adafruit learning system

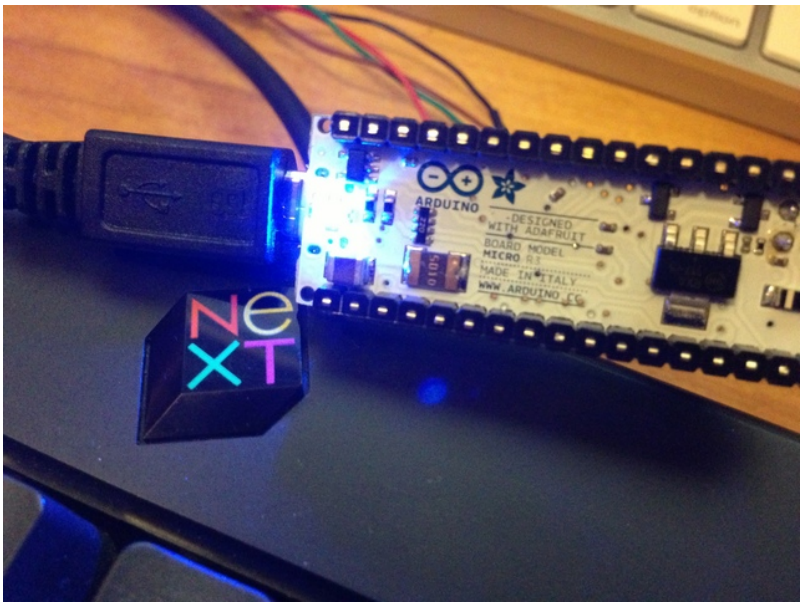
USB NeXT Keyboard with an Arduino Micro

Created by Ladyada

Last updated on 2013-06-24 10:15:41 PM EDT

Guide Contents

Overview



Ladyada and pt had an old NeXT (http://adafru.it/aTy) keyboard with a strong desire to get it running on a modern computer. These keyboards are durable, super clicky, and very satisfying to use! However, they are very old designs, specifically made for NeXT hardware, pre-ADB and pre-USB! That means you can't just plug the keyboard into an ADB or PS/2 port or PS/2 to USB converter (even though it looks similar). In fact, I have no idea what the protocol or pinout is named, so we'll just call it "non-ADB NeXT Keyboard"

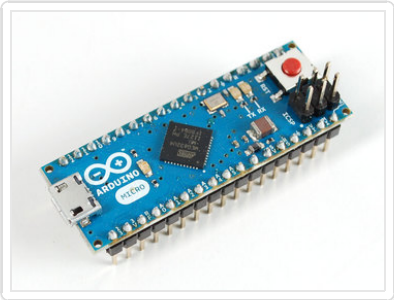
There is no existing adapter for sale, and no code out there for getting these working, so we spent a few days and with a little research we got it working perfectly using an Arduino Micro as the go between. Now this lovely black deck works like any other USB keyboard. Sure it weighs

more than our Macbook, but its worth it!

This project is for people with a little soldering and Arduino experience - we'll assume you've built some electronics before and have installed & uploaded code to an Arduino. If you haven't, check out many of the tutorials here at learn.adafruit.com for tons of ideas for practice!

Parts

Of course you'll need a **Non-ADB NeXT Keyboard** but there's some other parts you'll need!



First up, you'll need an Arduino Micro. If you can get one without headers that's great but as of this writing they're only available with headers. So we'll just clip them off. (http://adafru.it/1086)

You'll also need a MiniDIN-5 connector for the keyboard. We used this one from DigiKey (http://adafru.it/aX7)

A Micro-USB cable connects from the Micro to your computer (http://adafru.it/592)

