neat, its all fun to watch waveforms but what we -really want- is the Watts used. Remember, P = VI otherwise known as Watts = Voltage * Current. We can calculate total Watts used by multiplying the voltages and currents at each sample point, then summing them up over a cycle & averaging to get the power used per cycle. Once we have Watts, its easy to just multiply that by 'time' to get Watt-hours!

Download and run the wattcher.py - watt grapher script from the download page ()



Now you can watch the last hour's worth of watt history (3600 seconds divided by 2 seconds per sample = 1800 samples) In the image above you can see as I dim a 40-watt lightbulb. The data is very 'scattered' looking because we have not done any low-pass filtering. If we had a better analog sampling rate, this may not be as big a deal but with only 17 samples to work with, precision is a little difficult.

Done!

OK great! We have managed to read data, parse out the analog sensor payload and process it in a way that gives us meaningful graphs. Of course, this is great for instantaneous knowledge but it -would- be nice if we could have longer term storage, and also keep track of multiple sensors. In the next step we will do that by taking advantage of some free 'cloud computing' services!

Store

OK we are getting good data from our sensors, lets corral it into more useful chunks and store it in a database. We could make a database on the computer, but since we'd like to share this data, it makes more sense to put it online. There are custom services that are specifically designed to do this sort of thing like Pachube () but I'm going to reinvent the wheel and design my own web-app that stores and displays energy data. (Mostly I want to play around with Google App Engine!)

You have 5 minutes!

We get data every few seconds from the XBee modem inside the kill-a-watt. We could, in theory, put data into our database every 2 seconds but that would quickly balloon the amount of storage necessary. It would also make sorting through the data difficult. So instead lets add up all the sensor data for 5 minutes and then take the average.

We'll do this by keeping two timers and one tally. One timer will track how long its been since the last sensor signal was sent, and the other will track if its been 5 minutes. The tally will store up all the Watt-hours (Watt measurements * time since last sensor data). Then at the end we can average by the 5 minutes

This chunk of code goes near the beginning, it creates the timers and tally and initializes them

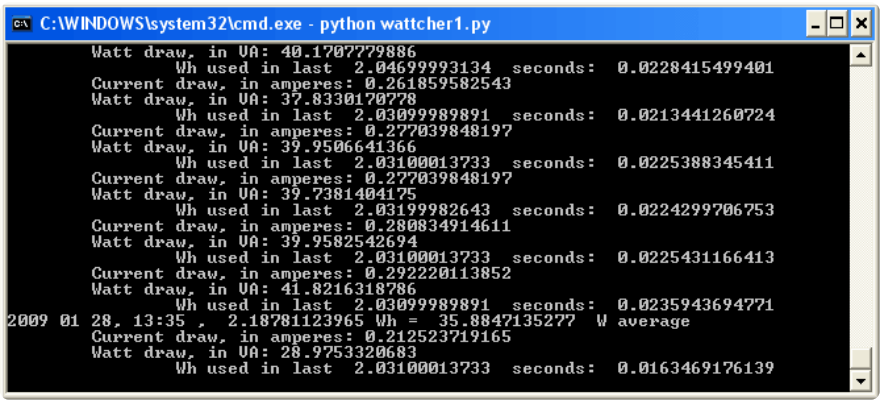
```
...
fiveminutetimer = lasttime = time.time()  # get the current time
cumulativewatthr = 0
...
```

Then later on, after we get our data we can put in this chunk of code:

```
        # add up the delta-watthr used since last reading
        # Figure out how many watt hours were used since last reading
        elapsedseconds = time.time() - lasttime
        dwatthr = (avgwatt * elapsedseconds) / (60.0 * 60.0)  # 60 seconds in 60
minutes = 1 hr
        lasttime = time.time()
        print "\t\tWh used in last ",elapsedseconds," seconds: ",dwatthr
        cumulativewatthr += dwatthr

        # Determine the minute of the hour (ie 6:42 -&gt; '42')
        currminute = (int(time.time())/60) % 10
        # Figure out if its been five minutes since our last save
        if (((time.time() - fiveminutetimer) &gt;= 60.0) and (currminute % 5 == 0)):
            # Print out debug data, Wh used in last 5 minutes
            avgwattsused = cumulativewatthr * (60.0*60.0 / (time.time() -
fiveminutetimer))
            print time.strftime("%Y %m %d, %H:%M"),", ",cumulativewatthr,"Wh =
",avgwattsused," W average")

            # Reset our 5 minute timer
            fiveminutetimer = time.time()
            cumulativewatthr = 0
```

Note that we calculate delta-watthours, the small amount of power used every few seconds. Then we can get the average watts used by dividing the watthours by the number of hours that have passed (about 1/12th). Instead of going by exact 5 minutes, I decided to only report on the 5's of the hour (:05, :10, etc) so that its easier to send all the data at once if theres multiple sensors that started up at different times.

Download wattcher-5minreporter.py from the Download page (). If you run this, youll get a steady stream of:



Near the end you can see the timestamp, the Watthrs used in the last few minutes and the average Wattage.

Multisensor!

We have good data but so far it only works with one sensor. Multiple sensors will mess it up! Time to add support for more than one XBee so that I can track a few rooms. I'll do that by creating an object class in python, and using the XBee address (remember that from part 1?) to track. I'll replace the code we just wrote with the following:

At the top, instead of the timer variables, I'll have a full class declaration, and create an array to store them:

```
####### store sensor data and array of histories per sensor
class Fiveminutehistory:
  def __init__(self, sensornum):
      self.sensornum = sensornum
      self.fiveminutetimer = time.time()  # track data over 5 minutes
      self.lasttime = time.time()
      self.cumulativewatthr = 0

  def addwatthr(self, deltawatthr):
      self.cumulativewatthr +=  float(deltawatthr)

  def reset5mintimer(self):
      self.cumulativewatthr = 0
      self.fiveminutetimer = time.time()

  def avgwattover5min(self):
      return self.cumulativewatthr * (60.0*60.0 / (time.time() -
self.fiveminutetimer))

  def __str__(self):
      return "[ id#: %d, 5mintimer: %f, lasttime; %f, cumulativewatthr: %f ]" %
(self.sensornum, self.fiveminutetimer, self.lasttime, self.cumulativewatthr)

######### an array of histories
sensorhistories = []
```

When the object is initialized with the sensor ID number, it also sets up the two timers and cumulative Watthrs tracked. I also created a few helper functions that will make the code cleaner

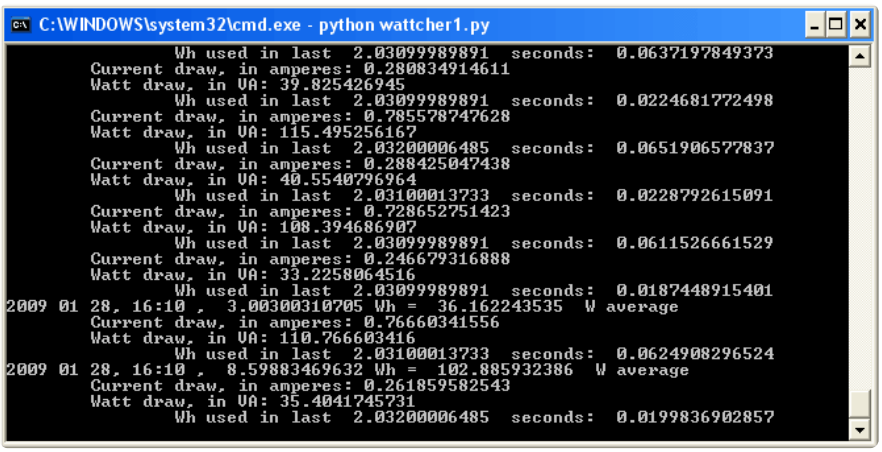
Right below that I'll create a little function to help me create and retrieve these objects. Given an XBee ID number it either makes a new one or gets the reference to it

```
####### retriever
def findsensorhistory(sensornum):
    for history in sensorhistories:
        if history.sensornum == sensornum:
            return history
    # none found, create it!
    history = Fiveminutehistory(sensornum)
    sensorhistories.append(history)
    return history
```

Finally, instead of the average Watt calculation code written up above, we'll replace it with the following chunk, which retreives the object and tracks power usage with the object timers:

```
        # retreive the history for this sensor
        sensorhistory = findsensorhistory(xb.address_16)
        #print sensorhistory

        # add up the delta-watthr used since last reading
        # Figure out how many watt hours were used since last reading
        elapsedseconds = time.time() - sensorhistory.lasttime
        dwatthr = (avgwatt * elapsedseconds) / (60.0 * 60.0)  # 60 seconds in 60
minutes = 1 hr
        sensorhistory.lasttime = time.time()
        print "\t\tWh used in last ",elapsedseconds," seconds: ",dwatthr
        sensorhistory.addwatthr(dwatthr)

        # Determine the minute of the hour (ie 6:42 -&gt; '42')
        currminute = (int(time.time())/60) % 10
        # Figure out if its been five minutes since our last save
        if (((time.time() - sensorhistory.fiveminutetimer) &gt;= 60.0) and
(currminute % 5 == 0)):
            # Print out debug data, Wh used in last 5 minutes
            avgwattsused = sensorhistory.avgwattover5min()
            print time.strftime("%Y %m %d, %H:%M"),",
",sensorhistory.cumulativewatthr,"Wh = ",avgwattsused," W average"

            # Reset our 5 minute timer
            sensorhistory.reset5mintimer()
```

The code basically acts the same except now it wont choke on multiple sensor data! Below, my two Kill-a-Watts, one with a computer attached (100W) and another with a lamp (40W):



Onto the database!

The App Engine

So we want to have an networked computer to store this data so we can share the data, but we really don't want to have to run a server from home! What to do? Well as mentioned before, you can use Pachube or similar, but I will show how to roll-your-own with Google App Engine (GAE) (). GAE is basically a free mini-webserver hosted by Google, that will run basic webapps without the hassle of administrating a database server. Each webapp has storage, some frameworks and can use Google accounts for authentication. To get started I suggest checking out the GAE website, documentation, etc (). I'll assume you've gone through the tutorials () and jump right into designing my power data storage app called Wattcher (a little confusing I know)

First, the app.yaml file which defines my app looks like this:

```
application: wattcher
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: wattcherapp.py
```

Pretty simple, just says that the app uses wattcherapp.py as the source file

Next, we'll dive into the python code for our webapp. First, the includes and database index. To create a database, we actually define it -in the python file-, GAE then figures out what kind of database to create for you by following those directions (very different than MySQL where you'd create the DB separately)

```
import cgi, datetime

from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db

class Powerusage(db.Model):
  author = db.UserProperty()           # the user
  sensornum = db.IntegerProperty()     # can have multiple sensors
  watt = db.FloatProperty()          # each sending us latest Watt measurement
  date = db.DateTimeProperty(auto_now_add=True)    # timestamp
```

We use the default includes. We have a single database table called Powerusage, and it has 4 entries: one for the user, one for the sensor number, one for the last reported Watts used and one for a datestamp

Each 'page' or function of our webapp needs its own class. Lets start with the function that allows us to store data in the DB. I'll call it PowerUpdate.

```
class PowerUpdate(webapp.RequestHandler):
  def get(self):

    # make the user log in
    if not users.get_current_user():
        self.redirect(users.create_login_url(self.request.uri))

    powerusage = Powerusage()

    if users.get_current_user():
        powerusage.author = users.get_current_user()
    #print self.request
    if self.request.get('watt'):
        powerusage.watt = float(self.request.get('watt'))
    else:
         self.response.out.write('Couldnt find \'watt\' GET property!')
         return
    if  self.request.get('sensornum'):
        powerusage.sensornum = int(self.request.get('sensornum'))
    else:
        powerusage.sensornum = 0   # assume theres just one or something

    powerusage.put()
    self.response.out.write('OK!')
```

When we send a request to do that with a GET call (ie requesting the webpage), we'll first make sure the user is authenticated and logged in so we know their name. Then we'll create a new database entry by initializing a new instantiation of Powerusage. Then we'll look the GET request for the watt data, which would be in the format watt=39.2 or similar. That's parsed for us, thankfully and we can also get the sensor number which is passed in the format sensornum=3. Finally we can store the data into the permanent database