Finally, this lovely Altoids tin will do a great job of holding it all

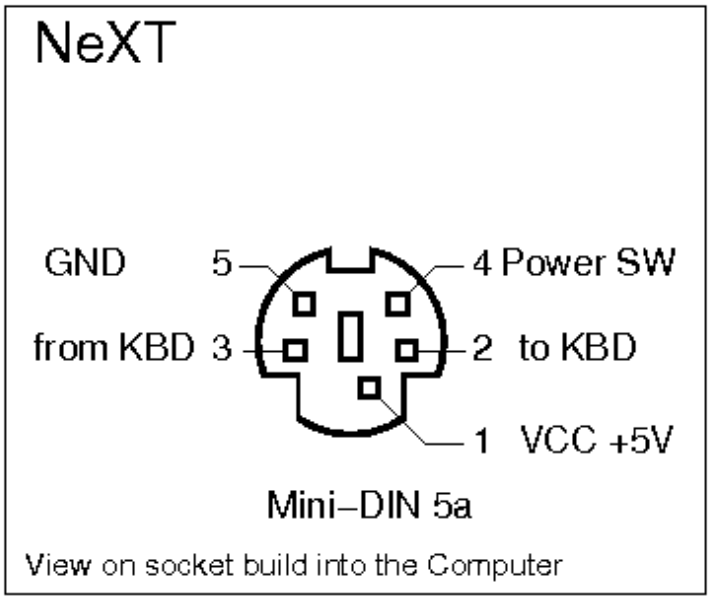
Tools!

You'll also want a soldering iron, solder (http://adafru.it/aTs), wire strippers (http://adafru.it/147), a rotary tool (to cut into the tin) and maybe some double-sided foam tape.

Research

The first thing to note is that the USB part (acting like a USB keyboard) is the easiest part of the project - there's already plenty of example code for how to do that with an Arduino Leonardo or Micro. The really tough part is figuring out how to read from the keyboard as it's not in any known or well documented protocol. The good news is whenever you're working with a really old technology, the computers back then were really slow and things weren't too complicated. Chances are whatever they did, it was meant to be simple and lightweight. Contrast this with a USB or Bluetooth or WiFi stack!

Our first stop is over at the awesome http://www.kbdbabel.org/conn/index.html (http://adafru.it/aTt) where the nice author has documented the pinout of the keyboard. This is great because we won't accidentally smash the electronics with the wrong voltage. Also, it gives us a hint of how to talk to it. Power and ground at 5V are nice, easy to work with voltages. There's an RX and TX pin so at least we don't have to deal with a bi-directional or differential signal (whew).