Next is a useful debugging function, it will simply print out all the data it has received for your account!

```
class DumpData(webapp.RequestHandler):
  def get(self):

    # make the user log in
    if not users.get_current_user():
        self.redirect(users.create_login_url(self.request.uri))

    self.response.out.write('&lt;html&gt;&lt;body&gt;Here is all the data you have
sent us:&lt;p&gt;')

    powerusages = db.GqlQuery("SELECT * FROM Powerusage WHERE author = :1 ORDER BY
date", users.get_current_user())
```

```
    for powerused in powerusages:
        if powerused.sensornum:
          self.response.out.write('&lt;b&gt;%s&lt;/b&gt;\'s sensor #%d' %
                                  (powerused.author.nickname(),
powerused.sensornum))
        else:
          self.response.out.write(&lt;b&gt;%s&lt;/b&gt;' %
powerused.author.nickname())

        self.response.out.write(' used: %f Watts at %s&lt;p&gt;' % (powerused.watt,
powerused.date))
    self.response.out.write("&lt;/body&gt;&lt;/html&gt;")
```

This function simply SELECT's (retrieves) all the entries, sorts them by date and prints out each one at a time

Finally we'll make a basic 'front page' that will show the last couple of datapoints sent:

```
class MainPage(webapp.RequestHandler):
  def get(self):

    self.response.out.write('&lt;html&gt;&lt;body&gt;Welcome to Wattcher!
&lt;p&gt;Here are the last 10 datapoints:&lt;p&gt;')

    powerusages = db.GqlQuery("SELECT * FROM Powerusage ORDER BY date DESC LIMIT
10")

    for powerused in powerusages:
        if powerused.sensornum:
          self.response.out.write('&lt;b&gt;%s&lt;/b&gt;\'s sensor #%d' %
                                  (powerused.author.nickname(),
powerused.sensornum))
        else:
          self.response.out.write('&lt;b&gt;%s&lt;/b&gt;' %
powerused.author.nickname())

        self.response.out.write(' used: %f Watts at %s&lt;p&gt;' % (powerused.watt,
powerused.date))
    self.response.out.write("&lt;/body&gt;&lt;/html&gt;")
```

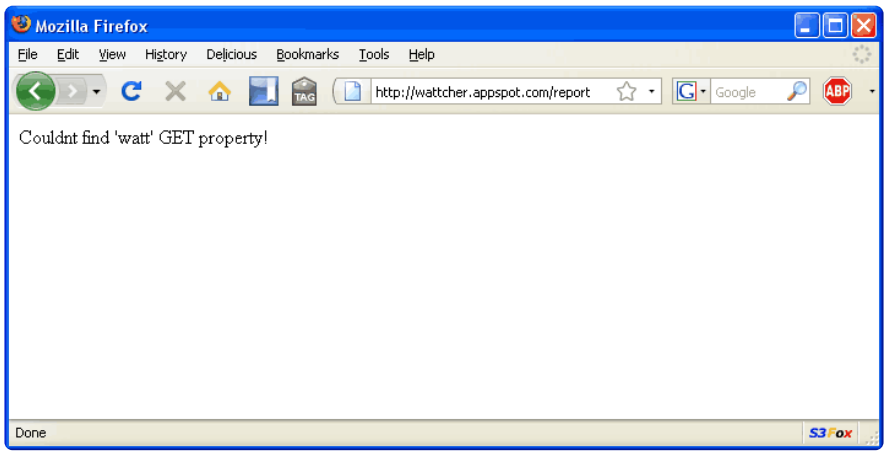
It's very similar to the DataDump function but its only 10 points of data and from all users, nice to use when you just want to 'check it out' but don't want to log in

Finally, we have a little initializer structure that tells GAE what pages link to what functions:

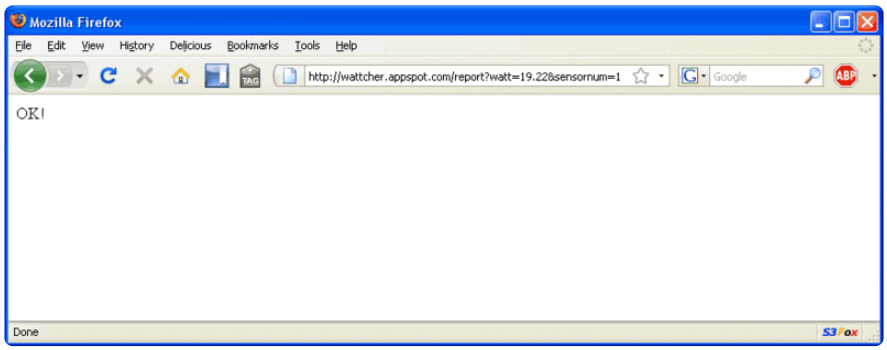
```
application = webapp.WSGIApplication(
    [('/', MainPage),
     ('/report', PowerUpdate),
     ('/dump', DumpData)],
    debug=True)

def main():
  run_wsgi_app(application)

if __name__ == "__main__":
  main()
```

Test!

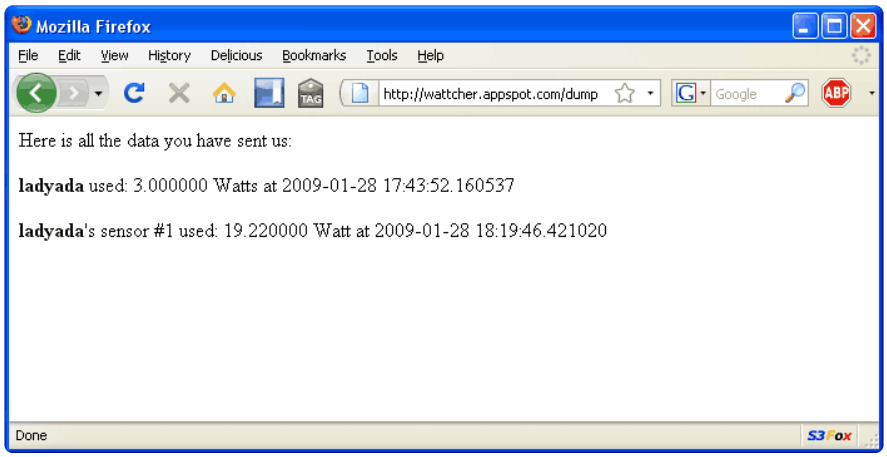
OK lets try it out, first lets visit http://wattcher.appspot.com/report ()



Remember we made it a requirement to supply -some- data. Lets try again http://wattcher.appspot.com/report?watt=19.22&sensornum=1 ()

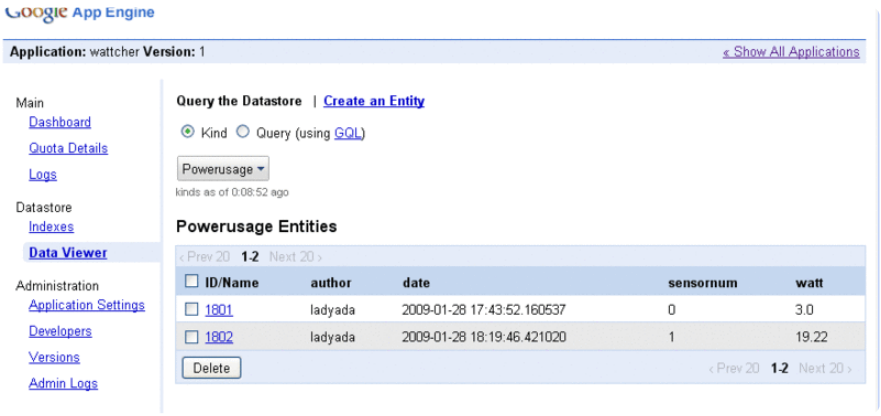


Yay we got an OK! Lets check out the data stored by visiting http://wattcher.appspot.com/dump ()



There's two entries because I did a little testing beforehand but you can see that there are 2 entries. Nice!

We can also visit the GAE control panel and browse the data 'by hand'



Anyways, now that that's working, lets go back and add the reporting technology to our sensor-reader script.

Getting the report out

Only a little more hacking on the computer script and we're done. We want to add support for sending data to GAE. Unfortunately right now our authentication is done through Google accounts so its not easy to run on an Arduino. To adapt it you'd have to send the username in the Report GET and hope nobody else uses the same one (unless you also add a basic password system)

Anyhow, I totally ripped off how to do this from some nice people on the Internet ()

Download appengineauth.py from the download page (), and change the first few lines if necessary. We hardcode the URL we're going to and the account/password as well as the GAE app name

```
users_email_address = "myaccount@gmail.com"
users_password      = "mypassword"
my_app_name = "wattcher"
target_authenticated_google_app_engine_uri = 'http://wattcher.appspot.com/report'
```

The real work happens at this function sendreport where it connects and sends the Watt data to the GAE site:

```
def sendreport(sensornum, watt):
    # this is where I actually want to go to
    serv_uri = target_authenticated_google_app_engine_uri + "?watt="+str(watt)
+"&amp;sensornum="+str(sensornum)

    serv_args = {}
```

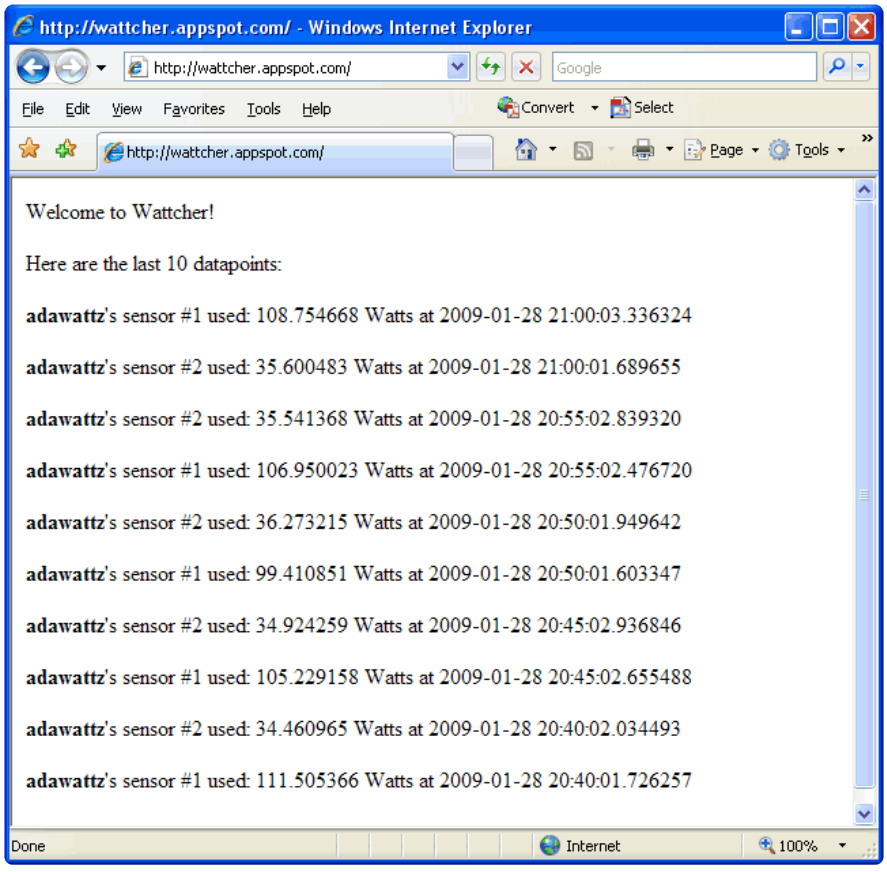
```
    serv_args['continue'] = serv_uri
    serv_args['auth']     = authtoken

    full_serv_uri = "http://wattcher.appspot.com/_ah/login?%s" %
(urllib.urlencode(serv_args))

    serv_req = urllib2.Request(full_serv_uri)
    serv_resp = urllib2.urlopen(serv_req)
    serv_resp_body = serv_resp.read()

    # serv_resp_body should contain the contents of the
    #  target_authenticated_google_app_engine_uri page - as we will have been
    #  redirected to that page automatically
    #
    # to prove this, I'm just gonna print it out
    print serv_resp_body
```

Finally, we wrap up by adding the following lines to our computer script, which will send the data nicely over to GAE!

```
  # Also, send it to the app engine
            appengineauth.sendreport(xb.address_16, avgwattsused)
```

You can download the final script wattcher.py - final from the download page ()!