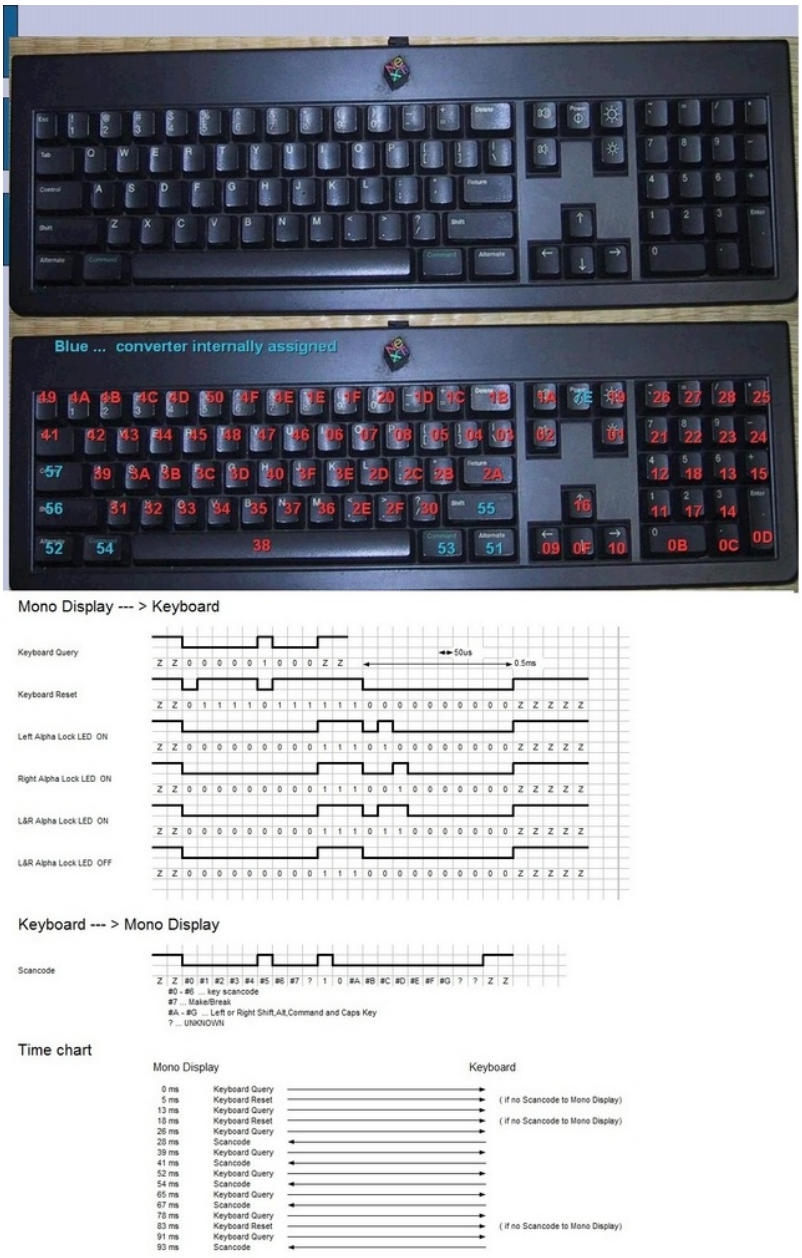


$Id: kbd_connector_next.fig,v 1.3 2007/12/09 20:23:55 akurz Exp $

Ok so now we can power it up. I applied +5V to VCC and ground to GND. I did see 5V on the "from KDB" pin, but unfortunately no actual data when keys were pressed. This means that the keyboard isnt 'dumb' - it expects some sort of clock or reset signal on the "to KBD" pin. While one could try to figure it out cold, its a lot of effort.

Ideally, we'd have a NeXT that we could plug the keyboard into and 'sniff' the traffic, that is the easiest way to do it. Unfortunately, we don't have one. We were in crisis! But then we kept searching and looking around (btw, searching for "next keyboard" is not a very efficient way to locate this brand of keyboard!) and we lucked out when we found a Japanese website of

serious keyboard enthusiast http://m0115.web.fc2.com/ (http://adafru.it/aTu) Its using frames so we weren't too optimistic we'd find a github repo, but after a lot of clicking we found the holy grail of NeXT timing information:



Yes! This is exactly what we need, not only does he include the timing diagram, but also the timeline for resetting and querying. 50microsecond timing is well within the abilities of a 16MHz microcontroller. Now we're ready to write code (see the next section for the code listing)

The only thing remaining was the scancode table. By this point I was 5 hours into this project and getting a little tired, when I realized that any operating system written for NeXT would have this all written up for me. In fact, there was an NetBSD port to NeXT and all the keyboard mapping data was there for me!

http://www.mirrorservice.org/sites/ftp.netbsd.org/pub/NetBSD/NetBSD-release-

6/src/sys/arch/next68k/d... (http://adafru.it/aTv)

Hooray!