Don't forget to visit wattcher.appspot.com to check out the lastest readings:



Graph

Making pretty pictures

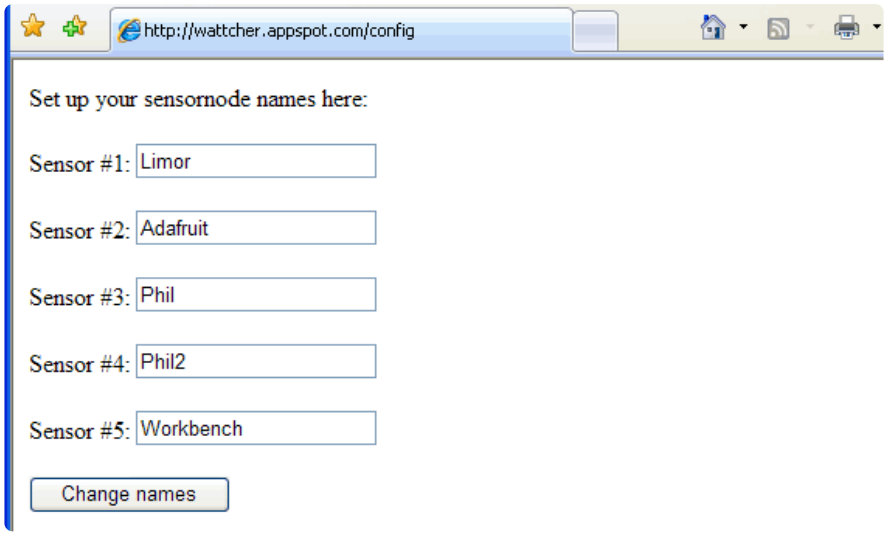
Data is great, but visualizations are better. In this step we'll manipulate our stored history so that we can make really nice graphs!

First we'll start by making our sensors named, so that its easier for us to keep track of which is what. Then we'll look at our graph options and data formats. Finally we'll reformat our data to make it ready for graphing

Configuring the sensor names

Its no fun to have data marked as "sensor #1" so I added a 'config' page where the app engine code looks at what sensor numbers have sent data to the database and then allows you to name them. Of course, you need to have the sensor on and sending data -first- before this will work

The configure screen looks something like this:

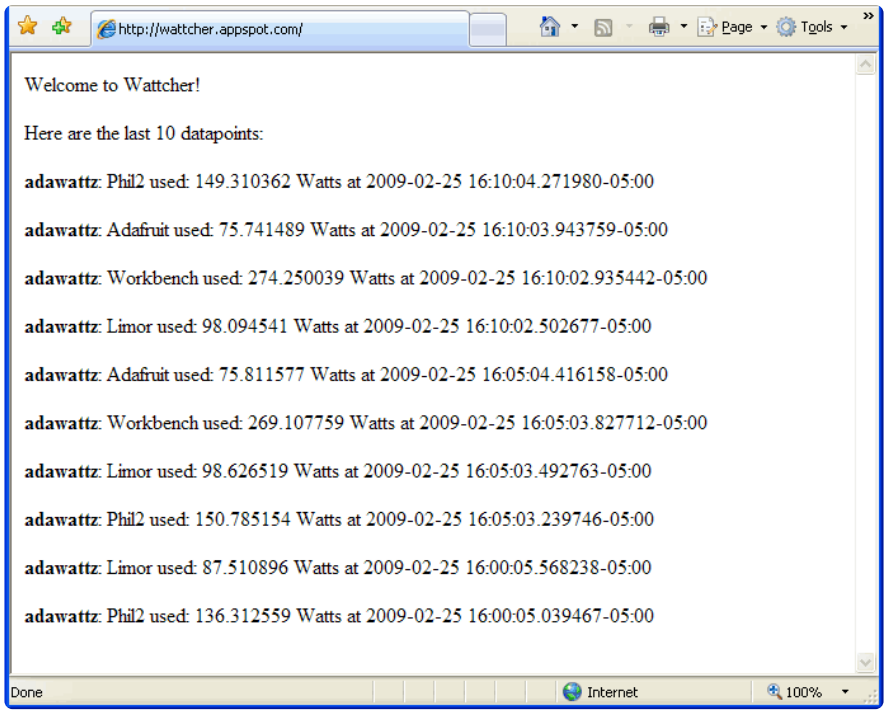


This code uses GET when it should really use POST. I'm kinda old and dont like debugging with POST so...yeah.

```
class Configure(webapp.RequestHandler):
  def get(self):
    # make the user log in if no user name is supplied
    if self.request.get('user'):
      account = users.User(self.request.get('user'))
    else:
       if not users.get_current_user():
```

```
          self.redirect(users.create_login_url(self.request.uri))
        account = users.get_current_user()


    self.response.out.write('&lt;html&gt;&lt;body&gt;Set up your sensornode names
here:&lt;p&gt;')

    # find all the sensors up to #10
    sensorset = []
    for i in range(10):
      c = db.GqlQuery("SELECT * FROM Powerusage WHERE author = :1 and sensornum = :
2", users.get_current_user(), i)
      if c.get():
        sensorset.append(i)

    self.response.out.write('&lt;form action="/config" method="get"&gt;')
    for sensor in sensorset:
      name = ""
      currnamequery = db.GqlQuery("SELECT * FROM Sensorname WHERE author = :1 AND
sensornum = :2", users.get_current_user(), sensor)
      currname = currnamequery.get()

      # first see if we're setting it!
      if self.request.get('sensornum'+str(sensor)):
        name = self.request.get('sensornum'+str(sensor))
        if not currname:
          currname = Sensorname()  # create a new entry
          currname.sensornum = sensor
          currname.author = users.get_current_user()
        currname.sensorname = name
        currname.put()
      else:
      # we're not setting it so fetch current entry
        if currname:
           name = currname.sensorname

      self.response.out.write('Sensor #'+str(sensor)+': &lt;input type="text"
name="sensornum'+str(sensor)+'" value="'+name+'"&gt;&lt;/text&gt;&lt;p&gt;')

    self.response.out.write("""&lt;div&gt;&lt;input type="submit" value="Change
names"&gt;&lt;/div&gt;
      &lt;/form&gt;
      &lt;/body&gt;
      &lt;/html&gt;""")
```

Now we can have more useful data in the history dump:

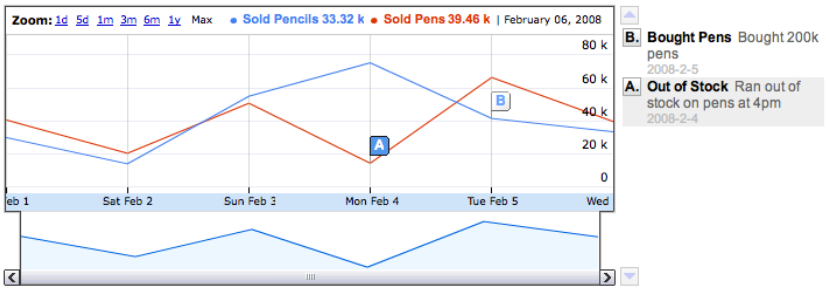


Now we can see that Phil is mostly to blame for our power bill!

Google Visualizer

So we have data and we'd like to see our power usage history. Graphing data is a lot of work, and I'm lazy. So I look online and find that Google -also- has a visualization API! ()This means I don't have to write a bunch of graphical code, and can just plug into their system. Sweet!

OK checking out the gallery of available visualizations (), I'm fond of this one, the Annotated Time Line ():



Note how you can easily see the graphs, scroll around, zoom in and out and each plot is labeled. Perfect for plotting power data!

To see the above visualizer as actively running outside the Learning System on a website, check it out the first example here ().

Data formatting

Theres a few restrictions to how we get the data to the visualization api and our best option is JSon data. As far as I can tell, JSON is what happened when everyone said "wow, XML is really bulky and wasteful". Anyhow, theres like 4 layers of framework and interpretive data structions and in the end there was a pretty easy to use library written by the Google Visualizations team that let me 'just do it' with a single call by putting the data into a python 'dictionary' in a certain format.

Lets go through the code in sections, since the function is quite long

```
class JSONout(webapp.RequestHandler):
  def get(self):

    # make the user log in if no user name is supplied
    if self.request.get('user'):
      account = users.User(self.request.get('user'))
    else:
       if not users.get_current_user():
         self.redirect(users.create_login_url(self.request.uri))
       account = users.get_current_user()

    # assume we want 24 hours of data
    historytimebegin = 24
    if self.request.get('bhours'):
      historytimebegin = int(self.request.get('bhours'))


    # assume we want data starting from 0 hours ago
    historytimeend = 0
    if self.request.get('ehours'):
      historytimeend = int(self.request.get('ehours'))

    # data format for JSON happiness
    datastore = []
    columnnames = ["date"]
    columnset = set(columnnames)
    description ={"date": ("datetime", "Date")}

    # the names of each sensor, if configured
    sensornames = [ None ] * 10
```

First up we get the user we're going to be looking up the data for. Then we have two variables for defining the amount of data to grab. One is "ehours" (end hours) and the other is "bhours". So if you wanted the last 5 hours, bhours would be 5 and ehours would be 0. If you wanted 5 hours from one day ago, bhours would be 29 and ehours would be 24. datastore is where we will corall all the data. columnnames and description are the 'names' of each column. We always have a date column, then

another column for each sensor stream. We also have a seperate array to cache the special sensor names.

onto the next section! Here is where we actually grab data from the database. Now app engine has this annoying restriction, you can only get 1000 points of data at once so what I do is go through it 12 hours at a time. The final datastore has all the points but its easier on the database, I guess. One thing thats confusing perhaps is each column has a name and a description. The name is short, say "watts3" for sensor #3, but the description might be "Limor's workbench". I dont even remember writing this code so maybe you can figure it out on your own?

```
    # we cant grab more than 1000 datapoints, thanks to free-app-engine restriction
    # thats about 3 sensors's worth in one day
    # so we will restrict to only grabbing 12 hours of data at a time, about 7
sensors worth

    while (historytimebegin &gt; historytimeend):
      if (historytimebegin - historytimeend) &gt; 12:
        timebegin = datetime.timedelta(hours = -historytimebegin)
        timeend = datetime.timedelta(hours = -(historytimebegin-12))
        historytimebegin -= 12
      else:
        timebegin = datetime.timedelta(hours = -historytimebegin)
        historytimebegin = 0
        timeend = datetime.timedelta(hours = -historytimeend)

      # grab all the sensor data for that time chunk
      powerusages = db.GqlQuery("SELECT * FROM Powerusage WHERE date &gt; :1 AND
date &lt; :2 AND author = :3 ORDER BY date", datetime.datetime.now()+timebegin,
datetime.datetime.now()+timeend, account)

      # sort them into the proper format and add sensor names from that DB if not
done yet
      for powerused in powerusages:
        coln = "watts" + str(powerused.sensornum)
        entry = {"date": powerused.date.replace(tzinfo=utc).astimezone(est), coln:
powerused.watt}
        if not (coln in columnset):
          columnnames.append(coln)
          columnset = set(columnnames)
          # find the sensor name, if we can
          if (len(sensornames) &lt; powerused.sensornum) or (not
sensornames[powerused.sensornum]):
            currnamequery = db.GqlQuery("SELECT * FROM Sensorname WHERE author = :1
AND sensornum = :2", account, powerused.sensornum)
            name = currnamequery.get()

            if not name:
              sensornames[powerused.sensornum] = "sensor #"+str(powerused.sensornum)
            else:
              sensornames[powerused.sensornum] = name.sensorname

          description[coln] = ("number", sensornames[powerused.sensornum])
          #self.response.out.write(sensornames)

        # add one entry at a time
        datastore.append(entry)
```