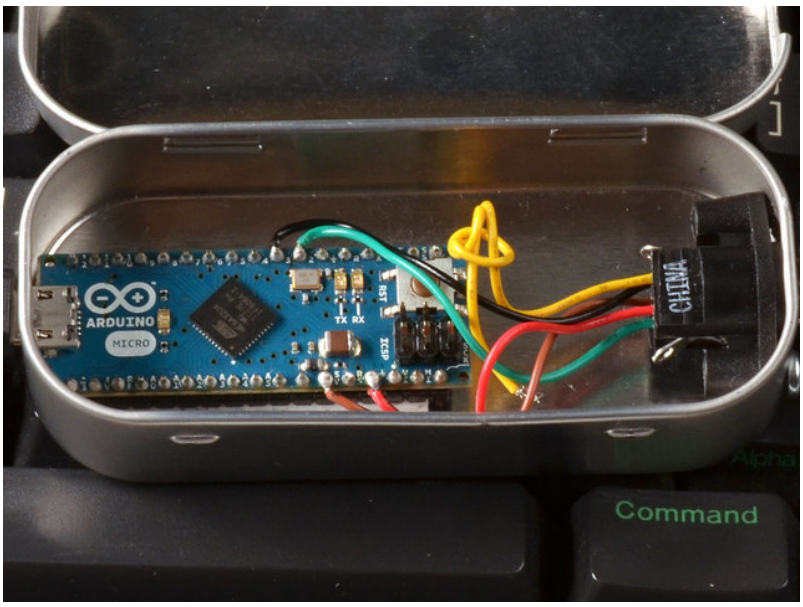
Wiring & Case

The wiring for this project is actually really easy - we used an Arduino micro to keep the size small. We just have to power the keyboard and connect the two data pins.

Watch out here, because the DIN wire colors do not correspond to classical wire coloring!

1. **Brown - VCC +5**
2. **Black - data to KBD**
3. Green - data from KBD
4. Yellow - power SW **(unused)**
5. **Red - Ground**

Solder the Brown wire to the +5V on the micro, the Red wire goes to the Ground pin. Then connect The green and black wires to two digital pins (I used 2 and 3)



Using this mint tin made for a nice case. A little dremel work, and the DIN connector would fit nicely at one end, the wiring inside, and the microUSB connector on the other end.

Use two small screws to attach the DIN connector. After the header pins have been cut off the Micro, it will fit nicely in the tin, be sure to use some thick foam tape to both hold down the board and keep the pins from shorting. A slot in the tin makes it easy to plug in a micro USB cable


Power
Command
Alternate

Code

You can grab the full Arduino sketch and the header files here at github. (http://adafru.it/aTw)Click DOWNLOADs and unzip the directory, then rename it USB_NeXT_Keyboard and place in your Arduino Sketch folder. Upload to a Leonardo or Micro.

This code is basically a 'word-for-word' interpretation of the Japanese timing diagram in the Research page, this is what allows us to reset the keyboard, query it for data and of course set the two LEDs!

```
// special command for setting LEDs
void setLEDs(bool leftLED, bool rightLED) {
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING *9);
  digitalWrite(KEYBOARDOUT, HIGH);
  delayMicroseconds(TIMING *3);
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING);

  if (leftLED)
     digitalWrite(KEYBOARDOUT, HIGH);
  else
     digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING);

  if (rightLED)
     digitalWrite(KEYBOARDOUT, HIGH);
  else
     digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING);
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING *7);
  digitalWrite(KEYBOARDOUT, HIGH);
}

void query() {
  // query the keyboard for data
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING *5);
  digitalWrite(KEYBOARDOUT, HIGH);
  delayMicroseconds(TIMING );
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING *3);
  digitalWrite(KEYBOARDOUT, HIGH);
}

void nextreset() {
  // reset the keyboard
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING);
  digitalWrite(KEYBOARDOUT, HIGH);
  delayMicroseconds(TIMING*4);
  digitalWrite(KEYBOARDOUT, LOW);
```

```
  delayMicroseconds(TIMING);
  digitalWrite(KEYBOARDOUT, HIGH);
  delayMicroseconds(TIMING*6);
  digitalWrite(KEYBOARDOUT, LOW);
  delayMicroseconds(TIMING*10);
  digitalWrite(KEYBOARDOUT, HIGH);
}
```

This is the bitbang-receiver code. It looks from a low pin transition, then delay()'s half a timing pulse, and reads 22 pulses out, stuffing one after the other into a 32 bit variable.

We use cli() and sei() to turn off interrupts for the best precision! Interrupts can garble data by changing the timing.

```
uint32_t getresponse() {
  // bitbang timing, read 22 bits 50 microseconds apart
  cli();
  while ( readkbd() );
  delayMicroseconds(TIMING/2);
  uint32_t data = 0;
  for (uint8_t i=0; i < 22; i++) {
     if (readkbd())
       data |= ((uint32_t)1 << i);
     delayMicroseconds(TIMING);
  }
  sei();
  return data;
}
```