Finally at the end of all the looping, we call the magic function that turns the dictionary into JSON, wrap it in the proper Google Visualization package, then spit it out!

```
    # OK all the data is ready to go, print it out in JSON format!
    data_table = gviz_api.DataTable(description)
    data_table.LoadData(datastore)
    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write(data_table.ToJSonResponse(columns_order=(columnnames),
                                  order_by="date"))
```

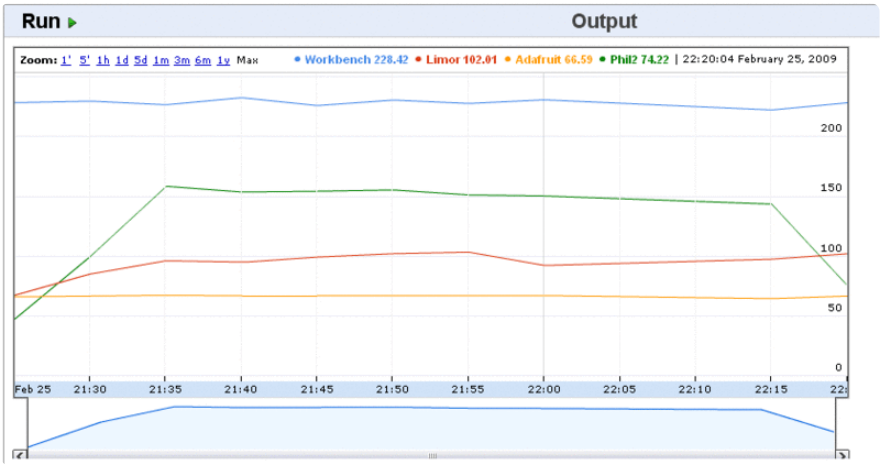
If you were to visit http://wattcher.appspot.com/visquery.json?user=adawattz@gmail.com&bhours=1 () it would output something like this:

```
google.visualization.Query.setResponse({'version':'0.5', 'reqId':'0',
 'status':'OK', 'table': {cols: [{id:'date',label:'Date',type:
 'datetime'},{id:'watts1',label:'Limor',type:'number'},{id:'watts5',
 label:'Workbench',type:'number'},{id:'watts2',label:'Adafruit',type:
 'number'},{id:'watts4',label:'Phil2',type:'number'}],rows: [{c:[{v:new
Date(2009,1,25,21,20,2)},{v:64.8332291619},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,20,3)},,{v:230.122099757},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,20,3)},,,{v:65.4923925044},{v:null}]},{c:[{v:new
Date(2009,1,25,21,20,4)},,,,{v:48.6947643311}]},{c:[{v:new
Date(2009,1,25,21,25,3)},,{v:228.409810208},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,25,3)},{v:67.3574917331},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,25,3)},,,{v:66.0046383897},{v:null}]},{c:[{v:new
Date(2009,1,25,21,25,4)},,,,{v:47.3892235642}]},{c:[{v:new
Date(2009,1,25,21,30,2)},{v:84.9379517795},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,30,3)},,,,{v:99.7553490071}]},{c:[{v:new
Date(2009,1,25,21,30,5)},,{v:229.73642288},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,30,6)},,,{v:66.6556291818},{v:null}]},{c:[{v:new
Date(2009,1,25,21,35,2)},,,{v:67.3146052998},{v:null}]},{c:[{v:new
Date(2009,1,25,21,35,3)},{v:96.2322216676},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,35,3)},,{v:226.678267688},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,35,4)},,,,{v:158.428422765}]},{c:[{v:new
Date(2009,1,25,21,40,3)},,{v:232.644574879},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,40,4)},,,,{v:153.666193493}]},{c:[{v:new
Date(2009,1,25,21,40,6)},,,{v:66.7874343225},{v:null}]},{c:[{v:new
Date(2009,1,25,21,40,12)},{v:95.0019590395},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,40,21)},{v:95.0144043571},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,40,23)},,,{v:66.8060307611},{v:null}]},{c:[{v:new
Date(2009,1,25,21,45,2)},,,{v:66.9814723201},{v:null}]},{c:[{v:new
Date(2009,1,25,21,45,3)},,{v:226.036818816},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,45,3)},{v:99.2775581827},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,45,4)},,,,{v:154.261889366}]},{c:[{v:new
Date(2009,1,25,21,50,4)},{v:102.104642018},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,50,4)},,,,{v:155.441084531}]},{c:[{v:new
Date(2009,1,25,21,50,5)},,,{v:67.0087146687},{v:null}]},{c:[{v:new
Date(2009,1,25,21,50,5)},,{v:230.678636915},,{v:null}]},{c:[{v:new
Date(2009,1,25,21,55,3)},{v:103.493297176},,,{v:null}]},{c:[{v:new
Date(2009,1,25,21,55,3)},,,,{v:151.309223916}]},{c:[{v:new
Date(2009,1,25,21,55,4)},,,{v:66.9174858741},{v:null}]},{c:[{v:new
Date(2009,1,25,21,55,4)},,{v:227.765325835},,{v:null}]},{c:[{v:new
Date(2009,1,25,22,0,3)},,,{v:67.0004310254},{v:null}]},{c:[{v:new
Date(2009,1,25,22,0,3)},,,,{v:150.389989112}]},{c:[{v:new
Date(2009,1,25,22,0,3)},,{v:230.892049553},,{v:null}]},{c:[{v:new
Date(2009,1,25,22,0,4)},{v:92.2432771363},,,{v:null}]},{c:[{v:new
Date(2009,1,25,22,15,3)},{v:97.5910440774},,,{v:null}]},{c:[{v:new
Date(2009,1,25,22,15,3)},,,,{v:143.722595861}]},{c:[{v:new
Date(2009,1,25,22,15,4)},,,{v:64.4898008851},{v:null}]},{c:[{v:new
Date(2009,1,25,22,15,4)},,{v:222.357617868},,{v:null}]}]}});
```

Anyways, you can kinda see the data, also note its actually a function call, this stuff is really kinky!

Now go to the Google Visualizations Playground () and enter in that URL into the sandbox

And you can see the visualization itself pop out! (this is just a screen shot so go do it yerself if you want to mess around)



OK go mess around, adding and changing bhours and ehours

Wrapping up the visualization

OK we're nearly done. Now we just need to basically grab the code from the sandbox and make it a subpage in our app engine...like so:

```
class Visualize(webapp.RequestHandler):
  def get(self):

    # make the user log in if no user name is supplied
    if self.request.get('user'):
      account = users.User(self.request.get('user'))
    else:
       if not users.get_current_user():
         self.redirect(users.create_login_url(self.request.uri))
       account = users.get_current_user()
```

```
    historytimebegin = 24 # assume 24 hours
    if self.request.get('bhours'):
      historytimebegin = int(self.request.get('bhours'))


    historytimeend = 0 # assume 0 hours ago
    if self.request.get('ehours'):
      historytimeend = int(self.request.get('ehours'))

    # get the first part, headers, out
    self.response.out.write('''
&lt;!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd"&gt;
&lt;html xmlns="http://www.w3.org/1999/xhtml"&gt;
&lt;head&gt;
  &lt;meta http-equiv="content-type" content="text/html; charset=utf-8" /&gt;
  &lt;title&gt;Google Visualization API Sample&lt;/title&gt;
  &lt;script type="text/javascript" src="http://www.google.com/jsapi"&gt;&lt;/
script&gt;
  &lt;script type="text/javascript"&gt;
    google.load("visualization", "1", {packages: ["annotatedtimeline"]});

    function drawVisualizations() {
    ''')

    # create our visualization
    self.response.out.write('''new google.visualization.Query("http://
wattcher.appspot.com/visquery.json?user='''+
                                 account.email()
+'''&amp;bhours='''+str(historytimebegin)+'''").send(
          function(response) {
            new google.visualization.AnnotatedTimeLine(
                document.getElementById("visualization")).
                draw(response.getDataTable(), {"displayAnnotations": true});
          });
          ''')

    self.response.out.write('''}

    google.setOnLoadCallback(drawVisualizations);
  &lt;/script&gt;
&lt;/head&gt;
&lt;body style="font-family: Arial;border: 0 none;"&gt;
&lt;div id="visualization" style="width: 800px; height: 250px;"&gt;&lt;/div&gt;
&lt;/body&gt;
&lt;/html&gt;''')
```

The first part is pretty straight forward, get the user name or login. Then we will assume the user wants 1 last day of data, so set bhours and ehours. Then we literally just print out the code we copied from Google's Visualization sandbox, done!

Viz Viz Viz

The only thing I couldn't figure out is how to get 3 visualizations going on at once (last hour, day and week) with the above code. It just kinda broke. So for the triple view I had to use iframes :(

```
class VisualizeAll(webapp.RequestHandler):
  def get(self):

    # make the user log in if no user name is supplied
    if self.request.get('user'):
      account = users.User(self.request.get('user'))
    else:
       if not users.get_current_user():
         self.redirect(users.create_login_url(self.request.uri))
       account = users.get_current_user()

    self.response.out.write('''
&lt;h2&gt;Power usage over the last hour:&lt;/h2&gt;
&lt;iframe src ="graph?user=adawattz@gmail.com&amp;bhours=1" frameborder="0"
width="100%" height="300px"&gt;
  &lt;p&gt;Your browser does not support iframes.&lt;/p&gt;
&lt;/iframe&gt;

&lt;h2&gt;Power usage over the last day:&lt;/h2&gt;
&lt;iframe src ="graph?user=adawattz@gmail.com&amp;bhours=24"  frameborder="0"
width="100%" height="300px"&gt;
  &lt;p&gt;Your browser does not support iframes.&lt;/p&gt;
&lt;/iframe&gt;

&lt;h2&gt;Power usage over the last week:&lt;/h2&gt;
&lt;iframe src ="graph?user=adawattz@gmail.com&amp;bhours=168"  frameborder="0"
width="300%" height="500px"&gt;
  &lt;p&gt;Your browser does not support iframes.&lt;/p&gt;
&lt;/iframe&gt;

      ''')
```

Anyhow, it works just fine.

Timecodes!

The final thing that wont be reviewed here is how I got the date and times to be EST instead of UTC. As far as I can tell, its kind of broken and mysterious. Check the code () if you want to figure it out.

Resources

Other power monitoring projects!

Get some good ideas here!

- "Carbon Penance" a power monitor by Annina Rust that punishes the user ()
- Jason Winter's Real-Time power monitor ()
- Mazzini's project pushes data onto Pachube ()
- Pachube has lots of other projects! ()

- Furnace monitoring, using a DAQ board and phototransistor ()

Power monitoring products

Wanna just buy it?

- Black & Decker home electric meter watcher ()
- DIY KYOTO's power-clip Wattson is pretty ()
- CurrentCost ()

Websites & Software

- Myenergyusage.org () (one fellow upgrading his Wattson's software by hand)
- Wattzon.org () (social energy information)

Download

Software

- The most recent python scripts () this is what you want if you've built a tweet-a-watt and you want to get your project running! Click on Download Source in the top right corner
- You can also try this shareware prorgam that's all pretty and nice ()
- GoogleMeter API python scripts - ()Uncompress and place all the files into the same directory as the wattcher.py files, you'll need this if you want to upload your data to google powermeter

X-CTU profiles

- Receiver, connected to computer ()
- Transmitter, embedded in the Kill-a-Watt. () Change the unique ID if you have more than one!

Design files

All this stuff (other than the XBee library and the AppEngineAuth library, which are not written by me) is for you in the Public Domain! These are for debugging and design purpose and show how the project is put together. If you just want to "run the code" see the "software" above

- XBee library ()
- AppEngineAuth library for python ()
- Wattcher.py - graph just the mains data ()
- Wattcher.py - graph mains and wattage data ()
- Wattcher.py - Reports averages every 5 minutes ()
- Wattcher.py - Sends data to Wattcher Google App ()

For the latest code, please check the google code repository where you'll find any new code (). And hey, are you good at this sort of code? I could really use some help. It works OK but it could look and run much better so please commit patches and hang out on the forum!