We spend most of the loop query()ing the keyboard for new data. If there's no new information, we'll get the 0x200600 "idle" response, which we can ignore. If we don't get that response, we can print out the keycode (what key was pressed) in the debugging serial console

```
void loop() {
  digitalWrite(LED, LOW);
  delay(20);
  uint32_t resp;
  query();
  resp = getresponse();

  // check for a 'idle' response, we'll do nothing
  if (resp == 0x200600) return;

  // turn on the LED when we get real resposes!
  digitalWrite(LED, HIGH);

  // keycode is the lower 7 bits
  uint8_t keycode = resp & 0xFF;
  keycode /= 2;

#ifdef DEBUG
  Serial.print('['); Serial.print(resp, HEX);  Serial.print("] ");
  Serial.print("keycode: "); Serial.print(keycode);
#endif
```

There's a special exception for keys like SHIFT and ALT - these don't send a keycode, you have to check the higher bits to see what key was pressed - you can have multiple ones pressed at once! These are turned into USB keyboard presses of the matching 'modifier' keys. See http://arduino.cc/en/Reference/KeyboardModifiers (http://adafru.it/aTx) for more!

```
if (keycode == 0) {
  // modifiers! you can remap these here,
  // but I suggest doing it in the OS instead
  if (resp & 0x1000)
    Keyboard.press(KEY_LEFT_GUI);
  else
    Keyboard.release(KEY_LEFT_GUI);

  if (resp & 0x2000) {
    Keyboard.press(KEY_LEFT_SHIFT);
  } else {
    Keyboard.release(KEY_LEFT_SHIFT);
  }
  if (resp & 0x4000) {
    Keyboard.press(KEY_RIGHT_SHIFT);
  } else {
    Keyboard.release(KEY_RIGHT_SHIFT);
  }
  // turn on shift LEDs if shift is held down
  if (resp & 0x6000)
    setLEDs(true, true);
  else
    setLEDs(false, false);

  if (resp & 0x8000)
    Keyboard.press(KEY_LEFT_CTRL);
  else
    Keyboard.release(KEY_LEFT_CTRL);

  if (resp & 0x10000)
    Keyboard.press(KEY_RIGHT_CTRL);
  else
    Keyboard.release(KEY_RIGHT_CTRL);

  if (resp & 0x20000)
    Keyboard.press(KEY_LEFT_ALT);
  else
    Keyboard.release(KEY_LEFT_ALT);
  if (resp & 0x40000)
    Keyboard.press(KEY_RIGHT_ALT);
  else
    Keyboard.release(KEY_RIGHT_ALT);

  return;
}
```

This is where all the hard work really happens. We convert the keycode into an ascii code using the NetBSD keycode table - for some codes, we have to perform a special action because the

ascii code isn't the same as the code we have to send down the USB bus. For example Escape, Return, Backspace, and the Arrows. The Delete key doesn't actually exist on the NeXT so we mapped the 'lower volume' button to it since they're in the same location. Then we can look at the 3rd byte to see if its a key down or key up command and send that press or release!

```
  for (int i = 0; i< 100; i++) {
    if (nextkbd_keydesc_us[i*3] == keycode) {
      char ascii = nextkbd_keydesc_us[i*3+1];

#ifdef DEBUG
      Serial.print("--> ");      Serial.print(ascii);
#endif

      int code;
      switch (keycode) {
        case 73: code = KEY_ESC; break;
        case 13: code = KEY_RETURN; break;
        case 42: code = KEY_RETURN; break;
        case 27: code = KEY_BACKSPACE; break;
        case 22: code = KEY_UP_ARROW; break;
        case 15: code = KEY_DOWN_ARROW; break;
        case 16: code = KEY_RIGHT_ARROW; break;
        case 9: code = KEY_LEFT_ARROW; break;
        // remap the 'lower volume' key to Delete (its where youd expect it)
        case 2: code = KEY_DELETE; break;

        default: code = ascii;
      }
      if ((resp & 0xF00) == 0x400) {  // down press
#ifdef DEBUG
        Serial.println(" v ");
#endif
        Keyboard.press(code);
        break;
      }
      if ((resp & 0xF00) == 0x500) {
        Keyboard.release(code);
#ifdef DEBUG
        Serial.println(" ^ ");
#endif
        break;
      }
    }
  }
```